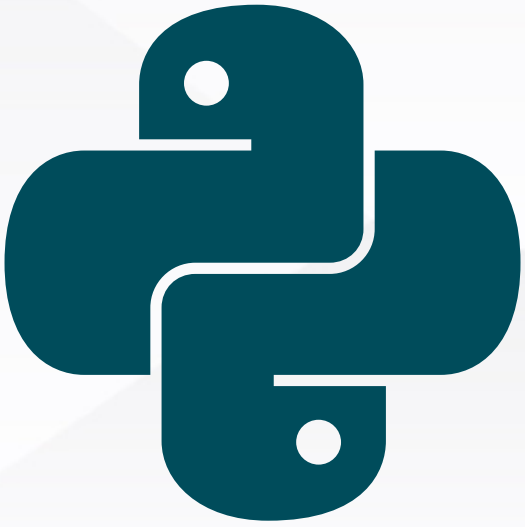
PYTHON PARA PRINCIPIANTES



DANIEL CORREA
PAOLA VALLEJO

Python Para Principiantes

Aprender a programar con Python de manera práctica y paso a paso

Daniel Correa – Paola Vallejo

Practical Books

Python Para Principiantes

por Daniel Correa y Paola Vallejo

Edición técnica: Ronald M. Martinod.

Revisora de código: Juliana Parra.
Revisor de código: Damián Duque.

Primera edición: Enero de 2023.
Actualización: V1.2.

Para mi padre José Fernando, quién trabajó arduamente toda su vida para sostener a su familia
- Daniel Correa

A mi familia, por su apoyo incondicional
- Paola Vallejo

Colaboradores

Acerca del autor

Daniel Correa es profesor, investigador, desarrollador de software y autor de múltiples libros de programación. Tiene un doctorado en informática. Es profesor de la Universidad EAFIT en Medellín, Colombia. Ha sido docente durante más de 10 años y coordina cursos de programación. Está interesado en arquitecturas de software, *frameworks* (como Laravel, Nest, Django, Express, Vue, React y Angular), desarrollo web y código limpio. Sigue a Daniel en Twitter en **@danielgarax**.

Acerca de la autora

Paola Vallejo es docente e investigadora. Actualmente, labora en la Universidad EAFIT en Medellín, Colombia. Imparte cursos de Programación básica e Ingeniería de Software. Ha sido docente durante más de 7 años. Está interesada en temas relacionados con arquitecturas de software, diseño de software y código limpio.

Acerca del editor técnico

Ronald M. Martinod es profesor asociado de la Universidad EAFIT desde 2015. Completó su formación en ingeniería mecánica, y posteriormente, obtuvo el título de doctor de la Universidad de Lorraine (Francia) en la Escuela de Ingeniería Informática y Matemática. Su experiencia investigativa la ha adquirido trabajando en proyectos de investigación que brindan apoyo a las organizaciones en la optimización de las políticas de mantenimiento y operación de redes de transporte urbano.

Acerca de la revisora de código

Juliana Parra es técnica en administración documental y trabaja en el área de informática en salud del hospital Pablo Tobón Uribe de Medellín. Apasionada por la lectura y por viajar.

Acerca del revisor de código

Damián Duque es estudiante de Ingeniería de Sistemas de la Universidad EAFIT con una gran pasión por el aprendizaje y el desarrollo de software (https://github.com/DamianDuque).

Tabla de contenido

Prefacio 7
Capítulo 01 – Introducción 11
Capítulo 02 – Computadoras y programación 17
Capítulo 03 – Python 25
Capítulo 04 – Hola mundo en Colab 29
Capítulo 05 – Variables enteras, flotantes y textos 41
Capítulo 06 – Entrada y salida de datos 53
Capítulo 07 – Condicionales simples 63
Capítulo 08 – Condicionales múltiples 77
Capítulo 09 – Ciclo while 91
Capítulo 10 – Textos 105
Capítulo 11 – Listas 119
Capítulo 12 – Diccionarios 133
Capítulo 13 – Ciclo for 147
Capítulo 14 – Funciones 157
Capítulo 15 – Contadores, acumuladores y banderas 175
Capítulo 16 – Archivos 183
Capítulo 17 – Librerías – Matplotlib 195
Capítulo 18 – Continúe su aprendizaje 203
Capítulo 19 – Solución de ejercicios 207

Prefacio

La programación está en todas partes. Tener habilidades de programación es algo indispensable en este mundo moderno, casi tan importante como saber escribir o hablar inglés. Actualmente, el "coding" o "programación" se ha convertido en el idioma de la "innovación".

Los programadores se han convertido en las estrellas de rock de nuestra época. Reconocemos personajes como Mark Zuckerberg, Bill Gates, o Elon Musk. Y no es un secreto que hoy en día es una de las profesiones mejor pagadas a nivel mundial.

En este libro descubriremos algunos de los misterios básicos de la programación, explicaremos algunas de las estructuras fundamentales y desarrollaremos cientos de piezas de código.

La gran diferencia entre este libro y otros similares, es que adoptaremos una estrategia que se rige por tres elementos fundamentales: (i) desarrollemos los conceptos con un enfoque **teórico/práctico**, (ii) explicaremos cada elemento **paso a paso**, y (iii) utilizaremos **imágenes** para explicar los elementos fundamentales.

Teórico/práctico: cada concepto que desarrollemos lo explicaremos con calma, de manera breve, con palabras que cualquier lector pueda entender (sin dar vueltas o utilizar palabras complejas o rimbombantes). Además, desarrollaremos múltiples piezas de código, ejemplos y ejercicios para que el lector practique y comprenda cómo aplicar cada elemento visto.

Paso a paso: iniciaremos por piezas de código cortas y simples. Además, cada instrucción que se desarrolle la explicaremos de manera adecuada. No supondremos que el lector mágicamente entiende todo.

Imágenes: para facilitar el entendimiento de múltiples conceptos y piezas de código (tanto simples como complejas), utilizaremos imágenes de apoyo. Utilizaremos diagramas, flujos de código paso a paso, y salidas por pantalla para que el lector verifique su progreso.

¿Para quién es este libro?

Este libro está diseñado para cualquier persona que quiera aprender a programar, puede ser utilizado por jóvenes, adultos, profesionales o profesores. Es un libro diseñado para principiantes o novatos.

El libro no requiere conocimientos previos en programación por parte del lector. Lo único que le recomendamos al lector, es que sepa leer, que conozca sobre aritmética básica (como sumar y

restar), que pueda utilizar una computadora, y que sepa seguir instrucciones. El resto, no lo tomaremos por sentado, y de ser necesario, lo explicaremos en este libro.

¿Qué cubre este libro?

Tal como lo indica el título, es un libro para principiantes. Este libro cubre elementos básicos y fundamentales de programación en Python. Incluyendo temas tales como: computadoras, programación, algoritmos, variables, condicionales, ciclos, funciones, listas, diccionarios, archivos y una breve introducción a librerías. Es un libro corto, diseñado para no abrumar al lector. Aunque podría haber abarcado más temas, se tratan únicamente los temas anteriores para explicarlos bien y en detalle.

Al finalizar la lectura del libro, el lector habrá adquirido conocimientos sobre elementos básicos en programación, que posteriormente podrá aplicar en proyectos más avanzados y que le permitirán adquirir nuevos conocimientos con mayor facilidad.

¿Qué experiencia previa tienen los autores?

Daniel y Paola tienen más de 18 años en conjunto de experiencia en docencia en programación. También han sido autores de otros libros de programación web. Durante estos años han enseñado a programar a miles de estudiantes de diferentes universidades.

En este libro recopilamos y plasmamos muchas de las estrategias utilizadas para enseñar a programar a diferentes audiencias de estudiantes.

Comentarios generales

Si tiene preguntas sobre cualquier aspecto de este libro, envíenos un correo electrónico a practicalbooksco@gmail.com y por favor mencione el título del libro en el asunto de su mensaje. Adicionalmente, **tenemos dos capítulos extra digitales que no están incluidos con el libro, pero que obsequiaremos a cualquier lector que nos escriba y los solicite**.

Errores en el libro (fe de erratas)

Aunque el libro ha sido revisado por múltiples personas, y hemos tomado todas las precauciones para garantizar su calidad, los errores ocurren. Si encuentra algún error en este libro, le pedimos y agradecemos que por favor nos escriba a practicalbooksco@gmail.com informando el error a corregir.

Piratería

Si encuentra algún enlace o copia ilegal del libro en Internet o en algún sitio, le pedimos y agradecemos que por favor nos proporcione ese enlace o información al correo practicalbooksco@gmail.com.

Obtener el libro en formato PDF

Si adquirió la versión impresa de este libro por Amazon o la versión Kindle, envíenos un correo electrónico a practicalbooksco@gmail.com con una captura de pantalla de la adquisición por Amazon y le obsequiaremos la versión digital PDF del libro (**que incluye todas las imágenes a color**).

Obtener actualizaciones del libro

Si desea recibir actualizaciones de este libro, envíenos un correo electrónico a practicalbooksco@gmail.com, además, lo suscribiremos a nuestra lista de correos.

Capítulo 01 – Introducción

Empezaremos nuestro viaje a aprender a programar con Python. En este libro, aprenderemos muchos de los elementos claves que se necesitan para programar en una computadora, y muchos de los elementos principales que componen Python.

Antes de iniciar con esos elementos, en este capítulo explicaremos cómo está diseñado este libro y cómo utilizarlo.

A continuación, desarrollaremos las siguientes secciones:

1. Capítulos del libro.
2. Cómo leer este libro.
3. Repositorio del libro.
4. Hoja de trucos.

1.1. Capítulos del libro

Veamos cómo se divide este libro y los elementos que se desarrollan en cada capítulo.

- **Capítulo 01 – Introducción.** Presenta una introducción al libro y a los elementos que lo componen.
- **Capítulo 02 – Computadoras y programación.** Introduce a los conceptos de computadoras, programación, lenguajes de programación y algoritmos. Además, rompe algunos mitos en cuanto a la programación.
- **Capítulo 03 – Python.** Introduce al lenguaje de programación Python. Muestra cómo se compara Python con otros lenguajes de programación y describe algunas aplicaciones que se podrían desarrollar con Python.
- **Capítulo 04 – Hola mundo en Colab.** Describe cómo usar la herramienta Colab para codificar en Python.
- **Capítulo 05 – Variables enteras, flotantes y texto.** Desarrolla el concepto fundamental de variables y presenta tres tipos de variables: enteras, flotantes y textos. Además, explica cómo mostrar información en la pantalla de la computadora.
- **Capítulo 06 – Entrada y salida de datos.** Describe cómo ejecutar operaciones de entrada y salida de datos. Hace énfasis en las funciones *print* (para imprimir información) e *input* (para capturar información por teclado).
- **Capítulo 07 – Condicionales simples.** Presenta conceptos básicos para manejo de condiciones en programación. Y luego muestra cómo implementar un condicional simple basado en la estructura *if-then* en Python.
- **Capítulo 08 – Condicionales múltiples.** Desarrolla otras estructuras condicionales tales como: *if-then-else*, *if-then-elif-else*, y condicionales anidados.

- **Capítulo 09 – Ciclo while.** Introduce el concepto de ciclos en programación y muestra cómo definir y utilizar un ciclo *while*.
- **Capítulo 10 – Textos.** Explica la importancia de la manipulación de textos en programación. Presenta las propiedades de los textos y algunas operaciones básicas y métodos disponibles.
- **Capítulo 11 – Listas.** Desarrolla el concepto de listas y muestra cómo implementar algunas operaciones básicas con listas.
- **Capítulo 12 – Diccionarios.** Desarrolla el concepto de diccionarios y muestra cómo implementar algunas operaciones básicas con diccionarios.
- **Capítulo 13 – Ciclo for.** Describe otro tipo de ciclo en Python llamado ciclo *for*, y muestra cómo utilizarlo para iterar sobre diferentes tipos de variables en Python.
- **Capítulo 14 – Funciones.** Detalla el comportamiento de las funciones en Python y muestra cómo crear funciones con múltiples variaciones.
- **Capítulo 15 – Contadores, acumuladores y banderas.** Describe tres operaciones comunes en programación: contar, acumular y realizar verificaciones. Para ello muestra cómo usar contadores, acumuladores y banderas.
- **Capítulo 16 – Archivos.** Presenta cómo almacenar información en archivos. Muestra cómo manipular diferentes tipos de archivos y analizar la información contenida en ellos.
- **Capítulo 17 – Librerías – Matplotlib.** Presenta una introducción a las librerías en Python y muestra cómo utilizar la librería Matplotlib (especializada en la generación de gráficos).
- **Capítulo 18 – Continúe su aprendizaje.** Presenta recursos adicionales para continuar el aprendizaje en programación y en Python.
- **Capítulo 19 – Solución de ejercicios.** Contiene la solución a los ejercicios planteados en este libro.

1.2. Cómo leer este libro

Veamos, a continuación, unas cuantas sugerencias que ayudarán a entender mejor los diferentes elementos que se desarrollan en el libro, y que facilitarán su lectura.

Lectura en orden

Si es la primera vez que programa le recomendamos leer este libro en orden. El libro está diseñado para ir aprendiendo diferentes conceptos y elementos en orden. Estos conceptos y elementos se reutilizarán en capítulos posteriores, esto quiere decir que, por ejemplo, para entender y disfrutar el "Capítulo 13 – Ciclo for", deberá comprender casi todos los capítulos anteriores. Ya que, dentro de este capítulo, utilizamos diversos elementos como listas, diccionarios, variables, y textos que se desarrollan en capítulos anteriores.

Aprender haciendo

Uno de los elementos clave de este libro es "Aprender haciendo", inspirado de uno de los autores de libros favoritos de Daniel y Paola, llamado Greg Lim. **En este libro aprenderemos a programar, programando**. Este libro presenta más de 100 piezas de código independientes a desarrollar. Le recomendamos que las codifique manualmente para que vaya adquiriendo habilidades de programación, y reconociendo fácilmente cada elemento utilizado. Cada una de esas piezas codifíquela manualmente, léala, analícela con detalle y ejecútela.

TIP: Aprender haciendo: "El ser humano es un buscador de conocimiento insaciable; sin embargo, no aprende lo que oye, lee, memoriza o estudia… sino lo que practica" (tomado de: *2019 - Samsó, R. - El Poder de la Disciplina*).

Realización de ejercicios

Antes de finalizar muchos de los capítulos encontrará un conjunto de ejercicios simples. Le recomendamos completar esos ejercicios antes de continuar con el próximo capítulo. Ya que de esta manera validará si comprendió los elementos vistos en cada capítulo, y si adquirió las habilidades esperadas. La solución a todos los ejercicios la encontrará en el último capítulo de este libro.

1.3. Repositorio del libro

En la mayoría de los capítulos, programaremos piezas de código en Python. Todas las piezas de código elaboradas en este libro se pueden encontrar en el repositorio del libro en el siguiente enlace: https://github.com/PracticalBooks/Python-Para-Principiantes. Si posteriormente realizamos algunas modificaciones al código, esas modificaciones se verán reflejadas en el repositorio.

Dudas y preguntas

Si en cualquier momento tiene una pregunta acerca de cualquier aspecto o elemento de este libro, o quieres discutir acerca de algo, le recomendamos que utilice la zona de discusión del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/discussions (ver Figura 1-1). De esta manera puede aprender de preguntas que hayan realizado otros lectores y los demás lectores y autores del libro pueden aprender de usted. Además, cualquier persona en la comunidad podría ayudarle a responder sus preguntas. **Nota:** para ver el botón de *"New discussion"*, debe tener una cuenta de GitHub.

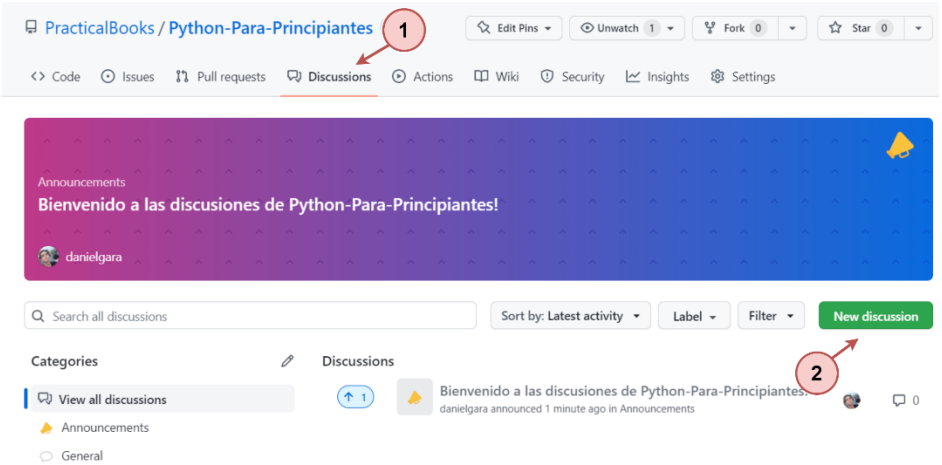


Figura 1-1. Zona de discusión del repositorio del libro.

Adicionalmente, puede escribir sus preguntas o sugerencias al correo practicalbooksco@gmail.com. Por favor, mencione el nombre del libro en el asunto del mensaje a enviar.

1.4. Hoja de trucos

Como material adicional al libro, encontrará la "Hoja de Trucos" de Python en la raíz del repositorio del libro https://github.com/PracticalBooks/Python-Para-Principiantes. Esta "Hoja de Trucos" fue elaborada por los autores de este libro, y contiene gran parte de los elementos que desarrollaremos en el libro.

Le sugerimos que se tome un momento para descargar la "Hoja de Trucos" y analizarla. Esta se compone de dos imágenes que contienen múltiples fragmentos de códigos separados por categorías. La primera imagen la puede descargar de este enlace: https://github.com/PracticalBooks/Python-Para-Principiantes/blob/main/Hoja-de-Trucos-Parte-1.png y la segunda imagen, de este enlace: https://github.com/PracticalBooks/Python-Para-Principiantes/blob/main/Hoja-de-Trucos-Parte-2.png.

Muchos de estos fragmentos contienen trucos (desde el T01 hasta el T85), los cuales a su vez contienen recuadros para seleccionar o marcar. Al final de la mayoría de los capítulos de este libro, encontrará instrucciones para identificar los trucos de la "Hoja de Trucos" que debe marcar (indicando los fragmentos de código que ya se explicaron tanto teóricamente como de manera práctica, ver Figura 1-2).

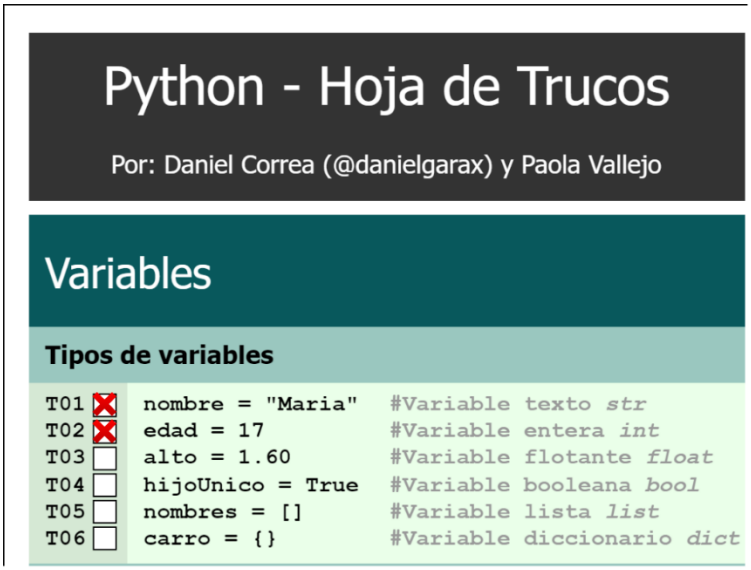


Figura 1-2. Fragmento de la Hoja de Trucos, donde el lector marca dos trucos vistos.

Si lo desea, imprima la “Hoja de Trucos” y vaya marcándola con lápiz o lapicero. O si lo prefiere, descargue la imagen y vaya marcándola de manera digital con programas como “Paint”, “Word” o cualquier otro. En total son 85 trucos o fragmentos de código que debería comprender al finalizar este libro.

Resumen

En este capítulo vimos una introducción al libro. Mostramos cómo está dividido el libro, y algunos elementos esenciales que se crearon para facilitar su lectura y el aprendizaje de los elementos que se enseñarán.

¡Ahora empecemos nuestro viaje al mundo de la programación!

Capítulo 02 – Computadoras y programación

¿Qué es una computadora? ¿Qué es programación? ¿Programar es difícil? En este capítulo explicaremos algunos de los fundamentos detrás de la programación.

A continuación, desarrollaremos las siguientes secciones:

1. Rompiendo mitos.
2. Computadoras.
3. Lenguajes de programación.
4. Algoritmos.

2.1. Rompiendo mitos

Antes de empezar a programar y conocer algunos conceptos clave de programación, es importante romper algunos mitos sobre la programación y los programadores (basado en: *2020 - Sweigart, A. - Automate the boring stuff with Python, 2nd Edition*). Veamos algunos de estos mitos.

¿Solo los ingenieros de software o ingenieros de sistemas deberían programar?

Esto es totalmente falso. Hoy en día cualquier persona (sin importar su edad, género, u ocupación) debería aprender, aunque sea un poco de programación. Muchos colegios ya lo enseñan desde secundaria e incluso primaria. La programación está en todos lados, en un celular, en un televisor, en un reloj, en un dron, en un carro, y en una lavadora. Y no solo eso, está en muchas de las cosas que nos rodean, en los semáforos, pantallas de publicidad, cámaras de vigilancia, y muchas más.

¿Los programadores necesitan saber mucha matemática?

Esto también es falso. La mayoría de la programación no requiere más matemática que la aritmética básica (sumas, restas, multiplicaciones y divisiones). Incluso no se preocupe si no sabe multiplicar o dividir, si por lo menos sabe utilizar una calculadora o Google, puede realizar muchos tipos de programas diferentes.

¿Aprender a programar es difícil?

Hoy en día aprender a programar es mucho más fácil que antes: algunos lenguajes de programación se han vuelto más simples y existen miles de recursos diferentes para utilizar según

los diferentes gustos, como videos en YouTube o TikTok, tutoriales en Internet, libros, juegos, aplicaciones, y muchos otros.

Y como lo mencionamos anteriormente, en este libro lo llevaremos de la mano, paso a paso, explicando todo en detalle y sin utilizar ejemplos complejos. Alguna vez la esposa de Daniel realizó un curso de programación básica donde el profesor en la primera clase presentó ejemplos con ecuaciones diferenciales. Le prometemos que ese no será el caso en este libro.

¿La programación solamente me sirve si trabajo como programador?

Esto es falso. Supongamos que usted es un abogado. ¿Debería entonces dejar su profesión de abogado y dedicarse a la programación? Probablemente no. Supongamos que un abogado tiene un caso donde necesita verificar cientos de documentos físicos y buscar en cuáles de esos documentos se menciona el uso de un medicamento particular. El abogado podría leer uno a uno cada documento. O podría escanear los documentos y hacer un pequeño código en Python que extraiga el texto de esos documentos y busque en ellos automáticamente el nombre del medicamento particular. La primera opción le podría llevar días, semanas o meses. La segunda unos cuantos minutos.

La programación toma las habilidades del abogado y las amplifica mucho más que las de sus colegas (le da un superpoder).

¿Si aprendo programación me volveré millonario y me compraré mi propio lambo (Lamborghini)?

Esperamos que así sea. Si ese es su caso, nos lo hace saber y ojalá invite a los autores de este libro a dar un paseo en su lambo. Pero veamos algunos números más realistas (según la oficina de estadísticas laborales de los Estados Unidos https://www.bls.gov/ooh/computer-and-information-technology/): (i) el salario promedio anual de diferentes ocupaciones relacionadas con computación y programación fue de 97.430 dólares, mucho más alto que el salario promedio anual de todas las ocupaciones (45.760 dólares). (ii) Se proyecta que las ocupaciones relacionadas con computación y programación crezcan un 15% de 2021 a 2031. Lo cual es mucho más rápido que el promedio de todas las ocupaciones.

TIP: Si requiere un poco de motivación acerca de la importancia de la programación, le recomendamos ver en YouTube el video corto titulado "El Super Poder de Programar". Allí famosos de distintas profesiones realizan una gran reflexión acerca de la programación.

Ahora que ya rompimos algunos mitos, empecemos a entender un poco mejor el mundo de la programación.

2.2. Computadoras

Una **computadora** es un dispositivo electrónico que almacena y procesa datos. Según en esa definición, nuestros celulares también son computadoras. Utilizamos las computadoras, por ejemplo, para guardar las fotos de nuestros últimos viajes, para colocar recordatorios del medicamento que tenemos que tomar, o para conversar con nuestros amigos. Allí estamos almacenando y procesando datos.

Una computadora la podríamos manipular para que haga ciertas tareas por nosotros. El problema es que las computadoras no entienden los idiomas humanos*. Y por eso, si queremos manipularlas para que hagan algo por nosotros debemos hablar algún idioma que ellas entiendan, y esto es lo que se conoce como un **lenguaje de programación**.

***Nota:** hoy en día existen asistentes como Alexa, Siri o el asistente de Google, a los cuales les podemos "hablar" y "solicitar" diferentes acciones. Internamente, estos asistentes fueron programados con lenguajes de programación. Y aunque son muy útiles, todavía tienen limitaciones. Por ejemplo, un abogado hoy en día no puede decirle a Alexa que por favor le analice los cientos de documentos físicos y le indique donde aparece el medicamento a buscar. Y es por eso, que para muchas tareas que queramos realizar, todavía debemos aprender a comunicarnos por medio de los lenguajes de las computadoras (lenguajes de programación).

2.3. Lenguajes de programación

Programar es definir una serie de instrucciones para que la computadora las siga, y haga algo por nosotros. Como ya vimos, para definir esas instrucciones debemos utilizar un lenguaje de programación.

Existen diferentes tipos de lenguajes de programación, veamos rápidamente tres categorías principales.

Lenguaje de máquina

Es el idioma nativo de una computadora. Este lenguaje está hecho en forma de código binario (unos y ceros). La Figura 2-1 muestra un programa (en lenguaje de máquina) que suma los números 1234 y 4321. Esta lista de unos y ceros contiene todos los comandos y datos necesarios

para completar esa tarea. La columna derecha es la continuación de la columna izquierda (tomado de: *1997 - Smith, S. W. - The scientist and engineer's guide to digital signal processing*).

```
10111001        00000000
11010010        10100001
00000100        00000000
10001001        00000000
00001110        10001011
00000000        00011110
00000000        00000010
10111001        00000000
11100001        00000011
00010000        11000011
10001001        10100011
00001110        00000100
00000010        00000000
```

Figura 2-1. Código binario que representa la suma de dos números (tomado de *Smith, S. W.*).

Lenguaje de ensamblador

Programar solo con unos y ceros es muy tedioso. Hasta el mago Mandrake se hubiera aburrido y cansado. Pero afortunadamente, a inicios de la era de la computación surgieron los lenguajes de ensamblador. Un lenguaje de ensamblador utiliza palabras descriptivas cortas para representar un conjunto de instrucciones. Luego, esas palabras e instrucciones se traducen a lenguaje de máquina. Una limitación de los lenguajes de ensamblador es que son dependientes de la máquina donde se crean las instrucciones. Esto quiere decir, que el código que yo haga probablemente no funcione en otra máquina con especificaciones diferentes. El siguiente código muestra cómo sumar los números 1234 y 4321 en lenguaje de ensamblador (tomado de: *1997 - Smith, S. W. - The scientist and engineer's guide to digital signal processing*).

Analizar

```
MOV CX,1234
MOV DS:[0],CX
MOV CX,4321
MOV DS:[2],CX
MOV AX,DS:[0]
MOV BX,DS:[2]
ADD AX,BX
MOV DS:[4],AX
```

Lenguaje de alto nivel

En la década de 1950 surgió una nueva generación de lenguajes de programación conocidos como "lenguajes de alto nivel". Estos lenguajes, a diferencia de los lenguajes de ensamblador, son independientes de la máquina. Esto quiere decir que puede codificar su programa y compartirlo con sus amigos y que ellos lo prueben sin problema en computadoras con especificaciones diferentes a la computadora en la cual fue desarrollado. Los lenguajes de alto nivel son similares al idioma inglés y son más fáciles de aprender y de utilizar. Normalmente, se enseña a programar en este tipo de lenguajes. Python, por ejemplo, es un lenguaje de alto nivel. El siguiente código muestra cómo sumar los números 1234 y 4321 utilizando una instrucción en lenguaje de alto nivel (en este caso Python).

Analizar

```
suma = 1234 + 4321
```

2.4. Algoritmos

Para finalizar este capítulo veamos qué son los algoritmos y qué relación tienen con programación.

Un **algoritmo** es una lista finita de instrucciones claras, ordenadas y precisas que describen un procedimiento para resolver un problema o realizar un cálculo. Los algoritmos se pueden diseñar con diagramas de flujo, con listas de pasos en un documento, o con pseudocódigo (que es escribir texto parecido al código a desarrollar). Y finalmente, un algoritmo se puede implementar en diferentes lenguajes de programación.

Lo que se recomienda es que antes de empezar a programar, se realice el diseño del algoritmo. Esto permitirá identificar mejor el problema a resolver y plantear diferentes alternativas. La Figura 2-2 muestra una estrategia para trabajar con algoritmos.

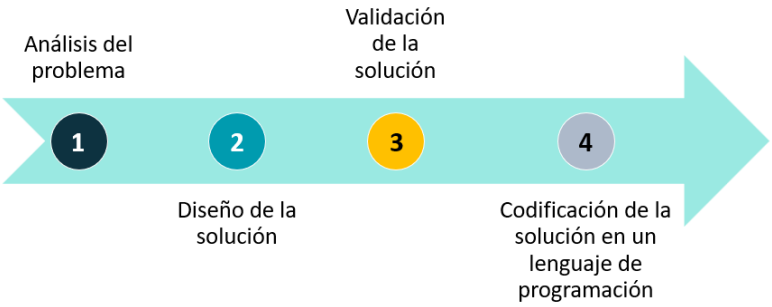


Figura 2-2. Estrategia de trabajo con algoritmos.

Apliquemos la estrategia de trabajo con algoritmos con un ejemplo específico. Supongamos que necesitamos “calcular la edad promedio de los cuatro trabajadores de una oficina”.

1) Análisis del problema

Consiste en responder una serie de preguntas aplicadas al problema a resolver o al cálculo a realizar. ¿Cuál es el objetivo buscado? ¿Cuáles son los datos de entrada? ¿Qué cálculos o procesos deben llevarse a cabo? ¿Cuáles son los datos de salida?

Si lo aplicamos al ejercicio de la oficina tendríamos que:

Objetivo: Calcular el promedio de edad de los trabajadores de una oficina.
Datos de entrada: Edad del trabajador 1, edad del trabajador 2, edad del trabajador 3, y edad del trabajador 4.
Procesos/cálculos: Sumar las edades de los trabajadores 1, 2, 3 y 4. Y dividir el total de la suma entre 4.
Datos de salida: Promedio de edad de los 4 trabajadores.

2) Diseño de la solución

Consiste en realizar un diseño del algoritmo que tratará de solucionar el problema o realizar el cálculo requerido. Para este diseño utilizamos la información recopilada en el paso anterior, y creamos diagramas de flujo, listas de pasos en un documento, o pseudocódigo.

La Figura 2-3 muestra dos formas de representar el diseño del algoritmo que nos servirá como base para solucionar el problema anterior.

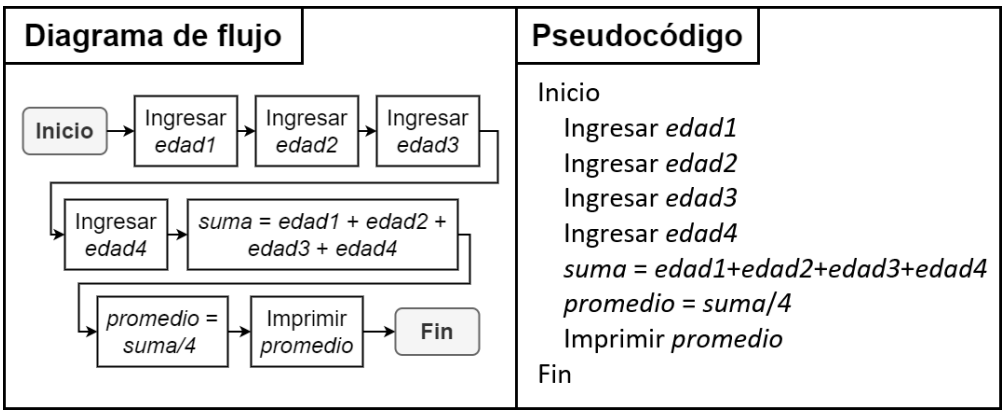


Figura 2-3. Diagrama de flujo y pseudocódigo del ejercicio.

3) Validación de la solución

Consiste en realizar una prueba o un conjunto de pruebas para verificar que el diseño del algoritmo es correcto. Estas pruebas se pueden realizar con lápiz y papel, calculadora o Excel, entre otros. La Figura 2-4 muestra cómo utilizar Excel para probar nuestro algoritmo.

	A	B
1	edad1	21
2	edad2	38
3	edad3	40
4	edad4	28
5	suma	127
6	promedio	31.75

Figura 2-4. Validación del algoritmo en Excel.

4) Codificación de la solución en un lenguaje de programación

Consiste en llevar el diseño del algoritmo a código real. El siguiente código muestra el ejercicio codificado en Python, y su ejecución se ve en la Figura 2-5. Por el momento solo analícelo. En capítulos posteriores le enseñaremos a codificar este tipo de programas y a probarlos.

Analizar

```
edad1 = 21
edad2 = 38
edad3 = 40
edad4 = 28
suma = edad1+edad2+edad3+edad4
promedio = suma/4
print(promedio)
```

```
31.75
```

Figura 2-5. Ejecución del código anterior.

Resumen

En este capítulo aprendimos los elementos fundamentales de la programación. Rompimos algunos mitos en cuanto a la programación. Aprendimos conceptos básicos sobre computadoras, programación, lenguajes de programación y algoritmos.

En el siguiente capítulo explicaremos los conceptos básicos sobre Python.

Capítulo 03 – Python

Python es uno de los lenguajes de programación más populares y utilizados a nivel mundial. En este capítulo explicaremos qué es Python y qué programas podríamos desarrollar si utilizamos Python.

A continuación, desarrollaremos las siguientes secciones:

1. Introducción.
2. Python comparado con otros lenguajes.
3. Tipos de desarrollos con Python.

3.1. Introducción

Python es un lenguaje de programación de alto nivel (https://www.python.org/). Fue creado a finales de los años ochenta por Guido van Rossum. El nombre de Python proviene de la afición de su creador por el grupo de humoristas británicos llamado “Monty Python”, y no por la “serpiente” como comúnmente se cree.

Python posee una licencia de código abierto, que, entre otras cosas, permite que los programadores puedan modificar su código, utilizarlo y/o redistribuirlo libremente sin tener que pagar al autor original.

Python se caracteriza por ser un lenguaje de programación amigable y fácil de aprender.

3.2. Python comparado con otros lenguajes

Muchos de los programadores utilizamos una plataforma llamada GitHub para almacenar y administrar el código de los programas que desarrollamos. En octubre de cada año, GitHub presenta un análisis de la información recopilada por este sitio, en un micrositio llamado Octoverse (https://octoverse.github.com/). Allí se presentan resultados sobre los millones de programadores que usan la plataforma, las regiones más activas, los lenguajes de programación más utilizados, los proyectos de código abierto más populares, y muchos otros resultados interesantes.

En la versión 2022 de Octoverse, encontramos los siguientes resultados en cuanto a los lenguajes más utilizados en la plataforma (ver Figura 3-1). Allí vemos que durante los últimos años Python ha estado entre los lenguajes más utilizados, ubicándose en 2022 en la segunda posición por debajo de JavaScript.

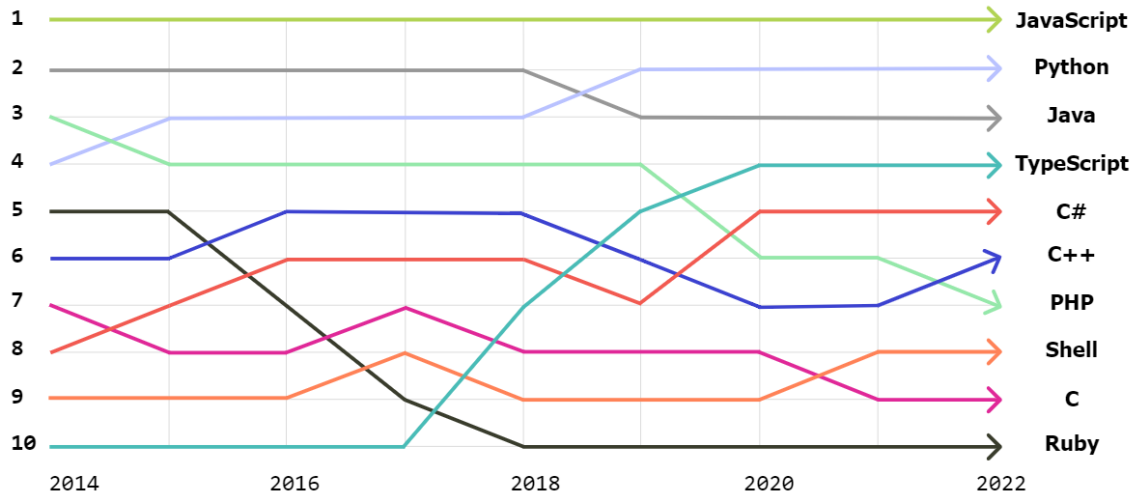


Figura 3-1. Top lenguajes de programación más utilizados en GitHub en los últimos años.

Pero ¿por qué Python es tan popular para aprender a programar?

Comparemos Python con Java a través de la implementación de un “Hola Mundo”, el ejemplo más simple que encontrará en cualquier lenguaje de programación. El objetivo es mostrar en la pantalla de la computadora el mensaje “Hola Mundo”.

El siguiente código muestra cómo programar un “Hola Mundo” en Java.

Analizar

```
public class Principal {
  public static void main(String[] args) {
     System.out.println("Hola Mundo");
  }
}
```

Y el siguiente código muestra cómo programar un “Hola Mundo” en Python.

Analizar

```
print("Hola Mundo")
```

Quizás ahora entienda por qué Python es considerado un lenguaje amigable y fácil de aprender.

3.3. Tipos de desarrollos con Python

¿Qué tipo de aplicaciones podríamos desarrollar con Python? Python provee un repositorio de paquetes de código abierto llamado "PyPI", donde programadores alrededor del mundo contribuyen desarrollando diferentes módulos que se pueden usar para desarrollar diferentes tipos de aplicaciones. Utilizando Python y utilizando algunos de estos módulos podríamos:

- Desarrollar aplicaciones web.
- Automatizar procesos o actividades (llamados *scripts*).
- Realizar análisis de datos.
- Desarrollar videojuegos.
- Extraer información de internet (llamado *web scrapping*).
- Manipular dispositivos IoT (cómo drones, sistemas embebidos o robots).
- Desarrollar aplicaciones de visión por computador (como reconocimiento automático de imágenes o de rostros).

Y la lista continúa y continúa.

El sitio web https://www.python.org/about/success presenta una lista de historias de éxito de Python en la vida real en dominios como artes, aviación, comercio digital, bioinformática, clima y muchos otros.

Resumen

En este capítulo aprendimos algunos elementos básicos de Python. Comparamos Python con otros lenguajes de programación. Y aprendimos algunos tipos de aplicaciones que podríamos desarrollar utilizando Python y sus diferentes módulos.

En el siguiente capítulo utilizaremos una herramienta llamada Colab, para codificar nuestro primer programa en Python.

Capítulo 04 – Hola mundo en Colab

Existen múltiples herramientas que podemos utilizar para programar en Python. En este libro utilizaremos Colab, un producto de Google que nos permite programar código Python desde nuestro navegador. En este capítulo aprenderemos a utilizar Colab y crearemos nuestros primeros programas.

A continuación, desarrollaremos las siguientes secciones:

1. Introducción.
2. Activación de Colab.
3. "Hola Mundo" en Colab.
4. Personalización de Colab.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo04. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

4.1. Introducción

Si queremos desarrollar códigos en Python tenemos múltiples opciones. Por ejemplo, podríamos descargar e instalar Python en nuestras computadoras desde su sitio web oficial (https://www.python.org/downloads/), podríamos usar el sitio web de Replit para programar en línea (https://replit.com/languages/python3), o podríamos usar cualquier otra herramienta. Cada una de estas opciones tiene sus ventajas y desventajas.

En este libro utilizaremos Google Colab o simplemente **Colab**, la cual es una herramienta creada por Google que permite a cualquier usuario escribir y ejecutar código Python en el navegador. Para utilizar Colab necesitamos contar con una cuenta de Google, y luego, desde Google Drive podremos crear documentos Colab donde programaremos en Python.

Entre las ventajas que tenemos al utilizar Colab con respecto a otras herramientas encontramos:

- **Cero instalación de programas:** para programar con Colab no necesitamos instalar nada. Simplemente, necesitamos una conexión a internet y una cuenta de Google.
- **Almacenamiento de datos en Internet:** dado que Colab se conecta con Google Drive, todos los programas que creemos se almacenarán automáticamente en Internet (en Google Drive).
- **Compartir nuestros programas:** los documentos de Colab se pueden compartir igual que cualquier archivo de Documento u Hoja de cálculo de Google.

A continuación, veremos cómo activar Colab. Pero recuerde que antes debe crear una cuenta de Google (si es que no cuenta con una). Para crear una cuenta de Google debe: (1) acceder a https://www.google.com/intl/es/account/about/, (2) dar clic en "Crear una cuenta" (ver Figura 4-1) y completar todos los pasos y la información que allí se solicita.

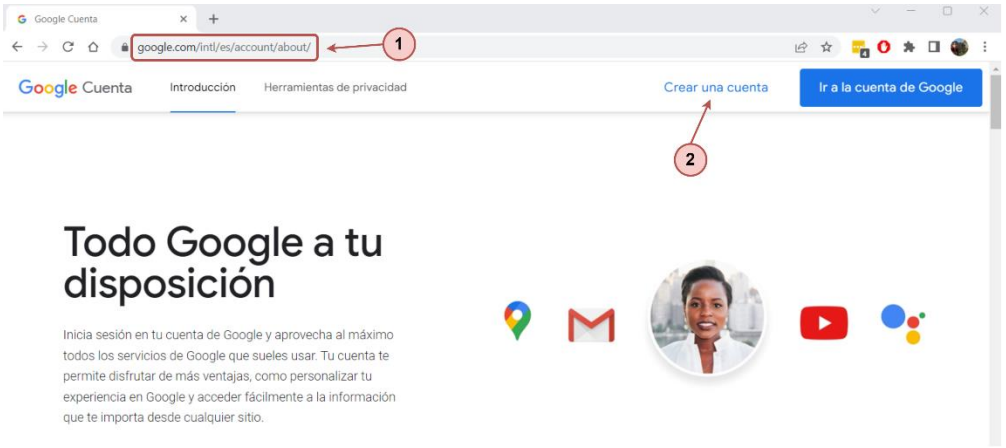


Figura 4-1. Sitio web para crear una cuenta de Google.

4.2. Activación de Colab

Importante: los siguientes pasos y capturas de pantalla pueden variar un poco debido a actualizaciones del sitio web de Google o de Colab. Por lo tanto, recuerde que, si tiene algún problema con la activación de Colab, puede buscar en Google o puede dejarnos un mensaje en la "zona de discusión" del repositorio del libro.

A continuación, complete los siguientes pasos para activar Colab con su cuenta de Google.

P1) Ingrese a Google Drive (https://drive.google.com/drive/my-drive). Recuerde que debe estar conectado con su cuenta de Google (ver Figura 4-2).

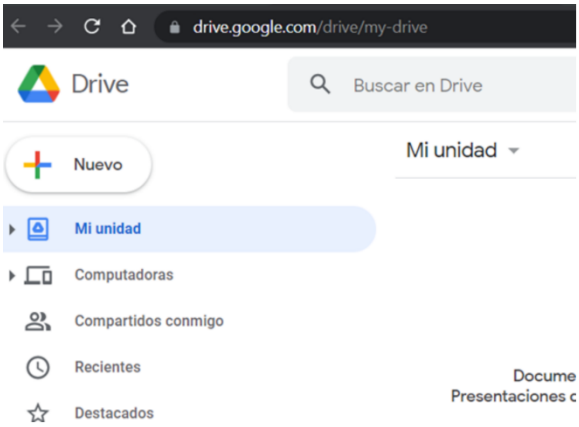


Figura 4-2. Accediendo a Google Drive.

P2) (1) De clic en "Nuevo" (en la parte superior izquierda). (2) Luego de clic en "Más". Y (3) luego presione clic en "+ Conectar más aplicaciones" (ver Figura 4-3).

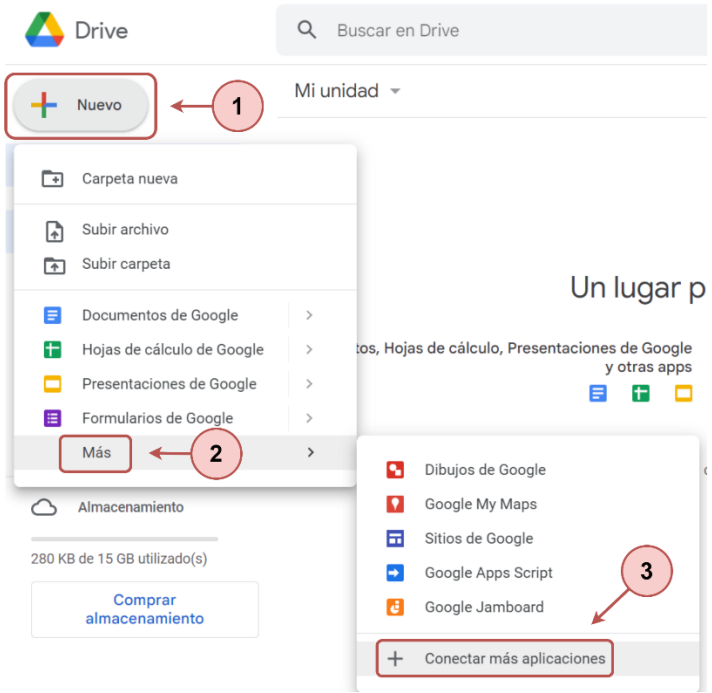


Figura 4-3. Conectando más aplicaciones a Google Drive.

P3) Una vez se abra la ventana emergente de la tienda de aplicaciones ("Google Workspace Martetplace"), (1) busque la palabra "Colab" en la barra superior de búsqueda y presione "Enter". Y (2) Luego de clic en "Colaboratory" (ver Figura 4-4).

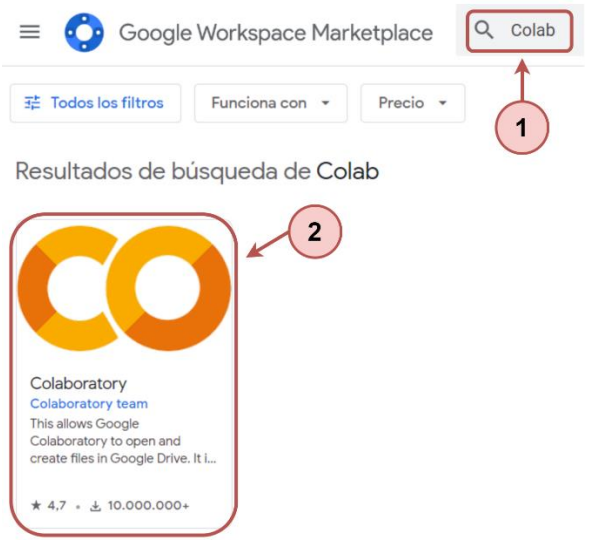


Figura 4-4. Buscando Google Colaboratory.

P4) A continuación, (1) de clic en “Instalar” (ver Figura 4-5).

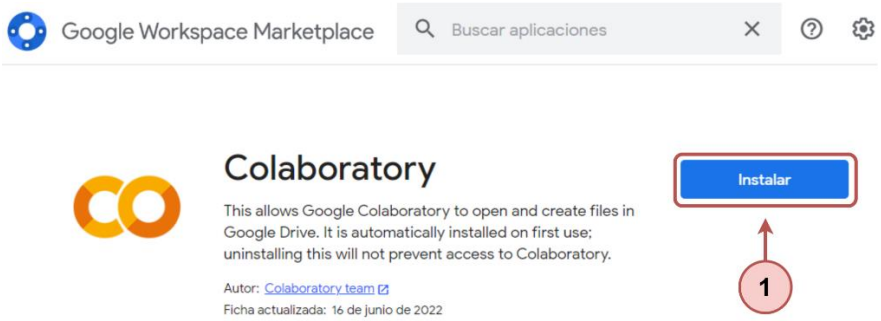


Figura 4-5. Instalando Google Colaboratory.

P5) (1) De clic en “Continuar” y luego (2) seleccione la cuenta en la que quiere instalar y activar Colab (ver Figura 4-6).

Figura 4-6. Brindando permisos y asociando cuenta a Colab.

P6) (1) De clic en “Aceptar” y luego (2) de clic en “Hecho” (ver Figura 4-7).

Figura 4-7. Aceptando y finalizando conexión con Colab.

P7) Finalmente cierre la tienda de aplicaciones de Google dando clic en la “X” en la esquina superior derecha de la ventana emergente.

Completando los pasos anteriores, ya puede crear documentos Colab en su cuenta de Google Drive. Ahora crearemos el primer documento Colab.

4.3. “Hola Mundo” en Colab

Para codificar en Python en Colab, primero debe crear un documento Colab siguiendo los pasos que se presentan a continuación.

P1) (1) De clic derecho en “Mi unidad” y (2) luego de clic en “Carpeta nueva”. A continuación, (3) cree una nueva carpeta llamada “Archivos Colab” y (4) de clic en “Crear” (ver Figura 4-8).

Figura 4-8. Creando una carpeta en Google Drive.

En esta carpeta se almacenarán todos los códigos que se desarrollen siguiendo este libro.

P2) De doble clic en la carpeta que acaba de crear (para ingresar a ella). Luego, (1) ubicado en la carpeta "Archivos Colab", (2) de clic en "Nuevo". Luego (3) de clic en "Más" y de (4) clic en "Google Colaboratory" (ver Figura 4-9).

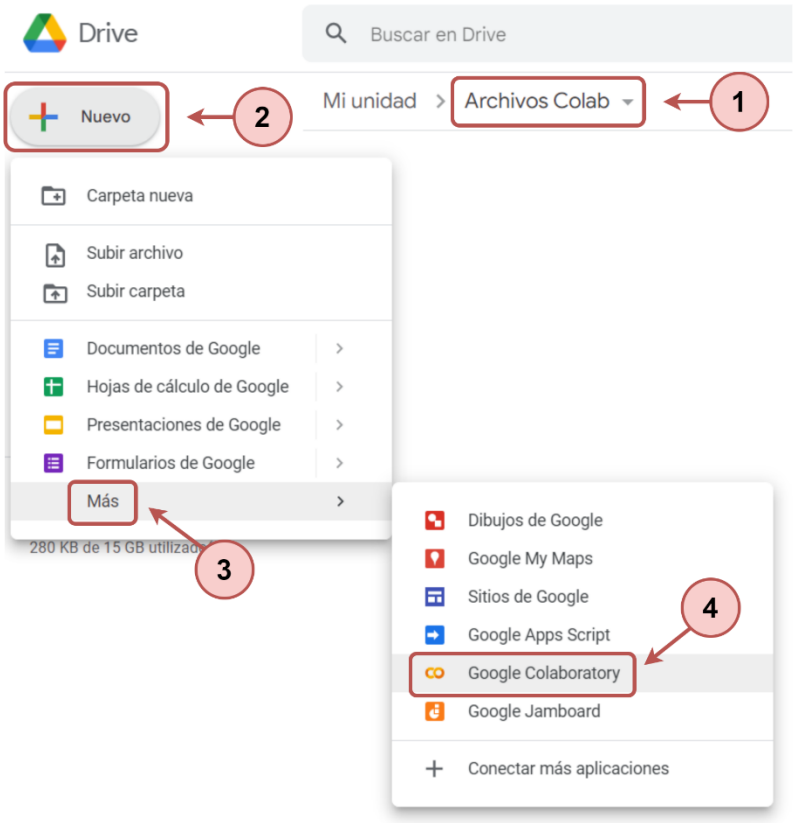


Figura 4-9. Creando un documento Colab.

Con este proceso acaba de crear su primer documento Colab.

P3) Al ejecutar el proceso anterior, se abrirá una nueva pestaña con el documento Colab. Ese documento Colab por defecto se nombra como "Untitled0.ipynb". Ahora de doble clic sobre "Untitled0.ipynb" y renómbrelo a "Cap-04-Hola-mundo-en-Colab.ipynb" (ver Figura 4-10).

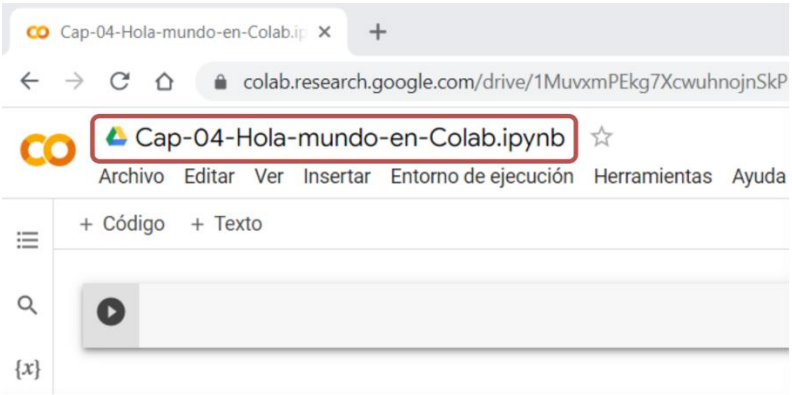


Figura 4-10. Renombrando documento Colab.

La recomendación es que, para cada capítulo de este libro, cree un nuevo documento Colab donde programe los ejercicios y códigos presentados.

P4) Ahora cree su primera nota tipo texto. (1) De clic en "+ Texto". (2) Luego escriba "Nuestro primer código – Hola mundo". Y finalmente, (3) de clic en la flechita hacia arriba "↑". Eso con el objetivo de colocar primero la explicación del código (ver Figura 4-11). Luego te mostraremos cómo colocar el código.

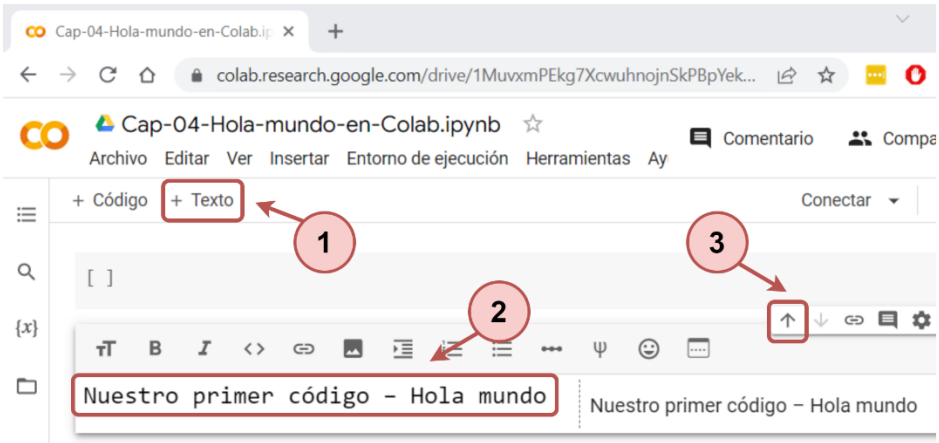


Figura 4-11. Creando una nota en Colab.

Colocar notas antes de código es muy útil para recordar fácilmente qué estamos programando. Esta es otra de las ventajas de Colab con respecto a otras herramientas similares.

P5) Ahora de clic en la zona con fondo gris que se encuentra debajo de nuestro primer texto (zona de código) y escriba *print("hola mundo")* en su interior (ver Figura 4-12).

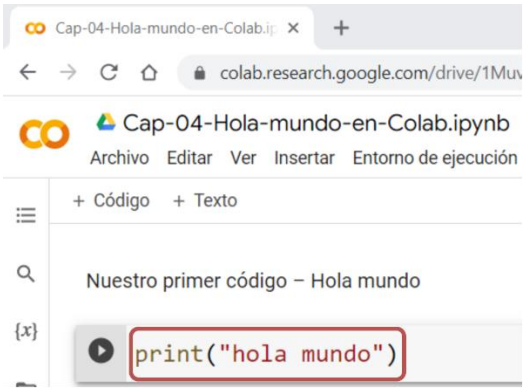


Figura 4-12. Programando un código en Colab.

P6) Por último, (1) de clic en la flecha blanca con fondo negro "▷". Para ejecutar el código anterior. (2) Debería ver el resultado de la ejecución debajo de la zona de código. Este proceso puede tomar unos cuantos segundos (ver Figura 4-13).

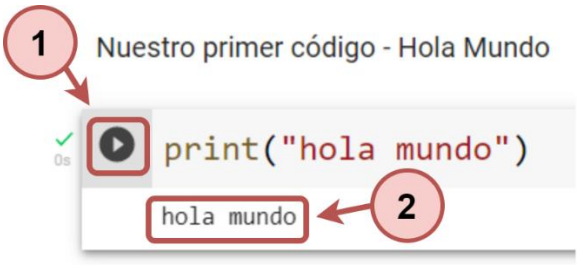


Figura 4-13. Ejecutando un código en Colab.

Felicidades, acaba de programar y ejecutar su primer código en Python.

4.4. Personalización de Colab

Antes de realizar un par de personalizaciones, practiquemos el uso de Colab sobre el documento actual.

Cree una nueva zona de texto (dando clic en “+ Texto”) y coloque el siguiente mensaje: *"Múltiples Hola Mundo"*. Luego cree una nueva zona de código (dando clic en “+ Código”) y añada el siguiente código:

Codificar y ejecutar

```
print("hola mundo")
print("hola mundo")
print("hola mundo")
```

La nueva sección deberá aparecer como se muestra en la Figura 4-14.

Figura 4-14. Añadiendo zonas a documento Colab.

Ahora ejecute el código anterior y el texto "hola mundo" deberá aparecer tres veces en la pantalla.

El problema con la visualización anterior es que a medida que añadimos más líneas de código, se hace más difícil identificar qué número de línea corresponde a qué instrucción. Por lo tanto, vamos a realizar una pequeña personalización en Colab.

(1) De clic en “Herramientas”, (2) luego clic en “Configuración”, (3) luego clic en “Editor”, y (4) seleccione las opciones: (i) “Mostrar números de línea” y (ii) “Mostrar guías de sangría”. Finalmente, (5) de clic en “Guardar” (ver Figura 4-15).

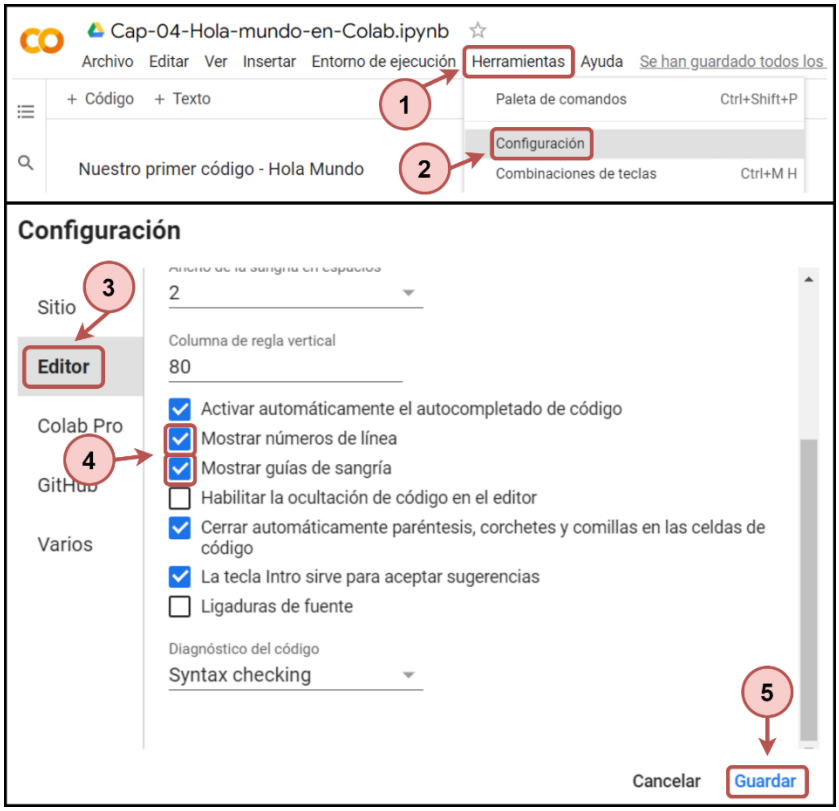


Figura 4-15. Personalización de Colab.

Al habilitar las opciones anteriores se facilitará la lectura del código desarrollado.

Versión de Python

Cada lenguaje de programación va evolucionando con el paso del tiempo. Aparecen nuevas versiones, con nuevas opciones y funcionalidades. Colab también va realizando actualizaciones en la versión de Python que utiliza. Al momento de escribir este libro, Colab está utilizando la versión 3.7.15 de Python. La Figura 4-16 muestra el código que se podría ejecutar si desea comprobar la versión de Python actual utilizada.

Figura 4-16. Comprobar versión de Python en Colab.

Errores críticos en Colab

Antes de finalizar este capítulo veamos la última sugerencia sobre Colab. **Préstele bastante atención a esta sugerencia, ya que será crucial en caso de que los códigos que desarrolla dejen de funcionar misteriosamente**.

Cada que se ejecuta una zona de código en Colab, la información de la ejecución queda guardada en la memoria (caché) de algo que se llama "Entorno de ejecución". Algunas veces, por errores en la lógica de programación, esa información almacenada genera problemas. Por ejemplo, en la Figura 4-17 puede observar que tratamos de volver a ejecutar el código de *print("hola mundo")* y obtenemos un error. Esto se debe a que en otra zona de código introdujimos a propósito un error que cambiaba el significado de *print* (en la memoria del entorno de ejecución).

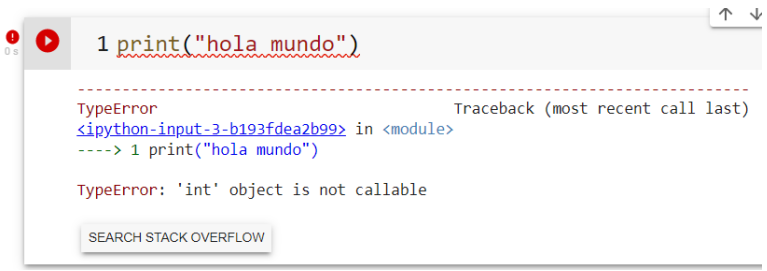


Figura 4-17. Error en Colab.

Entonces, si esto le sucede, si en algún momento observa un código que en teoría debería funcionar, pero no funciona. La solución es simple, (1) debe hacer clic en "Entorno de ejecución", y (2) luego dar clic en "Reiniciar entorno de ejecución". Finalmente, dar clic en "Sí" (ver Figura 4-18).

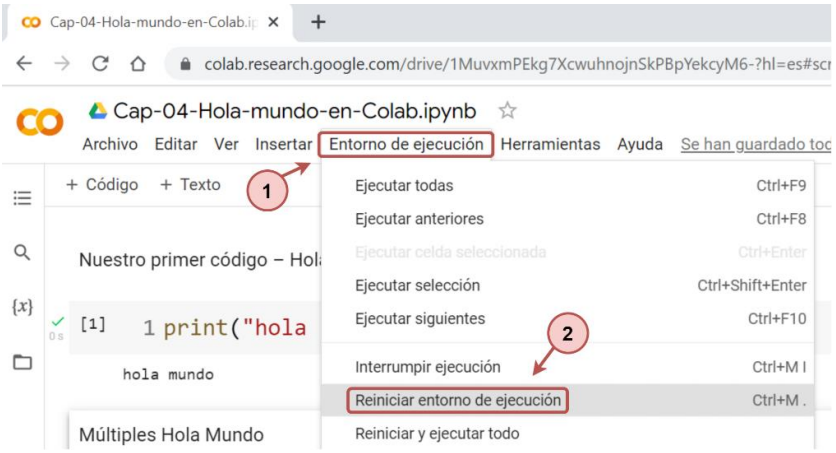


Figura 4-18. Reiniciar entorno de ejecución de Colab.

Resumen

En este capítulo aprendimos a utilizar Colab. Aprendimos a enlazar nuestra cuenta de Google Drive con Colab. Aprendimos a crear documentos Colab. Aprendimos a definir zonas de texto y código en Colab. Aprendimos a programar y ejecutar código en Colab. Y aprendimos a personalizar la apariencia de Colab.

En el siguiente capítulo aprenderemos a manipular variables en Python, uno de los temas fundamentales cuando iniciamos a programar.

Capítulo 05 – Variables enteras, flotantes y textos

Las variables son uno de los elementos clave que se utiliza en Python y en muchos de los lenguajes de programación. Las variables almacenan los valores (los datos) de nuestros programas. En este capítulo explicaremos cómo funcionan las variables y algunos de los principales tipos de variables.

A continuación, desarrollaremos las siguientes secciones:

1. Variables.
2. Definición de variables.
3. Tipos de variables.
4. Impresión de variables.
5. Modificación de variables.
6. Función type.
7. Comentarios en Python.
8. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo05. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

5.1. Variables

Una **variable** es un espacio reservado en la memoria del computador, donde almacenamos un valor. El siguiente código muestra cómo definir una variable en Python (en la siguiente sección explicaremos este proceso paso a paso). Allí vemos cómo definimos la variable *edad* con un valor de *17*.

Codificar

```
edad = 17
```

Al definir la variable *edad*, el computador reservará un espacio en su memoria para guardar el valor de esta variable (en este caso el número *17*).

En este libro nos imaginaremos las variables como “cajas” que contienen dos etiquetas y un papel. Por ejemplo, la Figura 5-1 representa gráficamente la variable *edad*.

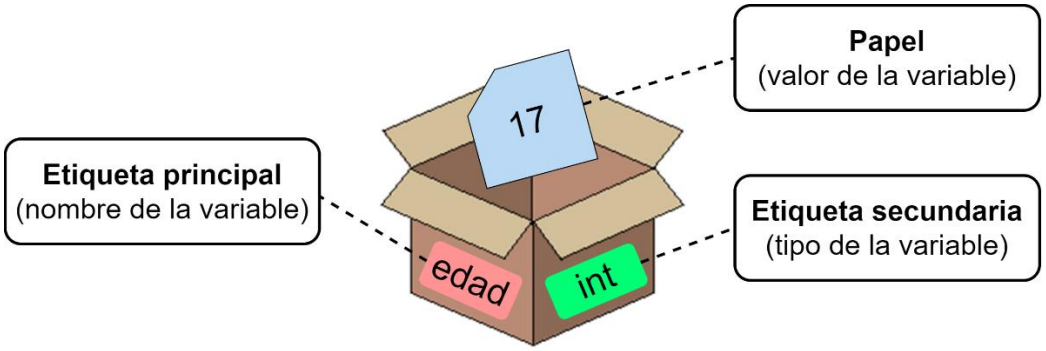


Figura 5-1. Representación gráfica de una variable.

La caja anterior contiene:

- **Nombre de la variable:** es la etiqueta principal de la caja (ubicada en el lado inferior izquierdo de la caja). El nombre lo definimos nosotros y nos ayudará a recordar qué es lo que estamos almacenando en la caja.
- **Valor de la variable:** es el papel que se almacena dentro de la caja (ubicado dentro de la caja). El valor lo definimos nosotros dependiendo de lo que deseemos almacenar.
- **Tipo de la variable:** es la etiqueta secundaria de la caja (ubicada en el lado inferior derecho de la caja). El tipo lo asigna Python dinámicamente dependiendo del valor que le asignamos a la variable. En este caso, Python le asigna un tipo *int* (entero) dado que estamos almacenando el número *17* (más adelante veremos diferentes tipos de variables).

Dependiendo de la complejidad del programa que estemos desarrollando, se utilizarán pocas variables, o se utilizarán cientos o miles de variables.

5.2. Definición de variables

En la sección anterior definimos nuestra primera variable. Ahora vamos a explicar ese proceso de definición en detalle. Para definir una variable en Python debemos utilizar la siguiente estructura:

Analizar estructura

```
nombreDeLaVariable = valorDeLaVariable
```

La estructura anterior se divide en tres partes:

- Primero debemos **darle un nombre a la variable**. Ese nombre se puede definir con letras y números y sin utilizar espacios. Debemos definir nombres que nos permitan recordar fácilmente qué es lo que estamos almacenando allí.
- Luego debemos colocar el operador "=". Este es un **operador de asignación**, el cual asigna el valor de la derecha en la variable de la izquierda.
- Finalmente, al lado derecho, se coloca **el valor que se quiere almacenar** en la variable.

Reglas para nombrar una variable

Veamos algunas reglas para nombrar variables:

- El nombre de una variable solo puede contener letras, números y guiones bajos (_). Utilizar cualquier otro carácter va a generar un error. Por ejemplo, no se puede usar una coma "," o un punto ".".
- El nombre de una variable debe iniciar con una letra o con un guion bajo. No puede iniciar con un número.
- No se puede usar una palabra reservada de Python. Por ejemplo, una variable no se puede llamar *def*, *if*, o *return* (algunas de estas palabras reservadas las explicaremos luego).

Como vimos anteriormente, darle un nombre apropiado a las variables es crucial para que nuestros códigos y programas sean entendibles.

Discusión rápida: Veamos la importancia del buen nombramiento de variables con estas dos frases tomadas del libro (*2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*). (i) *El principio de la sabiduría es llamar a las cosas por su nombre apropiado. - Confucio.* (ii) *¿Por qué es importante el nombramiento? Porque los buenos nombres hacen que el código sea más fácil de leer, y tenemos que leer el código para poderlo cambiar.*

Además de escoger nombres apropiados para las variables, también es muy recomendable utilizar un mismo estilo para la definición de esos nombres.

TIP: Cuando defina variables, utilice siempre el mismo estilo de nombramiento. En este libro utilizaremos *camelCase*. El ***camelCase*** es un estilo de escritura que consiste en la unión de dos o más palabras sin espacios entre ellas, pero las diferencian una letra mayúscula inicial a partir de la segunda palabra. De esta manera, si pensamos definir una variable que almacene la "edad de mi gato", la podríamos llamar *edadDeMiGato*.

5.3. Tipos de variables

Hasta el momento solo hemos definido una variable de tipo *int* (entero). Sin embargo, los programas que usamos diariamente no solo están compuestos de números enteros. Por ahora, analicemos tres tipos de variables comunes en Python.

Variables tipo *str* (texto)
Se definen como textos o caracteres entre comillas dobles o simples. Por ejemplo: “Hola”, “Casa”, “Feliz cumpleaños”, ‘Tengo 3 hermanas’.

Variables tipo *int* (entero):
Se definen como números que no contienen cifras decimales. Por ejemplo: -6, 1, 10, 25.

Variables tipo *float* (flotante):
Se definen como números que contiene cifras decimales. Por ejemplo: -1.25, 1.0, 1.25.

En el siguiente código observamos la definición de tres variables con distintos tipos. Además, en la Figura 5-2 vemos la representación gráfica de las tres variables definidas.

Codificar

```
nombre = "Maria"
edad = 17
estatura = 1.61
```

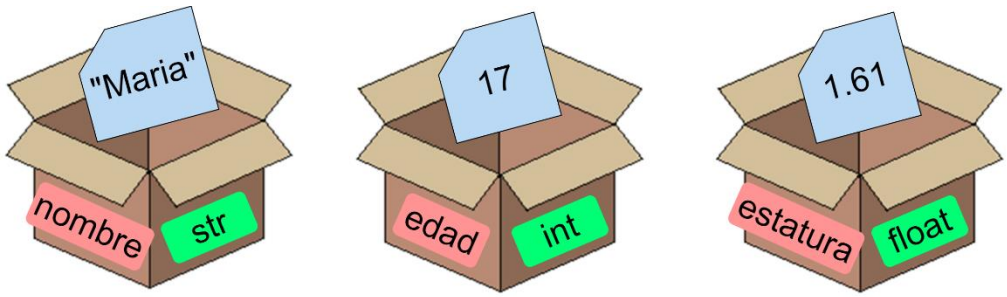


Figura 5-2. Representación gráfica de variables con distintos tipos.

5.4. Impresión de variables

Hasta este punto hemos definido variables, pero no las hemos utilizado en nuestros programas. Un proceso muy común es imprimir por pantalla el contenido de estas variables. Codifiquemos y ejecutemos el siguiente código.

Codificar y ejecutar

```
nombre = "Maria"
edad = 17
estatura = 1.61

print(nombre)
print(edad)
print(estatura)
```

En las tres últimas líneas del código anterior estamos utilizando la función *print*. La función ***print*** permite imprimir información por pantalla. En este caso, a la función *print* le pasamos entre los paréntesis el nombre de cada una de las variables. Y al ejecutar el código anterior (ver Figura 5-3), se mostrará por pantalla el valor almacenado en la variable *nombre* (o sea *Maria*), luego el valor almacenado en la variable *edad* (o sea *17*), y finalmente el valor almacenado en la variable *estatura* (o sea *1.61*).

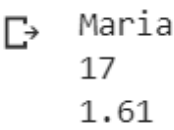


Figura 5-3. Ejecución del código anterior en Colab.

Discusión rápida: Los valores que se pasan entre paréntesis a las funciones se llaman argumentos. Un **argumento** es un valor que la función recibe para ejecutar acciones. En el caso anterior, le pasamos un argumento a la función *print*, el valor de ese argumento luego será mostrado por pantalla. El tema de funciones y argumentos lo discutiremos en detalle en el capítulo 14.

Por último, analicemos el siguiente código:

Codificar y ejecutar

```
edad = 17
print(edadd)
```

Al ejecutar el código anterior, veremos un error en la pantalla (ver Figura 5-4) que nos indica que la variable *edadd* no está definida. El problema es que el nombre de la variable contiene una letra "d" adicional. En Python debemos ser muy cuidadosos al usar el nombre de una variable, ya que el nombre debe ser exactamente igual en todas las líneas de código en que usemos la variable. Python incluso discrimina mayúsculas y minúsculas.

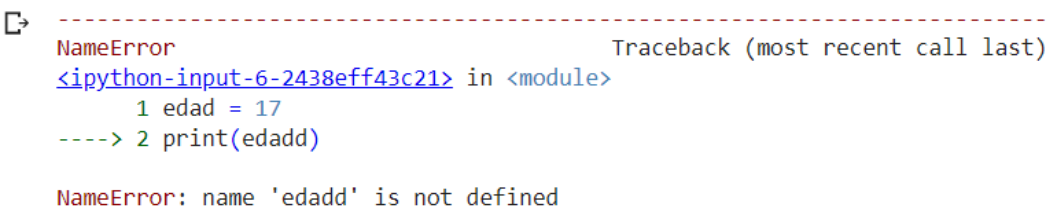


Figura 5-4. Error al usar una variable que no está definida.

5.5. Modificación de las variables

Las variables son, como su nombre lo indica, variables. Por ejemplo, en una variable se puede definir el precio de un producto, y posteriormente en el código se puede modificar ese precio (para aplicar, por ejemplo, un descuento).

Modificando el valor de una variable

El siguiente código muestra cómo definimos una variable *precioProducto*, imprimimos su contenido, luego modificamos su valor, y volvemos a imprimir su contenido. **Nota:** recuerde que **no debe codificar los números que aparecen a la izquierda**, esos números los utilizaremos para identificar fácilmente las líneas de código.

Codificar y ejecutar

```
1. precioProducto = 150
2. print(precioProducto)
3. precioProducto = 250
4. print(precioProducto)
```

La Figura 5-5 nos muestra lo que pasa internamente en la memoria del computador. **Las líneas 1 a 4 se ejecutan en orden**. Al ejecutar la línea 1 se define una variable *precioProducto* con valor de *150*, y al ejecutar la línea 3, a esa variable se le modifica el valor que antes estaba en *150* y ahora estará en *250*. Como las variables tienen el mismo nombre, Python solo reservará un espacio en memoria (**solo creará una caja**).

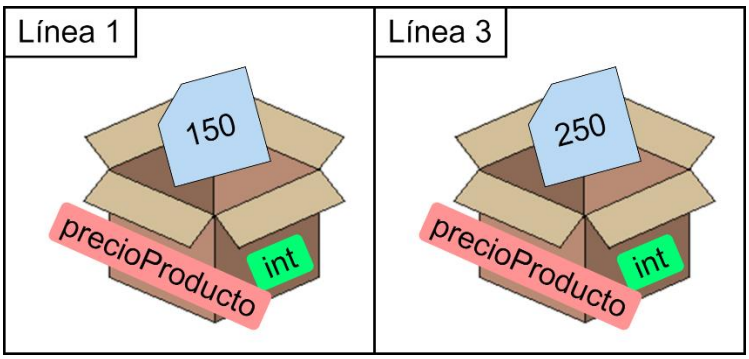


Figura 5-5. Modificando el valor de una variable.

Modificando el valor y el tipo de una variable

En Python es posible modificar el valor y el tipo de una variable. El siguiente código muestra cómo definimos una variable *precioProducto* (inicialmente con un valor *int*), imprimimos su contenido, luego modificamos su valor (a un valor *float*), y volvemos a imprimir su contenido.

Codificar y ejecutar

```
1. precioProducto = 150
2. print(precioProducto)
3. precioProducto = 250.5
4. print(precioProducto)
```

La Figura 5-6 nos muestra lo que pasa internamente en la memoria del computador. Al ejecutar la línea 1 se define la variable *precioProducto* con valor de *150*, y al ejecutar la línea 3 a esa

variable se le modifica el valor (ahora estará en *250.5*), lo cual, a su vez, genera que se le modifique su tipo (ahora será tipo *float*).

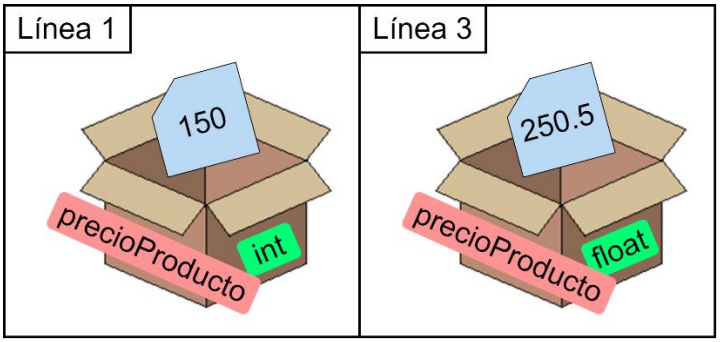


Figura 5-6. Modificando el valor y tipo de una variable.

Discusión rápida: Reconocer que las variables en Python manejan diferentes tipos nos ayudará a comprender mejor cómo funciona Python y las distintas posibilidades que tenemos para usar esas variables. Existen otros lenguajes de programación que manejan un tipado más estricto (por ejemplo, Java). En Java, a una variable *int*, no se le podría modificar el valor por un número *float* (solo se le podría modificar el valor por otro número *int*).

Reutilizando variables para definir nuevas variables

En muchos de los lenguajes de programación es posible definir o modificar variables a partir de los valores de variables previamente definidas. El siguiente código muestra cómo: (i) definimos una variable *edadLuis* con valor de *10*, (ii) definimos una variable *edadLaura* con valor de *20*, (iii) definimos una variable *sumaEdades* que será igual al valor contenido en la variable *edadLuis* más el valor contenido en la variable *edadLaura* (o sea *30*), (iv) definimos una variable *promedioEdades* que será igual a la variable *sumaEdades* divido entre *2*, e (v) imprimimos el valor almacenado en *promedioEdades* (15.0).

Codificar y ejecutar

```
1. edadLuis = 10
2. edadLaura = 20
3. sumaEdades = edadLuis + edadLaura
4. promedioEdades = sumaEdades/2
5. print(promedioEdades)
```

La Figura 5-7 muestra la impresión en pantalla al ejecutar el código anterior, como podemos observar el resultado es *15.0*, lo cual quiere decir que *promedioEdades* es de tipo *float*. Esto se debe a que la división en Python siempre genera un tipo *float* (sin importar el tipo de los números a dividir).

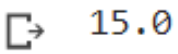


Figura 5-7. Ejecución del código anterior en Colab.

5.6. Función type

En las secciones anteriores identificamos los tipos de las variables imaginando cómo se almacenan estas variables en memoria. Sin embargo, Python ofrece una función llamada *type*, que nos permite identificar el tipo de cada variable. La función ***type*** recibe (entre los paréntesis) el nombre de una variable (un argumento) y nos retorna el tipo de esa variable (el tema de funciones, argumentos y retornos lo desarrollaremos en el capítulo 14). Luego, con la función *print* podemos imprimir en pantalla los tipos de esas variables.

El siguiente código muestra la definición de la variable *precioProducto* y posteriormente muestra por pantalla el tipo definido de esa variable (que inicialmente es tipo *int*). Luego la variable se modifica con un valor *float* y se vuelve a imprimir el tipo de la variable (en este caso *float*). La Figura 5-8 muestra la salida por pantalla al ejecutar este código.

Codificar y ejecutar

```
1. precioProducto = 150
2. print(type(precioProducto))
3. precioProducto = 250.5
4. print(type(precioProducto))
```

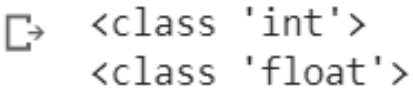


Figura 5-8. Ejecución del código anterior en Colab.

5.7. Comentarios en Python

Muchos de los lenguajes de programación permiten agregar comentarios a los códigos que vamos desarrollando. Estos comentarios se definen con el objetivo de dejar pistas acerca de qué hace el

código. Esas pistas pueden ser para mi "yo del futuro" (por ejemplo, si en dos semanas o dos meses retomo el proyecto y necesito recordar rápidamente que hacía esa pieza de código), o para que compañeros con los que esté codificando comprendan el código.

El siguiente código muestra dos maneras de definir comentarios en Python. La línea 1 muestra cómo definir **comentarios de una sola línea** (mediante el uso del símbolo hashtag "**#**"). Si una línea de código en Python inicia con "#", Python la ignorará (no la ejecutará). Las líneas 4 a 5 muestran cómo definir **comentarios de múltiples líneas** de código (mediante el uso de tres comillas dobles seguidas **"""**). Python ignorará todas las líneas que empiecen por tres comillas dobles, hasta encontrar otras tres comillas dobles que representen el cierre del comentario.

Codificar y ejecutar

```
1. # La siguiente línea representa el salario mensual de una persona
2. salario = 1000
3.
4. """ Dividiremos el salario entre 2, para repartirlo entre los integrantes
5. de la familia y mostraremos el resultado por pantalla """
6. print(salario/2)
```

5.8. Ejercicios

Ejercicios teóricos

E5.1. ¿De qué tipo es cada una de las siguientes variables?

Analizar

```
1. totalDeuda = 1450.7
2. nombreProducto = "Comedor"
3. lunas = 7
4. motores = "4"
```

E5.2. ¿Para qué sirve la función *print*?

E5.3. ¿En cuál de las siguientes líneas de código se presenta una manera incorrecta de definir una variable?

Analizar

```
1. edad gato = 7
2. Correo_electronico = "prueba@gmail.com"
3. copiaCorreo = Correo_electronico
```

Ejercicios prácticos

E5.4. Implemente un programa en el cual defina 4 variables: una que contenga su nombre, otra para su apellido, otra para su edad, y otra para su estatura. Y al final, imprima por pantalla el contenido de cada variable. **Nota:** utilice el tipo que considere más adecuado para cada variable.

Resumen

En este capítulo aprendimos a definir y manipular variables en Python. Aprendimos tres tipos de variables fundamentales para los programas que desarrollamos (*int*, *float* y *str*). Nos imaginamos las variables como cajas que contienen un nombre, un tipo, y un valor. Aprendimos que las variables se pueden modificar, cambiándoles, ya sea su valor, su tipo o ambos. Descubrimos que *print* nos sirve para imprimir datos por pantalla, descubrimos que *type* nos sirve para retornar el valor de una variable, y descubrimos dos maneras de definir comentarios en Python.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una “X” los siguientes trucos: T01, T02, T03, T07, T08, T09, T10 y T11.

La Figura 5-9 muestra un extracto de la hoja de trucos, en la cual se marcan los trucos aprendidos hasta el momento.

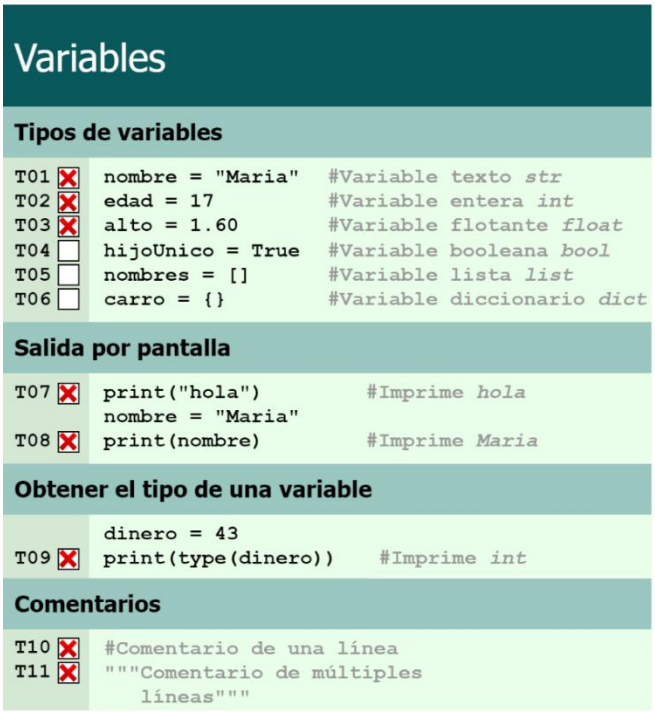


Figura 5-9. Fragmento de la Hoja de Trucos, donde el lector marca los trucos vistos en este capítulo.

En el siguiente capítulo retomaremos la impresión de información por pantalla (salida) y aprenderemos cómo ingresar nuestros propios datos al programa (entrada).

Capítulo 06 – Entrada y salida de datos

Muchos de los programas que utilizamos a diario son interactivos. Muchos de ellos requieren que como usuarios les ingresemos información, a través de formularios, instrucciones de voz, o videos, entre otros. Además, basados en la información ingresada, los programas nos generan respuestas. Estas respuestas son mensajes impresos en nuestra pantalla, reproducciones de audios o videos, mensajes de texto, entre otros.

En este capítulo aprenderemos cómo ingresar datos a nuestros programas a través del teclado (**entrada**). Aprenderemos, también, cómo esos programas nos muestran resultados (**salidas**) en la pantalla de nuestra computadora.

A continuación, desarrollaremos las siguientes secciones:

1. Entrada de datos por teclado.
2. Conversión de tipos de variables.
3. Función print.
4. Concatenación de datos en salida por pantalla.
5. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo06. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

6.1. Entrada de datos por teclado

En este libro trabajaremos principalmente con la recolección de información mediante el teclado del dispositivo utilizado para programar. Es muy común que un programa requiera el ingreso de información por teclado. Por ejemplo, un programa podría solicitar a un usuario que ingrese su nombre, su edad, su salario, su correo, o la calificación de algún curso, entre otros.

Función *input*

En Python, la función ***input*** permite capturar información por medio del teclado del usuario. Al codificar esta función, el programa esperará a que el usuario digite información mediante el teclado y que luego presione *Enter*. La información recogida normalmente se asigna a una

variable. Sin importar lo que el usuario ingrese por teclado, esa información se guardará como un tipo *str*.

La línea 2 del siguiente código muestra cómo utilizar la función *input*. Cuando el código se ejecute, el programa mostrará por pantalla el mensaje de la línea 1, y luego en la línea 2 esperará a que el usuario ingrese información mediante el teclado. Una vez el usuario ingresa la información y presiona *Enter*, la función *input* capturará el valor ingresado, y ese valor se asignará a la variable *nombre* (recuerde que por defecto el tipo de la variable será *str*). Luego la línea 3 imprimirá un mensaje, y en la línea 4 se imprimirá el valor almacenado en la variable *nombre*. La Figura 6-1 muestra la ejecución de la línea 2.

Codificar y ejecutar

```
1. print("Ingrese su nombre")
2. nombre = input()
3. print("Su nombre es:")
4. print(nombre)
```

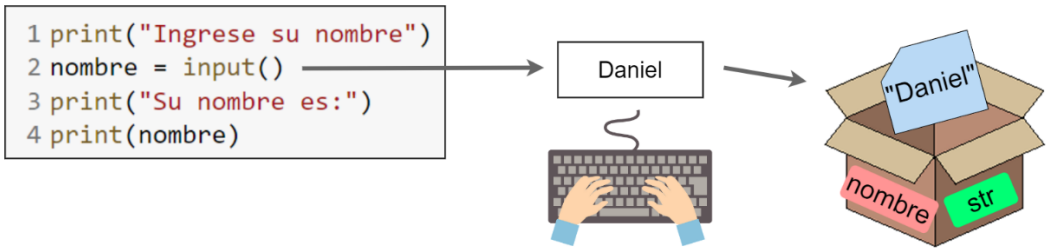


Figura 6-1. Ejecución de código con función *input*.

Función *input* – tipo por defecto

Por defecto, todo lo que se ingrese por teclado tras el llamado de la función *input* es tipo *str* (texto). No importa si lo que ingresa es un número o un decimal, igual se definirá como tipo *str* (texto).

El siguiente código muestra un programa en el que en la línea 1 se recibe por teclado la edad de una persona. Esa edad se almacena en la variable *edad*. Nótese que en la línea 1 estamos enviando un argumento (un texto) a la función *input*. Ese argumento es opcional, pero cuando se envía, la función *input* imprimirá por pantalla el valor del argumento recibido (en este caso imprimirá *"Ingrese su edad: "*). Luego, la línea 2 calculará la mitad de la edad recibida por teclado, dividiendo la variable *edad* entre *2* (y asignando el resultado a la variable *mitadDeEdad*). Finalmente, la línea 3 imprimirá por pantalla el valor almacenado en *mitadDeEdad*. Sin embargo,

tal como se observa en la Figura 6-2 el código mostrará por pantalla un error, indicando que no se puede dividir un valor *str* (haciendo referencia a la variable *edad*) entre un valor *int* (haciendo referencia al número *2*). En Python no es posible dividir un texto entre un entero. Además, ya confirmamos que la función *input* siempre retornará una variable *str* (sin importar si le ingresamos un número).

Codificar y ejecutar

```
1. edad = input("Ingrese su edad: ")
2. mitadDeEdad = edad/2
3. print(mitadDeEdad)
```

```
Ingrese su edad: 21
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-f910c790f3a2> in <module>
      1 edad = input("Ingrese su edad: ")
----> 2 mitadDeEdad = edad/2
      3 print(edad)

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Figura 6-2. Ejecución de código con función *input*.

Para solucionar el problema anterior debemos realizar una conversión de tipos que se explicará en la siguiente sección.

TIP: Colab siempre muestra los errores en inglés. Si tiene poco conocimiento en este idioma, le sugerimos copiar y pegar el texto del error en el traductor de Google o aquel de su preferencia para entender qué está pasando. Muchas veces invertimos mucho tiempo tratando de hacer funcionar nuestro código sin analizar lo obvio "lee el maldito mensaje de error" (tomado de: *2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*).

6.2. Conversión de tipos de variables

La conversión de tipos de variables es un proceso bastante común en programación. Algunas veces recibiremos información por pantalla que necesitaremos convertir a *int* o *float*. Algunas veces necesitaremos convertir números a textos para guardar información en archivos *txt* (texto plano) o *csv* (datos separados por comas), entre otros.

Funciones *int*, *float* y *str*

La función ***int*** permite convertir un número o un texto a tipo *int*.
La función ***float*** permite convertir un número o un texto a tipo *float*.
La función ***str*** permite convertir un valor a tipo *str*.

A todas las funciones anteriores se les debe enviar entre paréntesis (como un argumento), el valor a convertir. Ese valor pasado como argumento debe ser un valor válido para convertir. Por ejemplo, se puede ejecutar el código *int("3")*, puesto que el texto enviado como argumento contiene un valor válido a convertir a *int*. Sin embargo, no se puede ejecutar *int("Hola")*, ya que *"Hola"* no se puede representar numéricamente, por lo tanto, no se podrá convertir a *int*.

El siguiente código muestra cómo corregir el código de "calcular la mitad de la edad" para que no arroje un error. En este caso, utilizamos la función *int* para convertir el valor recogido por la función *input* a tipo *int*. En este caso, la línea 2 se podrá ejecutar, ya que realizará una división entre valores *int*, lo cual es válido. Y finalmente se calculará e imprimirá la mitad de la edad (ver ejecución en Figura 6-3).

Codificar y ejecutar

```
1. edad = int(input("Ingrese su edad: "))
2. mitadDeEdad = edad/2
3. print(mitadDeEdad)
```

```
Ingrese su edad: 21
10.5
```

Figura 6-3. Ejecución de código con función *input* y conversión de tipo.

El siguiente código muestra las diferencias al convertir los valores recibidos por la función *input* en distintos tipos de datos. La línea 1 muestra cómo recibir por teclado la edad del padre (como no hay conversión, la variable *edadPadre* será de tipo *str*). La línea 2 muestra cómo recibir por teclado la edad de la madre (como sí hay conversión, la variable *edadMadre* será de tipo *int*). La línea 3 muestra cómo recibir por teclado la estatura del padre (como sí hay conversión, la variable *estaturaPadre* será de tipo *float*). Finalmente, se imprime el tipo y valor de cada variable por pantalla. La Figura 6-4 muestra la ejecución del siguiente código.

Codificar y ejecutar

```
1. edadPadre = input("Ingrese edad: ")
2. edadMadre = int(input("Ingrese edad: "))
3. estaturaPadre = float(input("Ingrese estatura: "))
4. print(type(edadPadre))
5. print(type(edadMadre))
6. print(type(estaturaPadre))
7. print(edadPadre)
8. print(edadMadre)
9. print(estaturaPadre)
```

```
Ingrese edad: 30
Ingrese edad: 29
Ingrese estatura: 1.72
<class 'str'>
<class 'int'>
<class 'float'>
30
29
1.72
```

Figura 6-4. Ejecución del código anterior.

La Figura 6-5 muestra la representación gráfica de las variables previamente definidas.

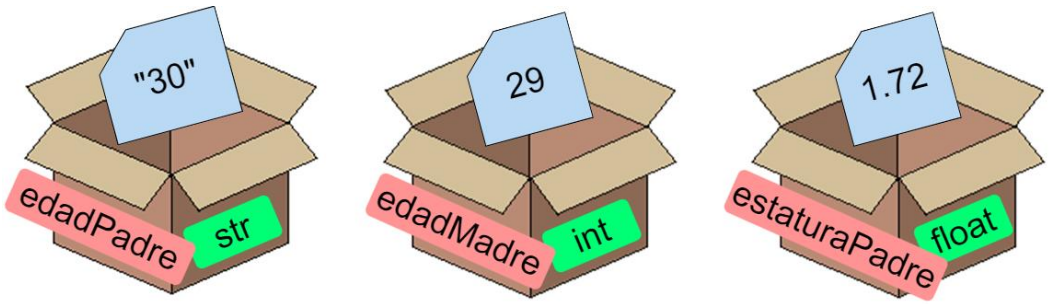


Figura 6-5. Representación gráfica de las variables del código anterior.

6.3. Función print

Hasta este punto hemos utilizado múltiples veces la función *print*. La función *print* será el mecanismo más utilizado en este libro para mostrar información o resultados de nuestros programas. A la función *print* se le envía un argumento, ya sea un texto o una variable. La función *print* toma ese argumento y lo muestra por pantalla.

El siguiente código muestra cómo definimos un par de variables y las mostramos por pantalla. Las líneas 3 y 5 muestran cómo utilizar la función *print* para mostrar por pantalla valores tipo texto, mientras que las líneas 4 y 6 muestran cómo utilizar la función *print* para mostrar por pantalla valores contenidos dentro de variables. Ambos casos son válidos en Python.

Codificar y ejecutar

```
1. nombre = "Paola"
2. edad = 30
3. print("Nombre: ")
4. print(nombre)
5. print("Edad: ")
6. print(edad)
```

El código anterior presenta un inconveniente, cuando tenemos muchas variables y muchos textos, tendríamos que definir decenas de líneas de código para imprimir en pantalla, uno a uno, cada valor y variable. En la siguiente sección veremos cómo solucionar este problema.

6.4. Concatenación de datos en salida por pantalla

Unir múltiples salidas y valores (que están relacionados) en un solo *print* resulta muy conveniente para reducir la cantidad de líneas de código.

En Python, utilizando un mismo *print* podemos imprimir múltiples valores y múltiples contenidos de variables. Para realizar este procedimiento, debemos utilizar el operador "+", el cual permite **concatenar** (juntar) múltiples textos. Algunas veces requeriremos la función *str* (para convertir un valor numérico a tipo *str*) para poder realizar el proceso anterior. Ya que en Python no podemos concatenar un texto con un número.

El siguiente código muestra cómo realizar el proceso de concatenación. La línea 3 utiliza el operador "+" para concatenar el texto *"Nombre: "* con la variable *nombreProducto* de tipo *str*. Esta concatenación es válida ya que ambos valores son *str*. Sin embargo, el código fallará al

ejecutar la línea 4, porque en esta línea utilizamos el operador "+" para concatenar el texto *"Precio: "* con la variable *precioProducto* de tipo *int*. Dado que Python no permite concatenar textos con números, el programa al ejecutarse mostrará un error por pantalla (ver Figura 6-6).

Codificar y ejecutar

```
1. nombreProducto = "Televisor"
2. precioProducto = 200
3. print("Nombre: "+nombreProducto)
4. print("Precio: "+precioProducto)
```

```
----> 4 print("Precio: "+precioProducto)

TypeError: can only concatenate str (not "int") to str
```

Figura 6-6. Error al tratar de concatenar tipos inválidos.

Para solucionar el problema anterior, debemos convertir el valor numérico a tipo *str*. El siguiente código muestra cómo en la línea 4 utilizamos la función *str* para convertir el valor almacenado en la variable *precioProducto* a tipo *str*, y ahora la concatenación es válida. La Figura 6-7 muestra la salida por pantalla de la ejecución del código anterior.

Codificar y ejecutar

```
1. nombreProducto = "Televisor"
2. precioProducto = 200
3. print("Nombre: "+nombreProducto)
4. print("Precio: "+str(precioProducto))
```

```
Nombre: Televisor
Precio: 200
```

Figura 6-7. Ejecución del código anterior.

Discusión rápida: En Python también es posible imprimir múltiples valores y variables dentro de un solo *print* al separar estos valores por comas "," (tal como lo muestra el siguiente código). En este caso, los valores separados por comas pueden ser de distintos tipos (no hay que convertirlos todos a *str*). Allí vemos que no necesitamos convertir *precioProducto* a *str*. Sin embargo, en este libro solo utilizaremos la concatenación, ya que al repetir este proceso vamos interiorizando el concepto de tipado de variables en Python. Lo cual será muy útil para entender conceptos posteriores.

Codificar y ejecutar

```
1. nombreProducto = "Televisor"
2. precioProducto = 200
3. print("Nombre:",nombreProducto,"- Precio:",precioProducto)
```

6.5. Ejercicios

Ejercicios teóricos

E6.1. ¿De qué tipo es cada una de las siguientes variables? **Nota:** suponga que en la línea 2 el usuario ingresa *35*.

Analizar

```
1. edadPadre = 44
2. edadMadre = input("Ingrese la edad: ")
3. edadModificada = int(edadMadre)
```

E6.2. ¿Para qué sirve la función *input*?

E6.3. ¿Cuál es el nombre de la función que se utiliza para convertir un valor a tipo *str*?

Ejercicios prácticos

E6.4. Implemente un programa en el cual le solicite al usuario que ingrese por teclado: su nombre, su apellido, su edad y su estatura. Almacene cada valor en una variable (tendrá cuatro variables). Y al final, imprima por pantalla el contenido y el tipo de cada variable. **Nota:** al solicitar los datos por pantalla, convierta los valores a los tipos más adecuados.

E6.5. Implemente un programa en el cual le solicite al usuario que ingrese por teclado: primero su nombre y luego su apellido. El programa deberá imprimir por pantalla el apellido, un espacio en blanco, y luego el nombre (todo junto en una misma línea). **Pista:** para dejar un espacio en blanco en el medio, concatene el apellido con lo siguiente: " ", y luego concatene el nombre.

Ejemplo de entada	**Ejemplo de salida**
Sebastian *Gomez*	*Gomez Sebastian*

E6.6. Un científico loco siempre calcula mal los años en los que probablemente un asteroide caerá a la tierra. Si el científico dice que un asteroide caerá en 10 años, es porque realmente caerá en 5 años (siempre se equivoca estimando el doble de la cantidad de años real). Implemente un programa en el cual le solicite al usuario que ingrese por teclado el número de años en los que caerá un asteroide (según el científico loco, en formato *float*). E imprima por pantalla el texto "El asteroide caerá en: ", concaténelo con el resultado real de los años en que caerá, y finalmente concaténelo con el texto " años".

Ejemplo de entada	Ejemplo de salida
28.4	*El asteroide caerá en 14.2 años*

Resumen

En este capítulo aprendimos cómo ejecutar operaciones de entrada y salida de datos. Para la entrada de datos aprendimos a utilizar la función *input* (la cual recoge información del teclado del usuario). Además, aprendimos a utilizar las funciones *int*, *str* y *float* para realizar conversiones de tipos, las cuales resultan convenientes cuando se necesita realizar operaciones matemáticas o concatenar datos en salidas por pantalla. Para la salida por pantalla aprendimos en detalle cómo utilizar la función *print*, cómo se le pueden pasar valores o variables, y cómo se pueden concatenar datos para reducir el número de *prints* mediante el uso del operador "+".

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una "X" los siguientes trucos: T12, T13, T14, T15 y T16.

En el siguiente capítulo aprenderemos cómo definir condiciones en nuestros programas.

Capítulo 07 – Condicionales simples

Otro elemento clave en los programas que utilizamos a diario es la evaluación de condiciones. Por ejemplo, cuando utilizamos un cajero y retiramos dinero, el programa del cajero verifica que el dinero a retirar sea menor o igual que el dinero que tenemos en nuestra cuenta (evalúa una condición). Cuando nos conectamos a nuestro correo electrónico, el programa verifica que el correo y contraseña que ingresemos coincidan con los datos de autenticación almacenados en el sistema. O cuando estamos jugando un videojuego, y nuestro personaje muere, el sistema verifica si contamos con más vidas, de lo contrario, termina el juego.

En este capítulo aprenderemos cómo definir y evaluar condiciones y todo lo subyacente a esta temática. Además, explicaremos cómo programar un condicional simple.

A continuación, desarrollaremos las siguientes secciones:

1. Introducción.
2. Variables booleanas.
3. Operadores de comparación.
4. Operadores lógicos.
5. Condicional if-then.
6. Flujo de ejecución del programa.
7. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo07. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

7.1. Introducción

Para comprender mejor la importancia de los condicionales analicemos el siguiente ejemplo. En la línea 1 del siguiente código se muestra cómo le solicitamos al usuario que ingrese la longitud del lado de un cuadrado, valor que convertimos a flotante y almacenamos en la variable *lado*. Luego, en la línea 2, calculamos el área del cuadrado con el valor del lado ingresado previamente. Y luego, en la línea 3 imprimimos el resultado por pantalla.

Codificar y ejecutar

```
1. lado = float(input("Ingrese el lado del cuadrado: "))
2. area = lado*lado
3. print("El área del cuadrado es: "+str(area))
```

Hasta el momento, el código parece bien definido. Si ejecutamos el código anterior, e ingresamos *8*, el programa nos mostrará por pantalla: *El área del cuadrado es: 64.0*. Sin embargo, las cosas empiezan a fallar cuando, por ejemplo, volvemos a ejecutar el código anterior, e ingresamos *-8*. En este caso, el programa se ejecutará normalmente y nos mostrará el mismo resultado: *El área del cuadrado es: 64.0*. Pero esto no es correcto, el programa nos arrojó un resultado erróneo. Esto debido a que no existe un cuadrado que tenga un lado con longitud negativa.

Para solucionar este problema, antes de calcular el área, **debemos verificar que la longitud del lado que nos ingrese el usuario por teclado sea positiva (mayor a cero)**. En caso de que el lado sea mayor a cero, podremos proceder y calcular el área e imprimir el resultado por pantalla, y en caso contrario, podríamos mostrar un mensaje de error.

Para brindarle a nuestro código la habilidad anterior, necesitamos aprender a utilizar condicionales. Pero antes de utilizarlos, veamos unos conceptos importantes.

7.2. Variables booleanas

Hasta este punto hemos trabajado con variables y valores *int*, *float* y *str*. Estos tres tipos de variables permiten almacenar una cantidad bastante amplia de valores. Para trabajar con condicionales, debemos aprender a utilizar un nuevo tipo: las variables booleanas.

Una **variable booleana (*bool*)** solo permite almacenar dos valores: verdadero (*True*) o Falso (*False*). Normalmente, *True* se asocia a “Sí” y *False* se asocia a “No”. Estas variables o valores son generalmente utilizados para evaluar condiciones o comparaciones (tal como veremos en este capítulo).

La línea 1 del siguiente código muestra la definición de una variable booleana llamada *tengoHijos*, que la utilizamos para indicar si tenemos hijos o no. Luego en la línea 2 imprimimos su tipo, y en la línea 3 su valor. La Figura 7-1 muestra la representación gráfica de la variable *tengoHijos*.

Codificar y ejecutar

```
1. tengoHijos = True
2. print(type(tengoHijos))
3. print(tengoHijos)
```

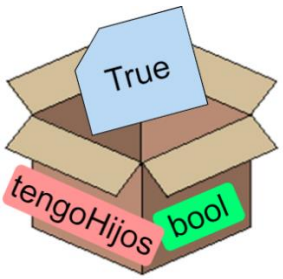


Figura 7-1. Representación gráfica de la variable *tengoHijos*.

Nota: los valores de las variables booleanas no se definen con comillas (*True* o *False* deben estar sin comillas). Si lo hacemos, entonces ya no estaríamos definiendo una variable *bool*, sino una *str*.

7.3. Operadores de comparación

Antes de definir condiciones en Python, debemos aprender a comparar valores. ¿Cómo comparamos dos valores?, ¿Cómo sé si un lado es mayor a cero, igual a cero, o menor a cero?

En programación, los **operadores de comparación** permiten comparar dos valores y evaluar el resultado de esa comparación en un solo valor booleano (*True* o *False*). Python proporciona seis operadores de comparación (que se describen en la Tabla 7-1).

Tabla 7-1. Operadores de comparación.

Operador	Significado
==	Igual a
!=	Diferente de
<	Menor que
>	Mayor que
<=	Menor o igual a
>=	Mayor o igual a

Practiquemos el uso de operadores de comparación con el siguiente ejemplo. Supongamos que se define una variable llamada *lado* con valor de *5*. Veamos en la Tabla 7-2 el resultado de realizar una serie de comparaciones sobre la variable *lado*.

Tabla 7-2. Comparaciones sobre *lado = 5*.

Comparación	Resultado
lado < 0	False
lado >= 2	True
lado != 5	False
lado == 5	True
lado > -3	True
lado <= 120	True

Discusión rápida: En Python es importante aprender a diferenciar el igual "=", del doble igual "==". El "=" (operador de asignación) se utiliza para asignar un valor a una variable. El "==" (operador de comparación) se utiliza para comparar dos valores.

7.4. Operadores lógicos

Los valores booleanos (*True* y *False*), también se pueden comparar entre ellos. Similar a como funcionan los operadores de comparación, los **operadores lógicos** permiten comparar valores booleanos y evaluar el resultado de las comparaciones en un solo valor booleano (*True* o *False*).

Los operadores lógicos son útiles cuando queremos realizar múltiples comparaciones juntas. Por ejemplo, verificar que una persona es mayor de edad, y que, a su vez, tiene dinero para comprar una cerveza.

Python proporciona tres operadores lógicos (ver Tabla 7-3).

Tabla 7-3. Operadores lógicos.

Operador	Significado
and	y
or	o
not	negación

Operador *and*

El operador *and* (y) compara dos valores booleanos o expresiones. Este comparador **evalúa la comparación a *True* si ambos valores booleanos (o expresiones) son verdaderos. En cualquier otro caso evalúa a *False***.

En la tabla 7-4 se ven las posibles combinaciones y resultados al utilizar el operador *and*.

Tabla 7-4. Tabla de verdad del operador *and*

Expresión	Resultado
True *and* True	True
True *and* False	False
False *and* True	False
False *and* False	False

Practiquemos el uso del operador *and* con el siguiente ejercicio. Supongamos que se define una variable llamada *edad* con valor de *18* y una variable *nombre* con valor *"Paola"*. Veamos en la Tabla 7-5 el resultado de evaluar una serie de expresiones sobre ambas variables.

Tabla 7-5. Uso del operador *and* sobre *edad = 18* y *nombre = "Paola"*.

Expresión	Resultado
(edad == 18) *and* (nombre == "Paola")	True
(edad > 20) *and* (nombre == "Paola")	False
(edad <= 60) *and* (nombre != "Rosa")	True
(edad != 18) *and* (nombre == "Paola")	False

Analicemos paso a paso la segunda expresión: *(edad > 20) and (nombre == "Paola")*. (1) Evaluamos la comparación de la izquierda. La *edad* no es mayor a *20*, por lo tanto, esa comparación evalúa a *False*. (2) Evaluamos la comparación de la derecha. El *nombre* sí es igual a *"Paola"*, en consecuencia, esa comparación evalúa a *True*. Finalmente (3) debemos evaluar: *(False) and (True)*. Y como ya vimos en la tabla de verdad del operador *and*, esa expresión evalúa a *False*. La Figura 7-2 muestra la evaluación de esta expresión paso a paso.

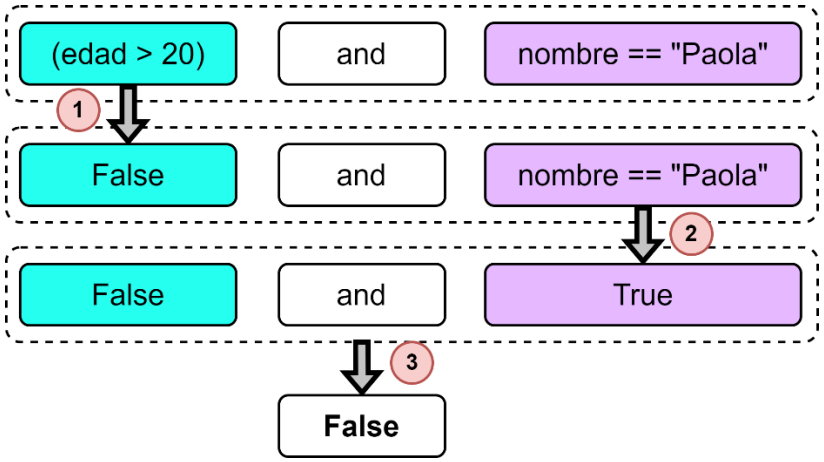


Figura 7-2. Evaluación de la expresión *(edad > 20) and (nombre == "Paola")*.

Operador *or*

El operador *or* (o) compara dos valores (o expresiones) booleanas. Este comparador **evalúa la comparación a *True* si algún valor booleano (o expresión) es verdadera. Solo si ambos son falsos, evalúa a *False*.**

En la tabla 7-6 se ven las posibles combinaciones y resultados al utilizar el operador *or*.

Tabla 7-6. Tabla de verdad del operador *or*

Expresión	Resultado
True *or* True	True
True *or* False	True
False *or* True	True
False *or* False	False

Practiquemos el uso del operador *or* con el siguiente ejemplo. Supongamos que se define una variable llamada *edad* con valor de *18* y una variable *nombre* con valor *"Paola"*. Veamos en la Tabla 7-7 el resultado de evaluar una serie de expresiones sobre ambas variables.

Tabla 7-7. Uso del operador *or* sobre *edad = 18* y *nombre = "Paola"*.

Expresión	Resultado
(edad == 18) *or* (nombre == "Paola")	True
(edad > 20) *or* (nombre == "Rosa")	False
(edad <= 60) *or* (nombre != "Rosa")	True
(edad != 18) *or* (nombre == "Paola")	True

Operador *not*

El operador *not* (negación) opera sobre un solo valor booleano o expresión. Este operador **simplemente evalúa al valor contrario que tiene el valor booleano de la expresión**.

En la tabla 7-8 se ven las posibles combinaciones y resultados al utilizar el operador *not*.

Tabla 7-8. Tabla de verdad del operador *not*

Expresión	Resultado
not True	False
not False	True

Practiquemos el uso del operador *not* con el siguiente ejemplo. Supongamos que se define una variable llamada *edad* con valor de *18* y una variable *tieneHijos* con valor *True*. Veamos en la Tabla 7-9 el resultado de evaluar una serie de expresiones sobre ambas variables.

Tabla 7-9. Uso del operador *not* sobre *edad = 18* y *tieneHijos = True*.

Expresión	Resultado
not(tieneHijos)	False
not(edad > 20)	True
not(edad <= 60)	False
not(edad == 18)	False

Ahora que sabemos utilizar operadores de comparación y operadores lógicos, podemos empezar a utilizar condicionales en Python.

7.5. Condicional if-then

Empecemos a trabajar con condicionales en Python. Los **condicionales** o sentencias condicionales son grupos de instrucciones que se pueden ejecutar o no en función del valor de una condición.

Existen varias estructuras de condicionales, en este capítulo explicaremos la más simple: la estructura *if-then*.

Estructura *if-then*

Un condicional *if-then* (si – entonces) posee una estructura como la presentada en la Figura 7-3.

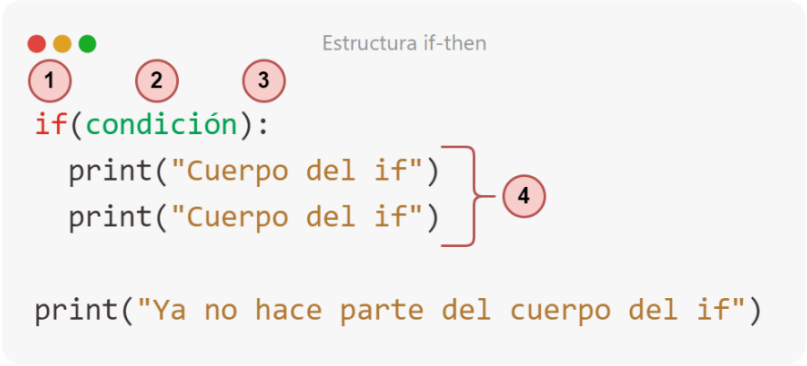


Figura 7-3. Estructura del condicional *if-then*.

El condicional *if-then* posee los siguientes elementos principales:

1. El condicional inicia utilizando la palabra reservada de Python ***if*** (que significa “si” en español).
2. Se define una **condición o expresión** que debe ser evaluable a un valor booleano. Esa condición o expresión se puede o no colocar entre paréntesis (son opcionales).
3. Se agregan dos puntos “**:**”.

4. Se define el **cuerpo del *if***. Allí se coloca la instrucción (o instrucciones) que se ejecutan si la condición o expresión anterior se evalúa a *True*. Esa instrucción (o instrucciones) se deben colocar como bloques de código indentados, es decir, mover de izquierda a derecha (con respecto a la ubicación del *if*) mediante una tabulación.

Ejemplo de diagrama para estructura *if-then*

Antes de codificar nuestro primer condicional, veamos cómo representar mediante un diagrama de flujo un escenario donde necesitemos aplicar condicionales. Supongamos que estamos desarrollando un programa para que nos avise si debemos comprar un helado para refrescarnos. El programa recibirá por pantalla la temperatura actual (en grados centígrados), y si la temperatura es mayor a 27, entonces imprimirá por pantalla el mensaje "Comprar helado". Por último, el programa siempre imprimirá el mensaje "Fin programa" antes de finalizar.

La Figura 7-4 muestra la representación gráfica del programa anterior. En este diagrama, la **sección *if*** (la condición) está representada por el rombo. Y la **sección *then*** está representada por el camino *True*, hasta que se reúne con el camino *False*. En este programa, solo cuando la condición se cumpla, se imprimirá el mensaje "Comprar helado" (se ejecutará la sección *then*).

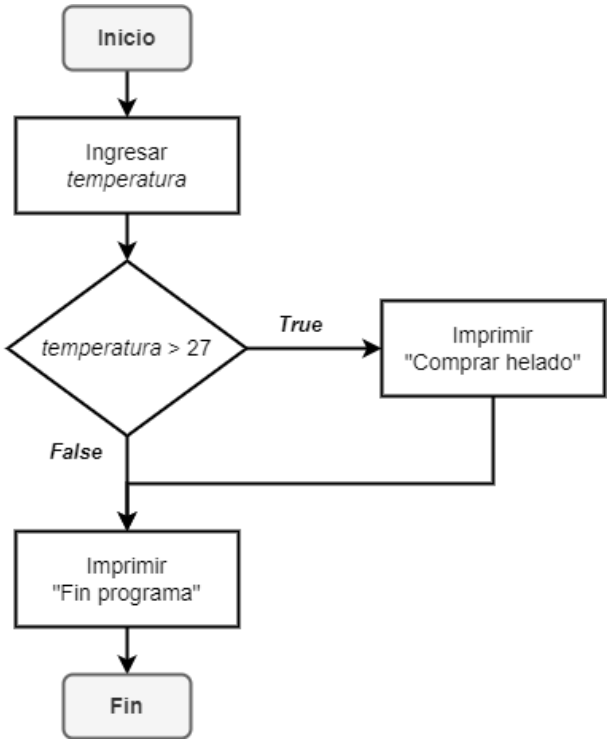


Figura 7-4. Diagrama de flujo del ejercicio "helado" con estructura *if-then*.

Ejemplo de código para estructura *if-then*

El siguiente código muestra la implementación del diagrama de flujo anterior. En la línea 1 se le pide al usuario que ingrese por teclado la temperatura actual. Luego, en la línea 3 se ejecuta el condicional que evalúa si la *temperatura* es mayor a *27*. En caso afirmativo, se ejecuta la línea 4 (ya que corresponde al cuerpo del condicional) y se imprime el mensaje *"Comprar helado"*. Finalmente, se ejecuta la línea 6 y se imprime el mensaje *"Fin programa"*.

Codificar y ejecutar

```
1.  temperatura = float(input("Ingrese la temperatura: "))
2.
3.  if(temperatura > 27):
4.    print("Comprar helado")
5.
6.  print("Fin programa")
```

Si ejecuta el programa anterior e ingresa por teclado el número *28*, obtendrá un resultado como el presentado en la Figura 7-5.

```
Ingrese la temperatura: 28
Comprar helado
Fin programa
```

Figura 7-5. Ejecución del código anterior ingresando *28*.

Discusión rápida: ¿Qué sucede cuándo al ejecutar el programa anterior, el número ingresado no es mayor a 27? Si se ejecuta el programa anterior y se ingresa el número 27, Python ignora (salta) las líneas que corresponden al cuerpo del *if* (aquellas indentadas con respecto al *if*). En este caso, ignora la línea 4. Sin embargo, la línea 6 si será ejecutada, ya que no hace parte del cuerpo del *if*.

Cuerpo de la estructura *if-then*

El cuerpo del condicional *if-then* es aquel que se encuentra en la línea posterior al *if*, y que se encuentra indentado (tabulado). Analicemos en detalle cómo funciona este cuerpo con el siguiente ejemplo. Supongamos que le piden modificar el código anterior, y que cuando la temperatura sea mayor a 27, se imprima el mensaje *"Comprar helado"* pero, se imprima también el mensaje *"Comprar malteada"* (tal como lo que indica la Figura 7-6).

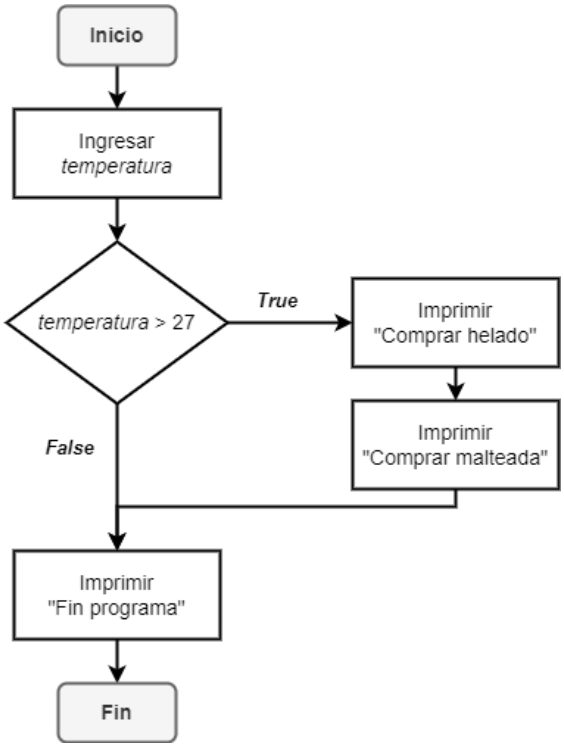


Figura 7-6. Diagrama de flujo del ejercicio “helado” modificado.

El siguiente código muestra un primer intento por resolver el ejercicio anterior. En este código se añadió la línea 5 donde se imprime el mensaje *"Comprar malteada"*. Sin embargo, si ejecutamos el código e ingresamos por pantalla el número *5* para la temperatura, la computadora nos mostraría siempre el mensaje *"Comprar malteada"* y *"Fin programa"*. Lo cual no corresponde con el comportamiento deseado, ya que solo podíamos comprar malteada si la temperatura era mayor a 27.

Codificar y ejecutar

```
1.  temperatura = float(input("Ingrese la temperatura: "))
2.
3.  if(temperatura > 27):
4.    print("Comprar helado")
5.  print("Comprar malteada")
6.
7.  print("Fin programa")
```

El problema se debe a que la línea 5 no pertenece al cuerpo del *if*, esto sucede porque no está indentada. Para solucionarlo, debemos aplicarle una tabulación para que haga parte del cuerpo del *if* (tal como lo muestra el siguiente código). Y ahora nuestro código es coherente con el ejercicio solicitado.

Codificar y ejecutar

```
1.  temperatura = float(input("Ingrese la temperatura: "))
2.
3.  if(temperatura > 27):
4.    print("Comprar helado")
5.    print("Comprar malteada")
6.
7.  print("Fin programa")
```

Discusión rápida: La indentación del código juega un papel crucial en Python. Hay que prestar demasiada atención a cuándo indentar un conjunto de instrucciones y cuándo no hacerlo, ya que puede cambiar totalmente la lógica de nuestro programa.

7.6. Flujo de ejecución del programa

Reconocer cuáles líneas de código se ejecutan y cuáles no, para un programa particular, es muy importante si queremos desarrollar lógica de programación. Hasta antes de este capítulo, dábamos por sentado que todas las líneas de nuestros programas se ejecutaban. Pero con la introducción de condicionales vimos que no es el caso.

Retomemos el ejercicio de "helado" y apliquemos un pequeño cambio, y luego mostraremos el flujo de ejecución del programa. Ahora el programa recibirá por pantalla la temperatura actual (en grados centígrados) y el dinero que tenemos (en dólares). Si la temperatura es mayor a 27 y el dinero es mayor o igual a 5, entonces el programa imprimirá por pantalla el mensaje "Comprar helado". Por último, el programa siempre imprimirá el mensaje "Fin programa" antes de finalizar.

El siguiente código muestra la implementación del ejercicio anterior. Como vemos se añade la línea 2 para solicitar por pantalla la cantidad de *dinero*, y se modifica el condicional en la línea 4 mediante el uso del operador lógico *and* para evaluar ambas comparaciones.

Codificar y ejecutar

```
1. temperatura = float(input("Ingrese la temperatura: "))
2. dinero = int(input("Ingrese el dinero: "))
3.
4. if(temperatura > 27) and (dinero >= 5):
5.   print("Comprar helado")
6.
7. print("Fin programa")
```

Flujo de ejecución para entrada *28* y *6*

Supongamos que ejecutamos el código anterior e ingresamos *28* y *6* respectivamente. La expresión de la línea 4 evaluará a *True*, y, por lo tanto, se ejecutará la línea 5. Si hacemos seguimiento paso a paso a las líneas que se ejecutan, encontramos que se ejecutan las líneas: 1, 2, 4, 5 y 7 (ver paso a paso de ejecución en Figura 7-7). **Nota:** recuerde que los números enmarcados en un círculo con fondo negro indican el paso a paso de ejecución del programa. En este caso, el paso 1 ejecuta la línea 1, el paso 2 la línea 2, el paso 3 la línea 4, el paso 4 la línea 5 y el paso 5 la línea 7.

Figura 7-7. Flujo de ejecución paso a paso del código anterior ingresando *28* y *6* respectivamente.

Flujo de ejecución para entrada *28* y *4*

Ahora, si volvemos a ejecutar el código anterior e ingresamos *28* y *4* respectivamente. La expresión de la línea 4 evaluará a *False*, y, por lo tanto, no se ejecutará la línea 5. Si hacemos

seguimiento paso a paso a las líneas que se ejecutan, encontramos que se ejecutan las líneas: 1, 2, 4 y 7 (ver paso a paso de ejecución en Figura 7-8).

```
1 temperatura = float(input("Ingrese la temp: "))
2 dinero = int(input("Ingrese el dinero: "))
3
4 if(temperatura > 27) and (dinero >= 5):
5     print("Comprar helado")
6
7 print("Fin programa")
```

Figura 7-8. Flujo de ejecución paso a paso del código anterior ingresando *28* y *4* respectivamente.

7.7. Ejercicios

Ejercicios teóricos

E7.1. Analice el siguiente código y mencione cuáles líneas de código se ejecutan.

Analizar

```
1. entradas = 5
2. if(entradas > 5):
3.   print("Tu y tu grupo de amigos pueden ingresar")
4. print("Fin programa")
```

E7.2. ¿Para qué sirve el operador lógico *and*, y cuándo evalúa a *True*?

E7.3. Considerando la variable *carga = 200* y la variable *tipo = "Tractor"*. Diga a que evalúan (*True* o *False*) cada una de las siguientes expresiones.

Expresión	Resultado
(carga == 200) *and* (tipo != "Camión")	¿?
(carga > 350) *or* (tipo == "Tractor")	¿?
(carga <= 600) *and* (tipo != "Tractor")	¿?
not(tipo == "Camión")	¿?

Ejercicios prácticos

E7.4. Implemente un programa en el cual le solicite al usuario que ingrese por teclado la distancia en kilómetros a la que se encuentra su pareja. Si la distancia ingresada es menor o igual a *4* imprima por pantalla *"Ya voy corriendo"*.

E7.5. Implemente un programa en el cual le solicite al usuario que ingrese por teclado el número de la camiseta de un jugador de futbol. Si el número es igual a *19* imprima por pantalla *"Andá pa' allá, bobo"*.

E7.6. Una estación de monitoreo de tsunamis necesita un sistema para emitir alertas. La estación debe emitir una alerta cuando ocurra un sismo de magnitud 5.2 o mayor. Implemente un programa en el cual le solicite al usuario que ingrese por teclado la magnitud de un sismo. Si la magnitud ingresada es mayor o igual a *5.2* imprima por pantalla *"Alerta de tsunami"*.

Ejemplo de entada	Ejemplo de salida
5.4	*Alerta de tsunami*

Resumen

En este capítulo aprendimos las bases para utilizar condicionales en Python y en programación en general. Aprendimos un nuevo tipo de variable (*bool*). Aprendimos a utilizar seis tipos de operadores de comparación (*>*, *>=*, *<*, *<=*, *==* y *!=*). Aprendimos los tres tipos de operadores lógicos (*and*, *or* y *not*). Y finalmente, aprendimos acerca de la estructura condicional *if-then*, y cómo el flujo de ejecución del programa varía dependiendo de cómo se evalúan las expresiones o condiciones en nuestros programas, y de cómo estén indentadas (tabuladas) las líneas debajo del condicional.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una "X" los siguientes trucos: T04, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26 y T27.

En el siguiente capítulo aprenderemos otros tipos de estructuras condicionales.

Capítulo 08 – Condicionales múltiples

En el capítulo anterior aprendimos los fundamentos de los condicionales en Python, en este capítulo, aprenderemos unas estructuras condicionales adicionales y desarrollaremos algunos ejercicios más complejos.

A continuación, desarrollaremos las siguientes secciones:

1. Estructura if-then-else.
2. Estructura if-then-elif-else.
3. Condicionales anidados.
4. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo08. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

8.1. Estructura if-then-else

En el capítulo anterior desarrollamos la estructura *if-then*, la cual permitía ejecutar una serie de instrucciones en caso de que una condición evaluara a *True*. Sin embargo, es común desear ejecutar instrucciones cuando la condición no se cumple (evalúe a *False*). Por ejemplo, calcular el área de un cuadrado cuando se ingrese un valor del lado mayor a cero, pero en caso contrario, imprimir un mensaje de error indicando que el valor del lado de un cuadrado no puede ser negativo.

En el caso de Python, la estructura *if-then* puede ampliarse con el uso de una sección *else* (generando una estructura *if-then-else*). El condicional ***if-then-else*** puede leerse de la siguiente manera: **Si** (*if*) esta condición es verdadera, **entonces** (*then*) ejecute el siguiente código. **De lo contrario** (*else*), ejecute este otro código.

Estructura *if-then-else*

Dado que anteriormente explicamos la estructura *if-then*, aquí solo detallaremos la nueva sección *else*, la cual posee una estructura como la presentada en la Figura 8-1.

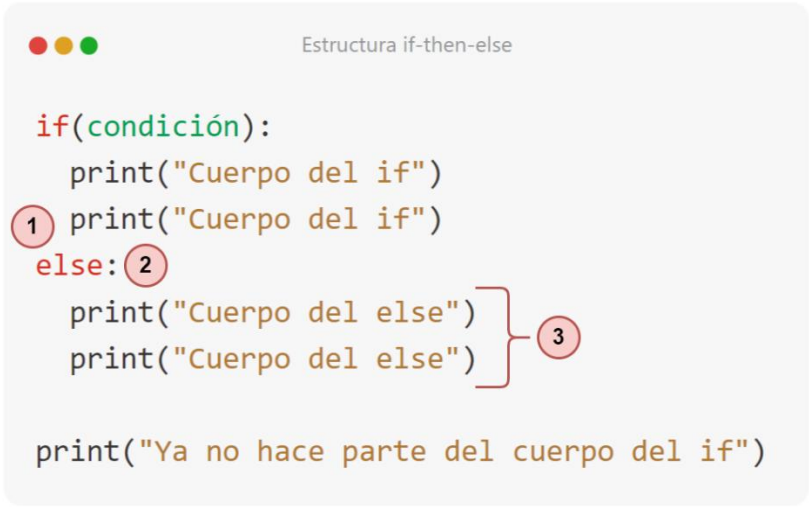


Figura 8-1. Sección *else* de la estructura *if-then-else*.

1. Se utiliza la palabra reservada de Python ***else*** (debe colocarse justo después de la sección *then* del condicional y debe mantener el mismo nivel de indentación que el *if* que lo contiene).
2. Se agregan dos puntos "**:**".
3. Se define el **cuerpo del *else***. Allí se coloca la instrucción (o instrucciones) que se ejecutarían si la condición o expresión (contenida en el *if*) se evalúa a *False*. Esa instrucción (o instrucciones) se deben colocar como bloques de código indentados, es decir, mover de izquierda a derecha (con respecto a la ubicación del *else*) mediante una tabulación.

Ejemplo de diagrama para estructura *if-then-else*

Retomemos el ejemplo del programa de compra de helado, pero con unas pequeñas modificaciones. El programa recibirá por pantalla la temperatura actual (en grados centígrados), y si la temperatura es mayor a 27, entonces imprimirá por pantalla el mensaje "Comprar helado". De lo contrario, el programa imprimirá por pantalla el mensaje "Comprar jugo de naranja". Por último, el programa siempre imprimirá el mensaje "Fin programa" antes de finalizar.

La Figura 8-2 muestra la representación gráfica del programa anterior. En este diagrama, la **sección *if*** (la condición) está representada por el rombo. La **sección *then*** está representada por el camino *True*. Y la **sección *else*** está representada por el camino *False*. En este programa, solo cuando la condición no se cumpla, se imprimirá el mensaje "Comprar jugo de naranja" (se ejecutará la sección *else*).

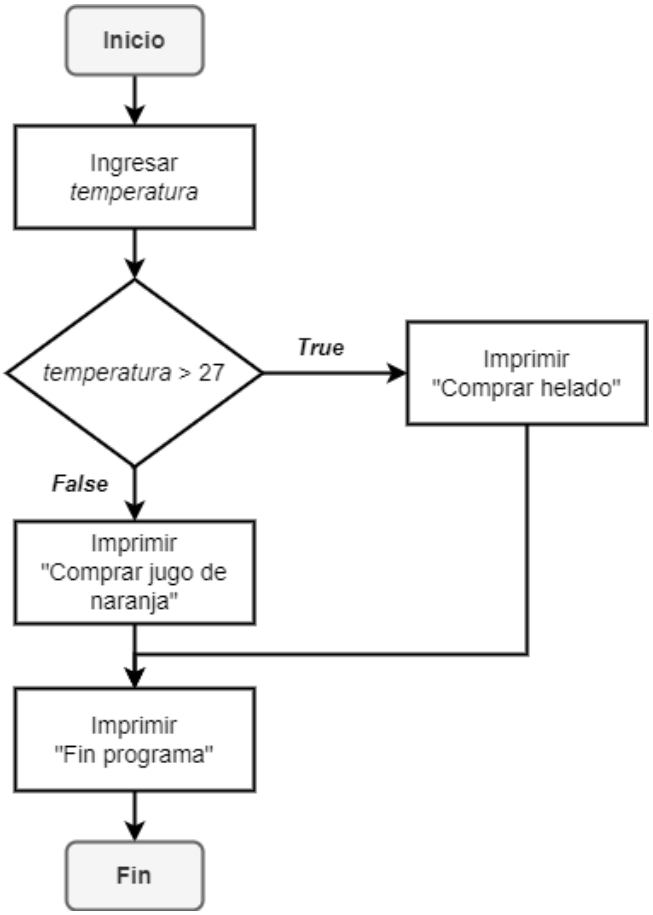


Figura 8-2. Diagrama de flujo del ejercicio “helado” con estructura *if-then-else*.

Ejemplo de código para estructura *if-then-else*

El siguiente código muestra la implementación del diagrama de flujo anterior. Dado que parte de este código ya lo hemos explicado, analicemos con atención los elementos nuevos. En la línea 5 se añade la sección *else*, y en la línea 6 se añade el cuerpo del *else* (la instrucción que se ejecutará cuando la condición evalúe a *False*).

Codificar y ejecutar

```
1.  temperatura = float(input("Ingrese la temperatura: "))
2.
3.  if(temperatura > 27):
4.    print("Comprar helado")
5.  else:
6.    print("Comprar jugo de naranja")
7.
8.  print("Fin programa")
```

Si ejecuta el programa anterior e ingresa por teclado el número *22*, deberá obtener un resultado como el presentado en la Figura 8-3. El programa inicialmente ejecutará la línea 1 (y le asignará a la variable *temperatura* el número *22*), luego ejecutará la línea 3 (aquí el resultado de la verificación será *False*, ya que *22* no es mayor que *27*), luego el programa saltará a la línea 5 (a la sección *else*), ejecutará la línea 6 y finalmente, ejecutará la línea 8.

```
Ingrese la temperatura: 22
Comprar jugo de naranja
Fin programa
```

Figura 8-3. Ejecución del código anterior ingresando *22*.

Ejercicio área del cuadrado

Dado que ya tenemos las herramientas necesarias, ahora podemos solucionar el ejercicio planteado cuando iniciamos el tema de condicionales. Ese ejercicio requería calcular el área de un cuadrado cuando nos ingresaran un lado con valor mayor a cero, e imprimir un mensaje de error para cualquier otro valor.

El siguiente código muestra la solución a ese ejercicio. Pero antes de ejecutarlo, vamos a responder a las siguientes preguntas: (i) si se ejecuta el siguiente programa y se ingresa por teclado el número *3*, ¿cuáles líneas se ejecutan? Y ¿cuántas variables se crearían en memoria? Y (ii) si se ejecuta el siguiente programa y se ingresa por teclado el número *-3*, ¿cuáles líneas se ejecutan? Y ¿cuántas variables se crearían en memoria?

Codificar y ejecutar

```
1.  lado = float(input("Ingrese el lado del cuadrado: "))
2.
3.  if(lado > 0):
4.    area = lado*lado
5.    print("El área del cuadrado es: "+str(area))
6.  else:
7.    print("El lado de un cuadrado no puede ser negativo")
```

Respuesta para ejecución con ingreso de valor *3*: (i) se ejecutarían las líneas 1, 3, 4 y 5. (ii) Se crearían dos variables (*lado* y *area*).

Respuesta para ejecución con ingreso de valor *-3*: (i) se ejecutarían las líneas 1, 3, 6 y 7. (ii) Se crearía una variable (*lado*).

8.2. Estructura if-then-elif-else

Un condicional, sin importar su estructura, puede tener máximo una sección *if*, y una sección *else* (que va al final). Sin embargo, un condicional puede opcionalmente ampliarse con el uso de una o varias secciones *elif*.

Una sección *elif* solo puede colocarse justo después de la sección *if*, o de otra sección *elif*.

La sección ***elif*** (que es la versión abreviada de *else if*), significa en español “de lo contrario, si”. Esta sección permite definir una condición adicional que es evaluada solo si todas las condiciones anteriores fueron falsas, y además define en su cuerpo un conjunto de instrucciones que serán ejecutadas si la condición del *elif* se evalúa como *True*.

Por ejemplo, supongamos que queremos verificar la edad de una persona para asignarla en una etapa de vacunación contra Covid-19. En este caso, un simple *if* y *else* no será suficiente, ya que tenemos muchas etapas de vacunación y múltiples verificaciones. Así, podríamos verificar, si la edad es mayor o igual a 70 años, asignar esa persona a la etapa 1 de vacunación. **De lo contrario, si** la edad es mayor o igual a 60 años y menor a 70 años, asignar esa persona a la etapa 2 de vacunación. **De lo contrario, si** la edad es mayor o igual a 30 años y menor a 60 años, asignar esa persona a la etapa 3 de vacunación. Y de lo contrario, asignar esa persona a etapa 4 de vacunación.

Estructura *if-then-elif-else*

Dado que anteriormente explicamos la estructura *if-then-else*, aquí solo detallaremos la nueva sección *elif*. La cual posee una estructura como la presentada en la Figura 8-4.

Figura 8-4. Sección *elif* de la estructura *if-then-elif-else*.

1. Se utiliza la palabra reservada de Python ***elif*** (debe colocarse luego de un *if* o de otro *elif*).
2. Se define una nueva **condición o expresión** que debe evaluar a un valor booleano. Esta condición solo se evaluará si todas las condiciones anteriores fueron evaluadas como *False*.
3. Se agregan dos puntos "**:**".
4. Se define el **cuerpo del *elif***. Allí se coloca la instrucción (o instrucciones) que se ejecutarían si la condición del *elif* se evalúa a *True*. Esa instrucción (o instrucciones) se deben colocar como bloques de código indentados, es decir, mover de izquierda a derecha (con respecto a la ubicación del *elif*) mediante una tabulación.

Ejemplo de diagrama para estructura *if-then-elif-else*

Retomemos el ejemplo del programa de compra de helado, pero con unas pequeñas modificaciones. El programa recibirá por pantalla la temperatura actual (en grados centígrados), y si la temperatura es mayor a 27, entonces imprimirá por pantalla el mensaje "Comprar helado". De lo contrario, si la temperatura es menor a 15, entonces imprimirá por pantalla el mensaje "Comprar chocolate". De lo contrario, el programa imprimirá por pantalla el mensaje "Comprar jugo de naranja". Por último, el programa siempre imprimirá el mensaje "Fin programa" antes de finalizar.

La Figura 8-5 muestra la representación gráfica del programa anterior. En este diagrama, la **sección *if*** (la condición) está representada por el primer rombo. La **sección *then*** está

representada por el camino *True* del primer rombo. La **sección *elif*** (la otra condición) está representada por el segundo rombo (que conecta con el camino *False* del primer rombo). Y la **sección *else*** está representada por el camino *False* del segundo rombo. En este programa, solo cuando la primera condición no se cumpla y la segunda condición sí se cumpla, se imprimirá el mensaje “Comprar chocolate” (se ejecutará la sección *elif*).

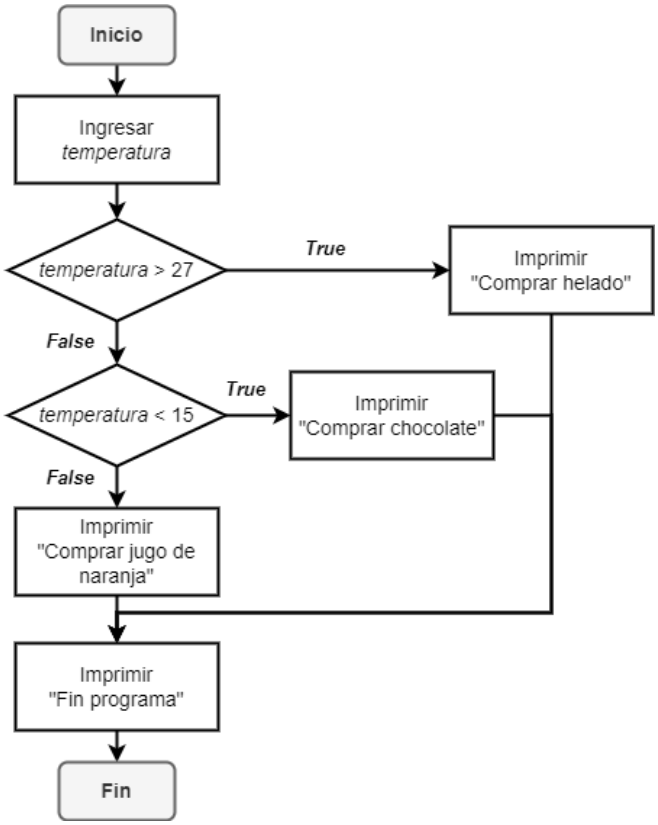


Figura 8-5. Diagrama de flujo del ejercicio “helado” con estructura *if-then-elif-else*.

Ejemplo de código para estructura *if-then-elif-else*

El siguiente código muestra la implementación del diagrama de flujo anterior. Dado que parte de este código ya lo hemos explicado, analicemos con atención los elementos nuevos. En la línea 5 se añade la sección *elif*, a esta sección se le agrega la nueva condición para verificar que la *temperatura* sea menor a *15*. Y en la línea 6 se añade el cuerpo del *elif* (la instrucción que se ejecutará cuando la condición del *elif* evalúe a *True*). Recuerde que el *elif* solo se ejecutará si las condiciones anteriores (en este caso el *if* anterior) evaluaron a *False*.

Codificar y ejecutar

```
1.  temperatura = float(input("Ingrese la temperatura: "))
2.
3.  if(temperatura > 27):
4.    print("Comprar helado")
5.  elif(temperatura < 15):
6.    print("Comprar chocolate")
7.  else:
8.    print("Comprar jugo de naranja")
9.
10. print("Fin programa")
```

Si ejecuta el programa anterior e ingresa por teclado el número *7*, deberá obtener un resultado como el presentado en la Figura 8-6. El programa inicialmente ejecutará la línea 1 (y le asignaría a la variable *temperatura* el número *7*), luego ejecutará la línea 3 (aquí el resultado de la verificación será *False*, ya que *7* no es mayor que *27*). Luego el programa saltará a la línea 5 (a la sección *elif*), en este caso la comparación evaluaría a *True*. Por lo tanto, ejecutará la línea 6. Finalmente, omitirá la sección *else* y ejecutará la línea 10.

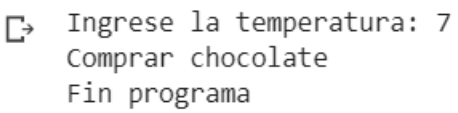


Figura 8-6. Ejecución del código anterior ingresando *7*.

Importancia del orden de las condiciones

Analicemos el siguiente código para responder la siguiente pregunta: ¿Con qué valor de *edad* se imprimiría por pantalla el mensaje “Adulto mayor”?

Codificar y ejecutar

```
1.  edad = int(input("Ingrese su edad: "))
2.
3.  if(edad >= 18):
4.    print("Mayor de edad")
5.  elif(edad >= 60):
6.    print("Adulto mayor")
```

Respuesta: con ninguno. Ya que la primera condición contiene los valores de la segunda condición. Si ingresamos por ejemplo *62*, primero se evaluará el *if* de la línea 3, y 62 es mayor o igual a *18*, entonces imprimiría *"Mayor de edad"* (y el *elif* se omitirá). Para este ejercicio las condiciones debían definirse en orden contrario (primero verificar *edad* mayor o igual a *60*, y luego verificar *edad* mayor o igual a *18*).

Nota: el orden de definición de las condiciones es importante cuando se usa *elif*.

8.3. Condicionales anidados

En programación es muy común contener condiciones dentro de otras condiciones, esto se llama **condicionales anidados**. A veces, primero se debe verificar una condición inicial, y si esa condición se cumple, entonces se realizan otra serie de acciones que pueden incluir nuevas condiciones.

Por ejemplo, vamos a suponer que estamos creando un pequeño sistema de acceso a atracciones de un parque de diversiones. Primero el programa le pedirá al usuario que ingrese el *tipo* de atracción a la que desea ingresar (solo hay dos opciones: 1 para montaña rusa y 2 para viaje en tren).

- Si el usuario **ingresa *1***: (i) el programa le pedirá al usuario que ingrese su *estatura*. (ii) Si la estatura es mayor o igual a 1.61 (vamos a suponer que es la estatura mínima para poder usar la montaña rusa), imprimirá *"Diviértete en la montaña rusa"*. Y (iii) de lo contrario, imprimirá *"Estatura no permitida"*.
- De lo contrario, si el usuario **ingresa *2***, imprimirá *"Diviértete en el tren"*.

Finalmente, siempre al final se imprimirá por pantalla el mensaje *"Fin programa"*.

A continuación, veamos la representación gráfica del ejercicio anterior, y luego su solución en código.

Ejemplo de diagrama para ejercicio “parque de diversiones”

La Figura 8-7 muestra la representación gráfica del programa anterior. En este diagrama, vemos el condicional anidado contenido en el recuadro con línea punteada. En caso de que el usuario seleccione *tipoAtraccion* igual a *1*, entonces luego deberá ingresar su *estatura* y finalmente se ejecutará el segundo condicional (condicional anidado).

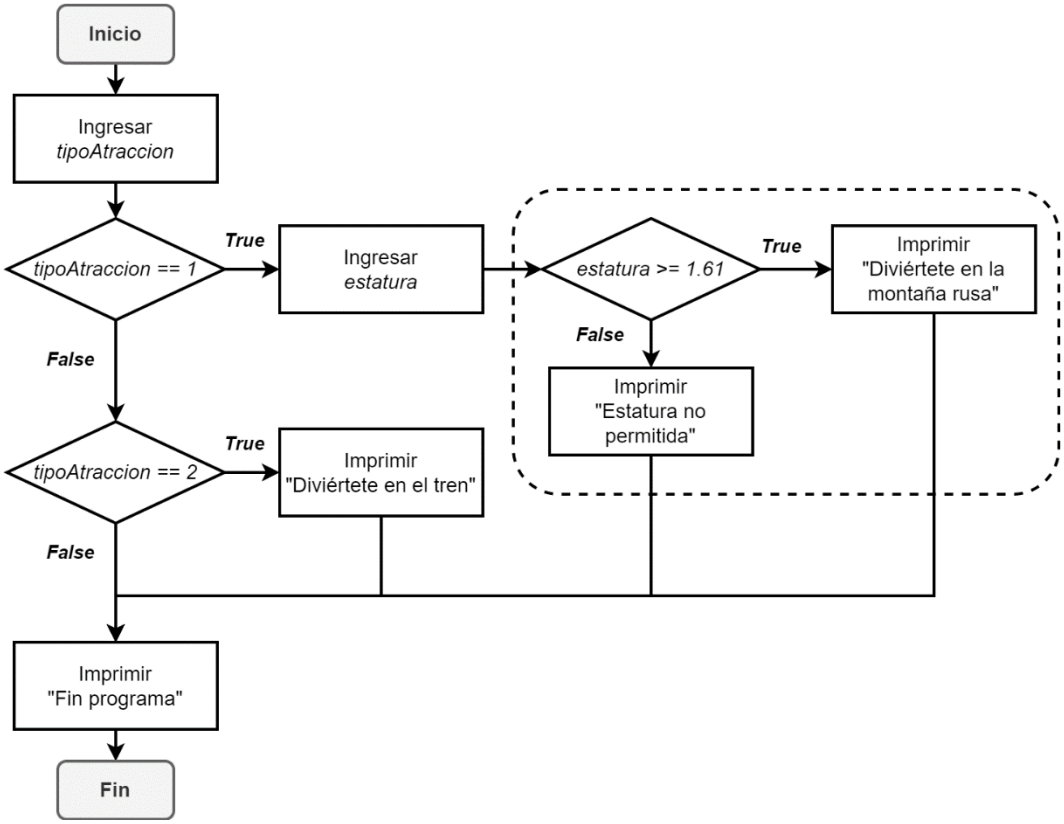


Figura 8-7. Diagrama de flujo del ejercicio “parque de diversiones” con condicional anidado.

Ejemplo de código para ejercicio “parque de diversiones”

El siguiente código muestra la implementación del diagrama de flujo anterior. En este caso nos enfocaremos en explicar el condicional anidado. En la línea 3 se verifica que, si el usuario ingresó *tipoAtraccion* igual a *1*, implica que quiere usar la montaña rusa. Solo para ese caso, se debe realizar una verificación adicional. En este caso, en la línea 4, se le solicita al usuario que ingrese su *estatura*. Y luego, se verifica en la línea 5 que la *estatura* ingresada sea mayor o igual a *1.61*. **Aquí encontramos el condicional anidado**. Dado que es un condicional, contenido en el cuerpo

de otro condicional, para el cuerpo del *if* y el *else* anidados se debe colocar doble tabulación (indentación).

Codificar y ejecutar

```
1.  tipoAtraccion = int(input("Ingrese el tipo 1) montaña 2) tren: "))
2.
3.  if(tipoAtraccion == 1):
4.    estatura = float(input("Ingrese su estatura: "))
5.    if(estatura >= 1.61):
6.      print("Diviértete en la montaña rusa")
7.    else:
8.      print("Estatura no permitida")
9.  elif(tipoAtraccion == 2):
10.   print("Diviértete en el tren")
11.
12. print("Fin programa")
```

Si ejecuta el programa anterior e ingresa por teclado *1* y *1.62* respectivamente, deberá obtener un resultado como el presentado en la Figura 8-8.

```
Ingrese el tipo 1) montaña 2) tren: 1
Ingrese su estatura: 1.62
Diviértete en la montaña rusa
Fin programa
```

Figura 8-8. Ejecución del código anterior ingresando *1* y *1.62* respectivamente.

Discusión rápida: En programación existen diferentes estrategias para la solución de un mismo problema. El ejercicio anterior lo hubiésemos podido solucionar sin condicionales anidados (tal como lo muestra el siguiente código). El problema del siguiente código es que la experiencia de usuario no es la mejor. Ya que sin importar el tipo de atracción que seleccione, el programa siempre pide ingresar la estatura (en la vida real si voy a ingresar a un viaje en tren no debería dar mi estatura). Es por esta razón, que la solución anterior (con condicionales anidados) parece encajar mejor para este problema particular.

Codificar y ejecutar

```
1.  tipoAtraccion = int(input("Ingrese el tipo 1) montaña 2) tren: "))
2.  estatura = float(input("Ingrese su estatura: "))
3.
4.  if(tipoAtraccion == 2):
5.    print("Diviértete en el tren")
6.  elif(tipoAtraccion == 1 and estatura >= 1.61):
7.    print("Diviértete en la montaña rusa")
8.  elif(tipoAtraccion == 1 and estatura < 1.61):
9.    print("Estatura no permitida")
10.
11. print("Fin programa")
```

8.4. Ejercicios

Ejercicios teóricos

E8.1. Analice el siguiente código y mencione cuáles líneas de código se ejecutan, en orden cronológico.

Analizar

```
1.  estrellas = 2
2.  if(estrellas > 3):
3.    print("Que buen servicio")
4.  else:
5.    print("Que mal servicio")
6.  print("Fin programa")
```

E8.2. Verdadero o falso. ¿Una sección *elif* solo se evalúa si todas las condiciones anteriores evaluaron a falso?

E8.3. Analice el siguiente código y diga qué imprime.

Analizar

```
1.  pinturaDerramada = 20
2.  if(pinturaDerramada >= 15):
3.    print("¿Qué es esooo?")
4.  elif(pinturaDerramada > 0):
5.    print("Podrías mejorar")
6.  else:
7.    print("Tienes talento")
8.  print("Fin programa")
```

Ejercicios prácticos

E8.4. Implemente un programa en el cual le solicite al usuario que ingrese por teclado su género musical favorito. Si el género es igual a *"Rock"* imprima por pantalla *"Tienes buen gusto"*, de lo contrario imprima *"Qué asco"*.

E8.5. En el colegio de su hija necesitan que ayude a diseñar un programa para saber quién ganó las elecciones para representante de grupo. Para estas elecciones hay dos estudiantes candidatos, y quien obtuvo la mayor votación es el ganador. Implemente un programa en el cual le solicite al usuario que ingrese por teclado: el nombre del primer estudiante, la votación del primer estudiante, el nombre del segundo estudiante y la votación del segundo estudiante. El programa deberá imprimir por pantalla el nombre del estudiante ganador (si hay empate debe imprimir el mensaje *"Empate"*).

Ejemplo de entada	**Ejemplo de salida**
Michael *40* *Ana* *50*	*Ana*

Resumen

En este capítulo aprendimos nuevas estructuras condicionales. Aprendimos a utilizar el *if-then-else*, el *if-then-elif-else*, y los condicionales anidados. Aprendimos que la sección *elif* se añade a un condicional cuando deseamos realizar múltiples verificaciones (añadir nuevas condiciones). Aprendimos que la sección *else* se añade cuando se quiere ejecutar instrucciones cuando las condiciones anteriores no se cumplieron. Y aprendimos que en algunos casos resulta útil contener condiciones e instrucciones dentro de otras condiciones.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una "X" los siguientes trucos: T28, T29 y T30.

En el siguiente capítulo aprenderemos las nociones básicas de ciclos en programación, mediante el ciclo *while* en Python.

Capítulo 09 – Ciclo while

Los ciclos o bucles son otro de los elementos fundamentales cuando trabajamos con programación. Los ciclos permiten ahorrarle tiempo y líneas de códigos al programador, ya que permiten ejecutar una serie de instrucciones múltiples veces. Python ofrece dos tipos de ciclos *while* y *for*. En este capítulo explicaremos cómo funcionan los ciclos y cómo funciona el ciclo *while*. El ciclo *for* lo desarrollaremos en un capítulo posterior, dado que resulta más fácil entenderlo luego de ver en detalle algunos tipos de variables como textos, listas o diccionarios.

A continuación, desarrollaremos las siguientes secciones:

1. Introducción.
2. Elementos principales de los ciclos.
3. Estructura clásica de un ciclo while.
4. Ejemplo de ciclo while con seguimiento paso a paso.
5. Otros ejemplos y errores comunes.
6. Ciclos controlados por valor centinela.
7. Instrucción break.
8. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo09. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

9.1. Introducción

La repetición de instrucciones o procesos es parte común de los programas que utilizamos a diario. Por ejemplo, una compañía puede contar con algún programa donde se recopile el salario de cada uno de los trabajadores para calcular la nómina total a pagar a fin de mes. Supongamos que la compañía está constituida por 40 trabajadores, con lo visto hasta el momento, sería necesario solicitar por teclado 40 salarios y luego sumarlos e imprimir la suma por pantalla. Cada solicitud de salario sería una línea de código, lo que nos lleva a un programa de mínimo 41 líneas de código. Afortunadamente, Python y otros lenguajes de programación ofrecen estructuras tipo ciclos que permiten repetir un conjunto de instrucciones múltiples veces.

Antes de empezar con ejercicios como el anterior, veamos cómo funcionan los ciclos con casos más simples.

Supongamos que nos piden imprimir por pantalla el mensaje *"Bienvenido a Python"* 3 veces. Si quisiéramos solucionar ese ejercicio con lo visto hasta el momento, deberíamos implementar un código como el que se presenta a continuación.

Codificar y ejecutar

```
1. print("Bienvenido a Python")
2. print("Bienvenido a Python")
3. print("Bienvenido a Python")
```

Ahora, supongamos que nos piden imprimir *"Bienvenido a Python"* 100 veces, tendríamos que repetir la instrucción *print("Bienvenido a Python")* 100 veces. Esto nos implicaría codificar 100 líneas de código. Pero si lo hacemos con un ciclo, ese mismo programa lo podemos codificar con unas cuantas líneas de código.

Un **ciclo** permite agrupar una instrucción o un conjunto de instrucciones que se ejecutarán una y otra vez (mientras se cumpla una condición). Es decir, usando un ciclo le podemos decir a la computadora que muestre por pantalla un texto 100 veces sin tener que codificar la instrucción de “la impresión por pantalla” 100 veces (será suficiente con codificarla una sola vez).

Veamos el siguiente ejemplo (que explicaremos luego):

Codificar y ejecutar

```
1. i = 1
2. while(i <= 3):
3.   print("Bienvenido a Python")
4.   i = i+1
5.
6. print("Fin programa")
```

Nota: si cambiamos el número *3* que aparece en la condición de la línea 2, por el número *100*, y ejecutamos el programa, nos aparecerá 100 veces por pantalla el mensaje *"Bienvenido a Python"*.

9.2. Elementos principales de los ciclos

En general, los ciclos están compuestos de cuatro elementos principales.

1. Inicialización de la variable de control: Se inicializa una variable que servirá para controlar la ejecución del ciclo.
2. Prueba sobre la variable de control: Se realiza una prueba sobre la variable de control para verificar si se puede o no seguir ejecutando el ciclo.
3. Cuerpo del ciclo: Se define la instrucción o conjunto de instrucciones que se ejecutarían repetitivamente mientras la prueba sobre la variable de control evalúe a *True*.
4. Cambio sobre la variable de control: Por lo general, antes de finalizar el cuerpo del ciclo, se realiza un cambio sobre la variable de control, esto permitirá que en algún momento el ciclo se detenga.

El diagrama de flujo de la parte izquierda de la Figura 9-1, muestra la representación gráfica de un ciclo con sus cuatro elementos principales. En la parte derecha de la figura, se desarrolla el ejemplo de imprimir 3 veces el mensaje *"Bienvenido a Python"*. Analizaremos ese diagrama paso a paso. Igualmente, durante todo el libro practicaremos con múltiples ejercicios el uso de ciclos.

Analicemos paso a paso la parte derecha de la Figura 9-1.

1. El programa empieza con la **inicialización de la variable** de control llamada *i*. A esa variable de control se le da un valor inicial de *1*. Para este ejercicio, *i* nos ayudará a llevar el conteo de cuantas veces se ejecuta el ciclo. Si queremos imprimir 3 veces un mensaje, entonces iniciamos en *1*.
2. Luego se desarrolla una **prueba sobre la variable de control**. En este caso vamos a verificar que *i* no pase de 3 (que siempre sea menor o igual a *3*). ¿Por qué 3? Porque es la cantidad de veces que queremos imprimir el mensaje.
3. La prueba evalúa a *True* (dado que *1* es menor o igual a *3*).
4. Se ejecuta el **cuerpo del ciclo**. Ahora se imprime *"Bienvenido a Python"* (llevamos una impresión).
5. Luego, se **cambia la variable de control** *i* aumentándole en *1* su valor (ahora vale *2*).
6. El flujo del programa continúa y nos lleva nuevamente a la **prueba sobre la variable de control**.
7. La prueba evalúa a *True* (dado que *2* es menor o igual a *3*).
8. Se ejecuta el **cuerpo del ciclo**. Ahora se imprime *"Bienvenido a Python"* (llevamos dos impresiones).
9. Luego, **se cambia la variable de control** *i* aumentándole en 1 su valor (ahora vale *3*).
10. El flujo del programa continúa y nos lleva nuevamente a la **prueba sobre la variable de control**.
11. La prueba evalúa a *True* (dado que *3* es menor o igual a *3*).

12. Se ejecuta el **cuerpo del ciclo**. Ahora se imprime *"Bienvenido a Python"* (llevamos tres impresiones).
13. Luego, se **cambia la variable de control** *i* aumentándole en 1 su valor (ahora vale *4*).
14. El flujo del programa continúa y nos lleva nuevamente a la **prueba sobre la variable de control**.
15. La prueba evalúa a *False* (dado que *4* no es menor o igual a *3*).
16. El ciclo se rompe (se ejecuta el camino *False*) e imprimimos *"Fin programa"*.

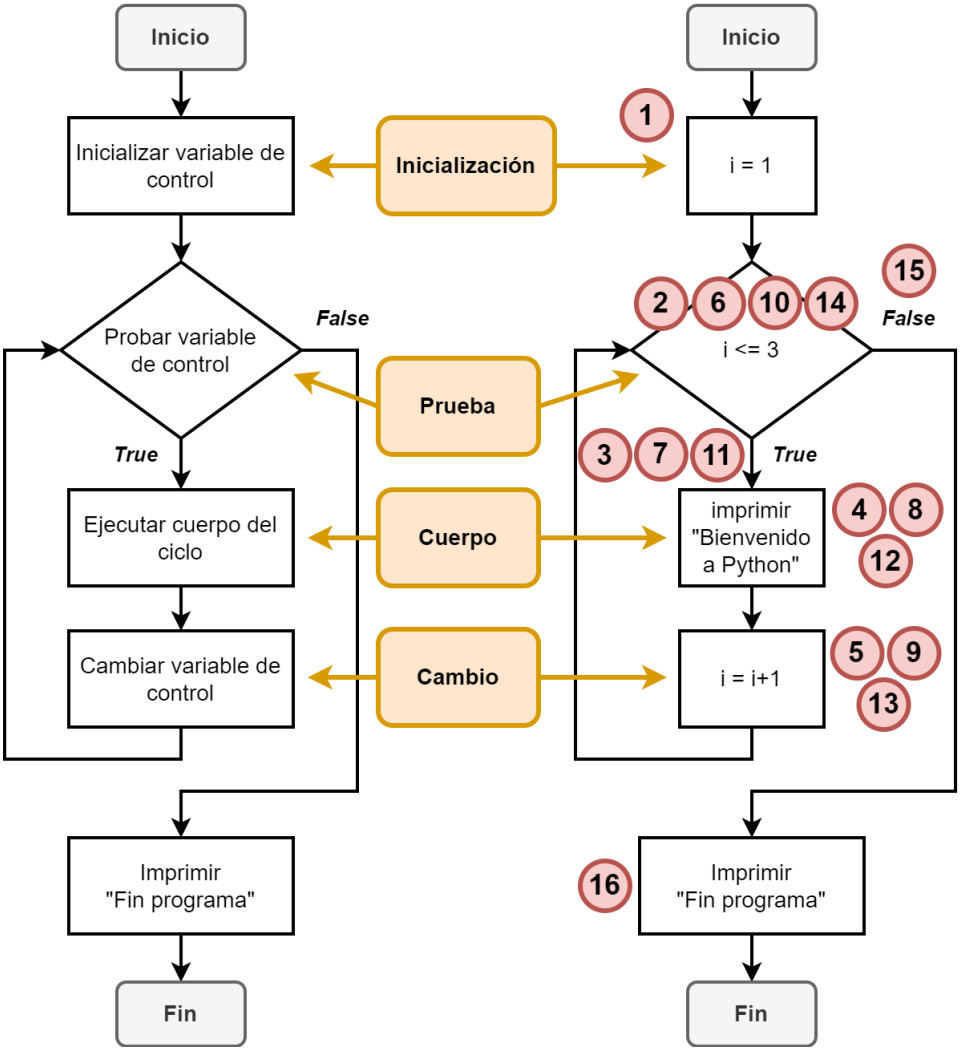


Figura 9-1. Diagrama de flujo de representación de un ciclo.

Luego de haber desarrollado estos conceptos generales de ciclos, veamos cómo funciona el ciclo *while* en Python.

9.3. Estructura clásica de un ciclo while

Un ciclo ***while*** en Python, ejecuta instrucciones repetidamente, mientras que una condición definida en el ciclo sea verdadera (*True*). Un ciclo *while* clásico posee una estructura como la presentada en la Figura 9-2.

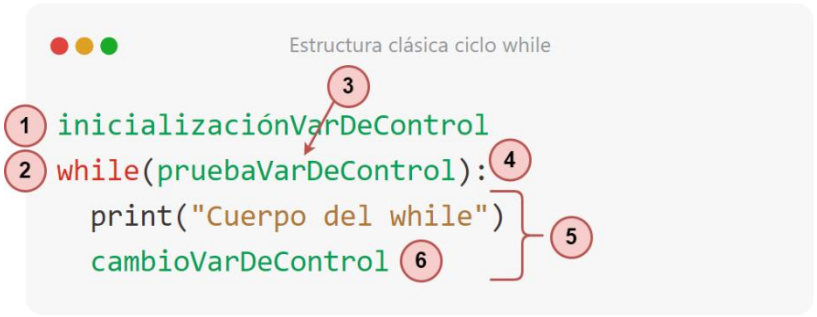


Figura 9-2. Estructura clásica de un ciclo *while*.

1. Se **inicializa** (define) la variable de control.
2. En la línea siguiente se utiliza la palabra reservada de Python ***while*** para empezar a definir el ciclo.
3. Se **prueba** la variable de control (se define una condición). La cual puede ir entre paréntesis (son opcionales).
4. Se agregan dos puntos "**:**".
5. Se define el **cuerpo del *while***. Allí se coloca la instrucción (o instrucciones) que se ejecutarían si la condición o expresión anterior se evalúa a *True*. Esa instrucción (o instrucciones) se deben colocar como bloques de código indentados. Es decir, mover de izquierda a derecha (con respecto al *while*) mediante una tabulación.
6. Antes de finalizar el cuerpo del *while* se realiza un **cambio** sobre la variable de control.

9.4. Ejemplo de ciclo while con seguimiento paso a paso

Supongamos que queremos imprimir los números del 1 al 5. Podríamos tomar como base el código de imprimir 3 veces *"Bienvenido a Python"*, modificar la prueba sobre la variable de control (ahora *i* será menor o igual a *5*), y modificar la línea de impresión para que imprima el valor contenido en nuestra variable de control. A continuación, se muestra el código desarrollado.

Codificar y ejecutar

```
1. i = 1
2. while(i <= 5):
3.   print(i)
4.   i = i+1
5.
6. print("Fin programa")
```

Si ejecuta el programa anterior, debe obtener un resultado como el presentado en la Figura 9-3.

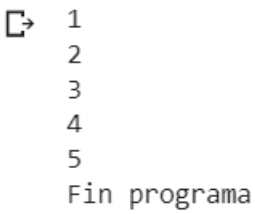


Figura 9-3. Ejecución del código anterior.

Flujo de ejecución

Nota: Usaremos la palabra ***iteración*** para referirnos al conjunto de instrucciones que se deben repetir mientras que una condición (prueba sobre la variable de control) se cumpla.

Hagamos seguimiento, paso a paso del código anterior, presentando atención a las líneas que se ejecutan y en qué orden lo hacen (ver Figura 9-4):

- **Paso 1:** Se ejecuta la línea 1 (la cual asigna a *i* un valor de *1*).
- **Pasos 2, 3 y 4:** Luego se realiza la primera iteración del ciclo (la condición evalúa a *True* dado que *1* es menor o igual a *5*) y se ejecutan las líneas 2, 3 y 4 (ahora *i* vale *2*).
- **Pasos 5, 6 y 7:** Luego se realiza la segunda iteración del ciclo (la condición evalúa a *True* dado que *2* es menor o igual a *5*) y se ejecutan las líneas 2, 3 y 4 (ahora *i* vale *3*).
- **Pasos 8, 9 y 10:** Luego se realiza la tercera iteración del ciclo (la condición evalúa a *True* dado que *3* es menor o igual a *5*) y se ejecutan las líneas 2, 3 y 4 (ahora *i* vale *4*).
- **Pasos 11, 12 y 13:** Luego se realiza la cuarta iteración del ciclo (la condición evalúa a *True* dado que *4* es menor o igual a *5*) y se ejecutan las líneas 2, 3 y 4 (ahora *i* vale *5*).
- **Pasos 14, 15 y 16:** Luego se realiza la quinta iteración del ciclo (la condición evalúa a *True* dado que *5* es menor o igual a *5*) y se ejecutan las líneas 2, 3 y 4 (ahora *i* vale *6*).
- **Paso 17:** Al inicio de la sexta iteración, la condición evalúa a *False* dado que *6* no es menor o igual a *5*. En este caso, el ciclo se rompe.

- **Paso 18:** Se imprime *"Fin programa"*.

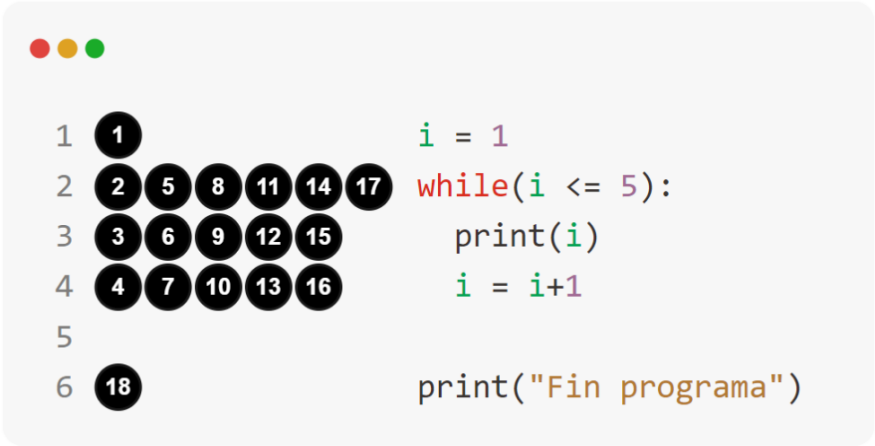


Figura 9-4. Flujo de ejecución paso a paso del código anterior.

TIP: PythonTutor (https://pythontutor.com/visualize.html) es un sitio web donde podemos observar paso a paso la ejecución de nuestros programas (tal cual lo que acabamos de hacer). Allí, el programador puede copiar y pegar el código que desarrolla en Colab y luego dar clic en "Visualize Execution" y empezar a hacer seguimiento paso a paso de qué líneas se van ejecutando. Si es su primera vez programando ciclos, le recomendamos utilizar ese sitio (u otro similar) con los códigos que necesita analizar con detalle.

9.5. Otros ejemplos y errores comunes

Ahora vamos a practicar la implementación de ciclos con tres ejercicios, y luego veremos unos cuantos errores comunes al programar ciclos.

Ejercicios

Ejercicio 1: Supongamos que nos piden imprimir los múltiplos de 5, desde el 5 hasta el 25. El siguiente código muestra la solución a ese ejercicio. Como podemos observar en la línea 1, la variable de control se inicializa con valor de *5* (dado que queremos empezar a imprimir desde el número *5*). En la línea 2, la prueba sobre la variable de control se define como *i* menor o igual a *25* (dado que queremos llegar hasta el número *25*). En la línea 3, se imprime el valor de *i* sobre el que estamos iterando. Y en la línea 4, el cambio en la variable de control se define como un aumento de *5* unidades (dado que vamos de 5 en 5).

Codificar y ejecutar

```
1. i = 5
2. while(i <= 25):
3.   print(i)
4.   i = i+5
```

Ejercicio 2: Supongamos que nos piden imprimir los números desde el 10 hasta el 0 (decrementando de 1 en 1). El siguiente código muestra la solución a ese ejercicio. Como podemos observar en la línea 1, la variable de control se inicializa con valor de *10* (dado que queremos empezar a imprimir desde el número *10*). En la línea 2, la prueba sobre la variable de control se define como *i* mayor o igual a *0* (dado que queremos llegar hasta el número *0* y la *i* parte de un número mayor a *0*). En la línea 3, se imprime el valor de *i* sobre el que estamos iterando. Y en la línea 4, el cambio en la variable de control se define como un decremento de *1* unidad.

Codificar y ejecutar

```
1. i = 10
2. while(i >= 0):
3.   print(i)
4.   i = i-1
```

Ejercicio 3: Supongamos que nos piden imprimir el mensaje *"Python es divertido"* tantas veces como un número que ingrese el usuario por teclado. El siguiente código muestra la solución a ese ejercicio. Como podemos observar en la línea 1, le solicitamos al usuario que ingrese el número de veces que desea imprimir el mensaje, y ese valor lo asignamos a la variable *numero*. Luego, la variable de control se inicializa con valor de *1*. En la línea 3, la prueba sobre la variable de control se define como *i* menor o igual a *numero* (dado que queremos llegar hasta el número ingresado por el usuario). En la línea 4, se imprime el mensaje *"Python es divertido"*. Y en la línea 5, el cambio en la variable de control se define como un aumento de *1* unidad. **Nota:** ejecútelo varias veces, ingresando números diferentes y, observe lo que sucede.

Codificar y ejecutar

```
1.  numero = int(input("Ingrese un número: "))
2.  i = 1
3.  while(i <= numero):
4.    print("Python es divertido")
5.    i = i+1
```

Errores comunes

Error 1: Analice el siguiente código e identifique dónde está el error.

Analizar

```
1.  i = 1
2.  while(i <= 5):
3.    print(i)
```

Respuesta error 1: En el código anterior falta el cambio sobre la variable de control. Si el código se codifica y ejecuta, imprimiría el número *1* de manera infinita. Ya que la prueba sobre la variable de control nunca será *False*.

Error 2: Analice el siguiente código e identifique dónde está el error.

Analizar

```
1.  i = 1
2.  while(i <= 5):
3.    print(i)
4.  i = i+1
```

Respuesta error 2: en el código anterior el cambio sobre la variable de control está mal indentado. Si el código se codifica y ejecuta, imprimiría el número *1* de manera infinita, ya que la variable de control nunca cambiaría su valor, porque el cambio está por fuera del cuerpo del ciclo.

Error 3: Analice el siguiente código e identifique dónde está el error.

Analizar

```
1.  i = 1
2.  while(i >= 0):
3.    print(i)
4.    i = i+1
```

Respuesta error 3: En el código anterior la prueba sobre la variable de control está mal definida. Si el código se codifica y ejecuta, imprimiría los números desde el *1* de forma incremental indefinidamente. Ya que la condición está evaluando que *i* sea mayor o igual a *0*, e *i* siempre será mayor o igual a 0 porque inicia en *1* y el cambio va aumentando su valor.

9.6. Ciclos controlados por valor centinela

Hasta el momento, los ciclos que hemos realizados se han basado en pruebas a variables de control que son numéricas y que no dependen de un valor especial. Conocemos de antemano cuántas veces se va a ejecutar cada ciclo (con un simple cálculo matemático o contando cómo cambia la variable de control).

Sin embargo, a veces queremos controlar ciclos basados en valores especiales que ingresan los usuarios como entradas por teclado. Por ejemplo, crear un ciclo que se ejecute hasta que el usuario ingrese el texto "salir".

A este tipo de ciclos se les conoce como **ciclos controlados por valor centinela**. En este caso, el ciclo solo se detendrá cuando se haya ingresado un valor especial (centinela).

El siguiente código muestra la definición de un ciclo controlado por valor centinela. El propósito es darle la bienvenida únicamente a un usuario cuyo nombre sea Daniel. En la línea 1 se inicializa la variable de control *nombre* como un texto vacío. Luego, en la línea 2 se verifica si *nombre* es diferente a *"Daniel"*. En caso afirmativo, se ejecuta la línea 3 (el cuerpo del ciclo) donde se le solicita al usuario por pantalla que ingrese un nombre, y ese valor se le asigna a la variable *nombre* (el cambio a la variable de control depende del ingreso de un valor por teclado por parte del usuario). Finalmente, cuando el ciclo se rompa, la línea 5 imprime el mensaje *"Bienvenido"*.

Codificar y ejecutar

```
1. nombre = ""
2. while(nombre != "Daniel"):
3.   nombre = input("Ingrese su nombre: ")
4.
5. print("Bienvenido")
```

Si ejecuta el programa anterior e ingresa por teclado *"Sara"*, luego *"Juliana"* y luego *"Daniel"*, deberá obtener un resultado como el presentado en la Figura 9-5.

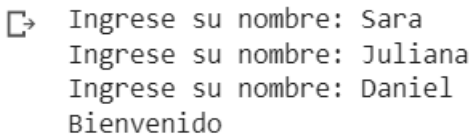


Figura 9-5. Ejecución del código anterior ingresando *"Sara", "Juliana" y "Daniel"*.

Otra opción para definir ciclos controlados por valor centinela, consiste en crear ciclos infinitos y agregarles en el cuerpo del ciclo una condición de parada. Para definir este tipo de ciclos, primero debemos comprender cómo funciona la palabra reservada de Python *break*.

9.7. Instrucción break

La palabra reserva de Python ***break*** permite detener la ejecución de un ciclo en cualquier momento. Si por algún motivo, el programa encuentra y ejecuta una instrucción *break* dentro del cuerpo de un ciclo, inmediatamente se finalizará la ejecución de ese ciclo.

El siguiente código muestra la definición de un ciclo infinito, el cual está controlado por un valor centinela *"Lima"*, y el uso de *break* para detener la ejecución del ciclo. Este código representa una pequeña trivia donde solo la respuesta correcta permitirá terminar la ejecución del juego.

En la línea 1 se define un ciclo infinito (no hay condición a evaluar, siempre será *True*). Luego, en la línea 2 se solicita por teclado al usuario que ingrese el nombre de la capital de Perú. Luego, en la línea 3 se verifica si *capital* es igual a *"Lima"*. En caso afirmativo, se ejecutan las líneas 4 y 5, donde se felicita al usuario por responder correctamente y luego se ejecuta la línea *break* (la cual detiene la ejecución del ciclo). En caso contrario, el ciclo se vuelve a ejecutar. Finalmente, cuando el ciclo se rompa, la línea 7 imprime el mensaje *"Fin programa"*.

Codificar y ejecutar

```
1. while(True):
2.   capital = input("Ingrese la capital de Perú: ")
3.   if(capital == "Lima"):
4.     print("Felicidades ganaste la trivia")
5.     break
6.
7. print("Fin programa")
```

Note que en el programa anterior también se definió un condicional anidado a un ciclo, esto es muy común en programación. Recuerde que debe ser muy cuidadoso a la hora de definir las indentaciones correspondientes para garantizar el funcionamiento correcto del programa.

9.8. Ejercicios

Ejercicios teóricos

E9.1. Analice el siguiente código y mencione cuáles líneas de código se ejecutan.

Analizar

```
1. i = 1
2. while(i < 3):
3.   print(i)
4.   i = i+1
5. print("Fin programa")
```

E9.2. ¿Cuáles son los cuatro elementos principales que componen la mayoría de los ciclos?

E9.3. ¿Qué imprime el siguiente programa si se ingresa *0* por teclado?

Analizar

```
1. i = int(input("Ingrese un número: "))
2. while(i <= 6):
3.   print(i)
4.   i = i+3
```

Ejercicios prácticos

E9.4. Implemente un programa en el cual le solicite al usuario que ingrese por teclado un número. Luego cree un ciclo que se repita tantas veces como el número ingresado (por ejemplo, si ingresa *3*, se deberá crear un ciclo que se ejecute 3 veces). Y luego, dentro del cuerpo del ciclo, imprima cuatro asteriscos *"****"*.

Ejemplo de entada	Ejemplo de salida
3	****
	****
	****

E9.5. Usted está creando un mecanismo de acceso a un sistema. Ese mecanismo consiste en ingresar apropiadamente una palabra clave. Implemente un programa con un ciclo infinito. Luego, en el cuerpo del ciclo, solicítele al usuario que ingrese por teclado una palabra. Si la palabra es diferente a *"Roberto"* imprima el mensaje *"Clave incorrecta"*, y el ciclo se volverá a ejecutar. De lo contrario, imprima *"Clave correcta"* y detenga el ciclo con un *break*.

Ejemplo de entada	Ejemplo de salida
Cielo	*Clave incorrecta*
Martillo	*Clave incorrecta*
Roberto	*Clave correcta*

Resumen

En este capítulo aprendimos las nociones básicas de ciclos en programación. Aprendimos los elementos principales de los ciclos: (i) la inicialización de la variable de control, (ii) la prueba sobre la variable de control, (iii) el cuerpo del ciclo, y (iv) el cambio sobre la variable de control. Aprendimos la estructura clásica de un ciclo *while*. Aprendimos a identificar errores comunes cuando definimos ciclos *while*. Y aprendimos a definir ciclos controlados por valor centinela y el uso de la palabra reservada de Python *break*.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una "X" los siguientes trucos: T31, T32, T33, T34 y T35.

En el siguiente capítulo aprenderemos algunas propiedades principales de las variables tipo *str* y algunas funciones para manipular textos.

Capítulo 10 – Textos

La mayoría de los programas informáticos que utilizamos a diario manipulan textos. Muchos de los sitios web que visitamos nos presentan información en forma de texto, este libro presenta información en forma de texto, los tweets son texto, y así podríamos continuar todo el día. Aprender a manipular esos textos nos dará un superpoder, ya que son demasiadas las aplicaciones que podemos realizar cuando aprendemos a manipular textos.

En este capítulo explicaremos cómo funcionan las variables o valores *str*, algunas de sus propiedades principales y algunas de las operaciones más comunes.

A continuación, desarrollaremos las siguientes secciones:

1. Introducción.
2. Propiedades de los textos.
3. Obtener el tamaño de un texto.
4. Extracción de un carácter o subtexto.
5. Búsqueda de un carácter o subtexto.
6. Otros métodos.
7. Ciclos con textos.
8. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo10. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

10.1. Introducción

Como mencionamos anteriormente, manipular textos es un proceso muy común en los programas que usamos a diario. Veamos unos cuantos ejemplos de la importancia de manipular textos:

- Cuando busca el nombre de un amigo en Facebook, Facebook compara que ese nombre parcialmente coincida con algunos de los nombres de su lista de amigos.
- Cuando se conecta a Gmail, Gmail compara que su usuario y contraseña coincidan con los almacenados en el sistema de acceso de usuarios.

- Si sabes manipular textos, podrías extraer toda la información de un archivo Excel y manipularla en tus programas para encontrar información relevante y calcular promedios, mínimos, máximos, entre otros.
- Otros casos útiles incluyen: filtros de spam, sistema de detección de intrusos, motores de búsqueda, detección de plagio, bioinformática, análisis forense digital, raspado web (*web scraping*) y sistemas de recuperación de información.

Variables *str*

En capítulos anteriores aprendimos a definir variables y valores *str*. El siguiente código consolida lo aprendiendo hasta el momento. En la línea 1 vemos cómo definir una variable *str* con comillas dobles. En la línea 2 repetimos el proceso, pero con comillas simples. En la línea 3 mostramos como concatenar textos. En la línea 5 y 6 vemos cómo imprimir por pantalla una variable *str* y un valor *str* respectivamente. Finalmente, en la línea 7 vemos cómo imprimir el tipo de una variable, en este caso la salida en pantalla será *str*. La Figura 10-1 muestra el resultado por pantalla.

Codificar y ejecutar

```
1. nombre = "Juliana"
2. apellido = 'Parra'
3. nombreCompleto = nombre + " " + apellido
4.
5. print(nombreCompleto)
6. print("30 años")
7. print(type(nombreCompleto))
```

```
Juliana Parra
30 años
<class 'str'>
```

Figura 10-1. Ejecución del código anterior.

Además de lo anterior, las variables y valores *str* tienen propiedades y operaciones muy interesantes que analizaremos en las próximas secciones.

10.2. Propiedades de los textos

Toda variable o valor *str* en Python, tiene tres propiedades que exploramos en detalle en esta sección:

1. Los *str* están compuestos por **caracteres**, los cuales son letras o símbolos individuales. Por ejemplo, el *str "Hola 1"* está compuesto por 6 caracteres: el carácter “H”, el carácter “o”, el carácter “l”, el carácter “a”, el carácter “ ” (un espacio en blanco), y el carácter “1”.
2. Los *str* tienen un **tamaño (*longitud*)**, el cual es el número de caracteres que contiene el *str*. Por ejemplo, el *str "Hola 1"* tiene un tamaño de 6.
3. Los caracteres en un *str* aparecen en una secuencia ascendente. Lo cual significa que cada carácter tiene un **número de posición** (también llamado **índice**) dentro del *str*. Los índices inician desde el número 0. Por ejemplo, para el *str "Hola 1"*, el carácter “H” se encuentra en el índice 0, y el carácter “1” se encuentra en el índice 5.

Repasando las propiedades de los textos

Analicemos el siguiente código.

Analizar

```
1. mensaje = "Hola, Grupo"
```

Basados en la variable *mensaje* (que se detalla en la Figura 10-2), responda: (i) ¿Cuál es el tamaño de *mensaje*?, (ii) ¿Qué carácter se encuentra en la posición 0?, (iii) ¿Qué carácter se encuentra en la posición 5? Y (iv) ¿Qué carácter se encuentra en la posición 10?

mensaje	H	o	l	a	,		G	r	u	p	o
posición	0	1	2	3	4	5	6	7	8	9	10

Figura 10-2. Representación gráfica de la variable *mensaje*.

Respuestas: (i) *11*, (ii) *H*, (iii) un espacio en blanco y (iv) *o*.

10.3. Obtener el tamaño de un texto

Para obtener el tamaño de un texto podemos utilizar la función *len*. La función ***len()*** recibe un argumento, ya sea una variable que contiene un *str* o un valor *str*, y retorna el número de elementos (en este caso caracteres) que contiene la variable o valor enviado.

El siguiente código muestra cómo utilizar la función *len* para calcular el número de caracteres de una variable *str* y un valor *str*. En la línea 3 se obtiene el tamaño del valor almacenado en *mensaje*, y se asigna a la variable *tamaño1*. En la línea 4 se obtiene el tamaño del valor "Medellín" y se asigna a la variable *tamaño2*. Finalmente, los tamaños calculados se imprimen en las líneas 6 y 7 (ver ejecución en Figura 10-3).

Codificar y ejecutar

```
1. mensaje = "Hola, Grupo"
2.
3. tamaño1 = len(mensaje)
4. tamaño2 = len("Medellín")
5.
6. print(tamaño1)
7. print(tamaño2)
```

```
11
8
```

Figura 10-3. Ejecución del código anterior.

10.4. Extracción de un carácter o de un subtexto

Primero veamos cómo extraer un carácter, y luego cómo extraer un subtexto (conjunto de caracteres).

Extracción de un carácter

Para extraer un carácter de un texto particular debemos seguir el procedimiento que se menciona a continuación. Primero colocamos el **nombre de la variable** donde se almacena el texto, luego abrimos y cerramos corchetes "**[]**", y en el medio de los corchetes colocamos la **posición** del carácter que queremos extraer. Este procedimiento nos retornará el carácter en la posición especificada.

El siguiente código muestra cómo extraer diversos caracteres de un texto.

- En la línea 2 extraemos el carácter en la posición *0* de la variable *mensaje*, la cual contiene el texto *"Hola, Grupo"* (o sea la *H*) y lo asignamos a *caracterPos0*.
- En la línea 3 extraemos el carácter en la posición *4* de la variable *mensaje*, la cual contiene el texto *"Hola, Grupo"* (o sea la coma *","*) y lo asignamos a *caracterPos4*.
- En la línea 4 estamos tratando de extraer el carácter en la última posición de la variable *mensaje*. Para esto, entre los corchetes, primero calculamos el tamaño de *mensaje* y le restamos una unidad, ya que la última posición siempre será el tamaño menos una unidad (debido a que las posiciones inician desde cero).
- Finalmente, los caracteres extraídos se imprimen en las líneas 6 a 8 (ver Figura 10-4).

Codificar y ejecutar

```
1. mensaje = "Hola, Grupo"
2. caracterPos0 = mensaje[0]
3. caracterPos4 = mensaje[4]
4. caracterPosFinal = mensaje[len(mensaje)-1]
5.
6. print(caracterPos0)
7. print(caracterPos4)
8. print(caracterPosFinal)
```

```
H
,
o
```

Figura 10-4. Ejecución del código anterior.

Analice rápidamente el siguiente código e indique qué imprimiría.

Codificar y ejecutar

```
1. marca = "Tesla"
2. caracterPos6 = marca[6]
3. print(caracterPos6)
```

Respuesta: imprimiría un error. La línea 2 está mal definida, ya que la variable *marca* no contiene un carácter en la posición *6* (los caracteres llegan hasta la posición 4).

Extracción de un subtexto

A veces queremos extraerle una porción o subtexto específico a un texto. Por ejemplo, de un texto muy largo, queremos obtener solo la porción donde aparece el precio de un producto o la dirección de una tienda (alguna vez Daniel implementó algo parecido en una empresa para la que trabajó).

Para extraer una serie de caracteres (o también llamado subtexto) de un texto particular debemos seguir el procedimiento que se describe a continuación. Primero colocamos el **nombre de la variable** donde se almacena el texto, luego abrimos y cerramos corchetes "**[]**", y en el medio de los corchetes colocamos: (i) la **posición inicial** desde la cual queremos empezar a extraer caracteres, (ii) luego dos puntos "**:**" y (iii) la **posición final** hasta la cual queremos extraer. Este

procedimiento nos retornará el subtexto extraído. **Nota:** Python no incluirá en la extracción el carácter ubicado en la posición final especificada. Por lo tanto, siempre defina la posición final como la ubicación del último carácter deseado a extraer, más una unidad.

El siguiente código muestra cómo extraer un par de subtextos (para extraer la parte del texto que hace alusión al precio de un producto y el precio de dicho producto).

- En la línea 2 extraemos desde el carácter en la posición *0* hasta el carácter en la posición *6* (recuerde que el carácter ubicado en la posición final *7* no se incluye) de la variable *texto*, o sea extraemos el subtexto *Precio:* y lo asignamos a *subTexto1*.
- En la línea 3 extraemos desde el carácter en la posición *8* hasta el carácter en la posición *10* (recuerde que el carácter ubicado en la posición final *11* no se incluye), o sea extraemos el subtexto *250* (el precio del producto) y lo asignamos a *subTexto2*.
- Finalmente, los subtextos extraídos se imprimen en las líneas 5 y 6 (ver Figura 10-5).

Codificar y ejecutar

```
1. texto = "Precio: 250"
2. subTexto1 = texto[0:7]
3. subTexto2 = texto[8:11]
4.
5. print(subTexto1)
6. print(subTexto2)
```

```
Precio:
250
```

Figura 10-5. Ejecución del código anterior.

10.5. Búsqueda de un carácter o subtexto

Buscar caracteres o subtextos dentro de un texto es una operación muy común cuando se manipulan textos. Por ejemplo, existen sistemas que inspeccionan los mensajes que enviamos en diferentes redes sociales, para verificar que el mensaje no contiene *spam* (o contenido basura), o enlaces a páginas con virus, entre otros.

Para buscar un carácter o un subtexto dentro de una variable o valor *str*, debemos utilizar el **método *find*** y el siguiente procedimiento. Primero colocamos el **nombre de la variable** donde se almacena el texto, luego colocamos un punto "**.**", luego colocamos ***find()***, y dentro de los paréntesis colocamos el **subtexto** que deseamos buscar. Este subtexto a buscar debe estar entre comillas o debe ser el nombre de una variable que contenga otro texto. El método *find* retornará:

(i) la posición en la que aparece por primera vez el subtexto dentro del texto. O (ii) retornará *-1* si no se encuentra el subtexto dentro del texto.

El siguiente código muestra cómo buscar un par de caracteres y un subtexto en un texto particular.

- En la línea 2 buscamos el carácter *"o"* dentro del texto contenido en la variable *mensaje*. En este caso, el método *find* nos retornará la posición *1*, ya que allí aparece por primera vez el carácter *"o"*, y ese valor quedará asignado a la variable *posO*.
- En la línea 3 buscamos el subtexto *"Grupo"* dentro del texto contenido en *mensaje*. En este caso, el método *find* nos retornará la posición *6*, ya que desde allí empieza a aparecer ese subtexto, y ese valor quedará asignado a la variable *posGrupo*.
- En la línea 4 buscamos el carácter *"Z"* dentro del texto contenido en *mensaje*. En este caso, el método *find* nos retornará la posición *-1*, ya que *"Z"* no se encuentra en el texto, y ese valor quedará asignado a la variable *posZ*.
- Finalmente, las posiciones encontradas se imprimen en las líneas 6 a 8 (ver Figura 10-6).

Codificar y ejecutar

```
1. mensaje = "Hola, Grupo"
2. posO = mensaje.find("o")
3. posGrupo = mensaje.find("Grupo")
4. posZ = mensaje.find("Z")
5.
6. print(posO)
7. print(posGrupo)
8. print(posZ)
```

```
1
6
-1
```

Figura 10-6. Ejecución del código anterior.

Por último, analicemos el siguiente código para entender mejor la diferencia entre una función y un método.

Analizar

```
1. mensaje = "Hola, Grupo"
2. posH = mensaje.find("H")
3. print(posH)
4.
5. edad = 28
6. pos8 = edad.find(8)
```

En la línea 2 del código anterior observamos el uso del método *find*. Un **método** siempre estará asociado a una variable o valor de un tipo particular. Allí, vemos que el método *find* está asociado a la variable *mensaje* (a través de un punto en el medio). El método *find* solo está disponible para valores o variables *str*. En la línea 6 vemos cómo se está tratando de utilizar el método *find* con una variable *int*. Si ejecutamos el código, la línea 6 generará un error, ya que para valores o variable *int* no existe un método *find* disponible. Por otro lado, en la línea 3 observamos la utilización de la función *print*. Una **función** nunca estará asociada a una variable o valor de un tipo particular.

Discusión rápida: Siendo más precisos, los métodos deben estar asociados a objetos de una clase, o a una clase particular. Los conceptos de "objetos" y "clases" corresponden a un tipo de programación llamada "programación orientada a objetos" que Python soporta y usa extensivamente. Este es un tema que no desarrollaremos en este libro. Pero si este libro tiene éxito (y los lectores lo solicitan), podríamos desarrollarlo en un próximo libro.

Búsqueda de un carácter o subtexto basado en una posición inicial

En los ejemplos anteriores vimos cómo el método *find* recibía un argumento (el carácter o subtexto a buscar). Otra variante del método *find* recibe dos argumentos (el primer argumento es el carácter o subtexto a buscar, y el segundo argumento es la posición desde la cual empezar a buscar). Esto se usa cuando queremos buscar la segunda aparición de un subtexto particular, o cuando utilizamos como referencia una posición particular para empezar a buscar un subtexto.

Para entender cómo funciona *find* con dos argumentos, veamos el siguiente ejemplo. Supongamos que un trabajador de un supermercado utiliza el siguiente formato para almacenar la información de cada producto. Primero coloca el nombre del producto, luego un guion "-", luego el precio del producto, luego otro guion "-" y luego la cantidad disponible de ese producto.

Analizar

```
NombreProducto-Precio-Cantidad
```

Ahora supongamos, que desarrollamos un programa donde ese formato lo asignamos a una variable *str*. Con la manipulación de textos y el uso de *find* con dos argumentos, podríamos extraer el valor de la cantidad de cualquier producto sin importar la longitud de su nombre o la de su precio (siempre y cuando se utilice un formato como el anterior).

El siguiente código muestra cómo utilizar los elementos anteriores, para extraer la cantidad de un producto basado en el formato anterior.

- En la línea 2 buscamos el carácter "-" dentro del texto contenido en la variable *producto*. En este caso, el método *find* nos retornará la posición *5*, ya que allí aparece por primera vez ese carácter, y ese valor quedará asignado a la variable *posGuion1*.
- En la línea 3 buscamos el carácter "-" dentro del texto contenido en la variable *producto*, pero esta vez, a partir de la posición encontrada anteriormente más uno (+1), esto quiere decir que vamos a buscar la segunda aparición del carácter "-". En este caso, el método *find* nos retornará la posición *9*, ya que allí aparece por segunda vez ese carácter, y ese valor quedará asignado a la variable *posGuion2*.
- En la línea 4 extraemos desde el carácter en la posición *posGuion2+1* (es decir, desde la posición siguiente en la que se encuentra el segundo guion encontrado), hasta el carácter en la última posición (es decir, *len(producto)*). Así que extraemos el subtexto *20* (la cantidad del producto) y lo asignamos a *subTextoCantidad*.
- Finalmente, las posiciones encontradas y el subtexto extraído se imprimen en las líneas 6 a 8 (ver Figura 10-7). **Nota:** si lo desea, modifique los datos del producto (su nombre, precio y cantidad), y desde que respete el formato anterior, al final el código siempre le deberá mostrar la cantidad del producto.

Codificar y ejecutar

```
1. producto = "Laser-100-20"
2. posGuion1 = producto.find("-")
3. posGuion2 = producto.find("-", posGuion1+1)
4. subTextoCantidad = producto[posGuion2+1:len(producto)]
5.
6. print(posGuion1)
7. print(posGuion2)
8. print(subTextoCantidad)
```

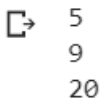


Figura 10-7. Ejecución del código anterior.

Con este ejemplo, adquirimos habilidades para extraer diferentes porciones de diferentes textos. Podríamos analizar textos de miles o cientos de miles de caracteres, y extraer información valiosa de ellos practicando lo anterior.

10.6. Otros métodos

Además del método *find*, existen otros 47 métodos disponibles para variables y valores *str*. Estos métodos se pueden encontrar en la documentación oficial de Python en el siguiente enlace: https://docs.python.org/es/3/library/stdtypes.html#string-methods. Obviamente, el libro quedaría muy extenso si los explicamos todos, pero veamos en el siguiente código algunos de estos métodos en acción. Esta vez, trate de entender por usted mismo qué está pasando y cómo funcionan esos métodos (dejamos pistas en forma de comentario). Y luego ejecute el código y vea los resultados de la Figura 10-8. Y si tiene alguna duda, recuerde que puede utilizar la **"zona de discusión"** del repositorio del libro para realizar preguntas.

Codificar y ejecutar

```
1.  mensaje = "Ubicación: provenza"
2.
3.  mensajeMinus = mensaje.lower() # retorna el texto en minúscula
4.  print(mensajeMinus)
5.
6.  mensajeMayus = mensaje.upper() # retorna el texto en mayúscula
7.  print(mensajeMayus)
8.
9.  conteo = mensaje.count("a") # cuenta cantidad de "a"
10. print(conteo)
11.
12. nuevoMensaje = mensaje.replace("a", "x") # retorna reemplazando "a" por "x"
13. print(nuevoMensaje)
```

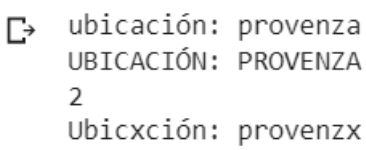


Figura 10-8. Ejecución del código anterior.

10.7. Ciclos con textos

Veamos cómo podemos utilizar los ciclos para recorrer textos. El siguiente código muestra cómo recorrer y mostrar los caracteres de una palabra ingresada por teclado (ver ejecución en Figura 10-9).

- En la línea 1 le pedimos al usuario que ingrese una palabra por teclado, y asignamos ese valor a la variable *palabra*. Para explicar este ejercicio supongamos que ingresamos la palabra *Carro*.
- En la línea 3 definimos la variable de control *i* inicializada en *0*. La variable de control nos servirá para controlar el ciclo, pero, además, nos indicará en qué posición del texto ingresado estamos en cada iteración. Inicializamos la variable *i* en *0*, ya que representa la posición del primer carácter que existe en el texto ingresado.
- En la línea 4 creamos el ciclo *while* y definimos la condición. Este ciclo se ejecutará siempre y cuando *i* sea menor que el tamaño del contenido de la variable *palabra*. El tamaño varía dependiendo de lo que ingresemos por pantalla. Si continuamos con el ejemplo, el tamaño del texto ingresado sería *5*, dado que *Carro* contiene *5* caracteres. Esto quiere decir que *i* siempre tendrá que ser menor a *5* (lo cual representa posiciones válidas a las que podemos acceder del texto ingresado).
- En la línea 5 inicia el cuerpo del *while*. En este caso, accedemos e imprimimos el carácter en la posición *i* almacenado en *palabra*. La primera vez que se ejecute el ciclo accederemos a *palabra[0]*, o sea al carácter *C*.
- En la línea 6 realizamos el cambio de la variable de control. A *i* le aumentaremos una unidad (dado que queremos ir pasando por todas las posiciones del texto ingresado). En este caso, *i* empezó siendo *0*, y ahora será *1*. Luego, el código volverá a ejecutar la línea 4 y verificar si el ciclo se puede repetir o no.
- Cuando *i* sea *1*, la línea *5* imprimirá *palabra[1]* (o sea *a*). Luego, *i* aumentará una unidad y será *2* y la línea *5* imprimirá *palabra[2]* (o sea *r*). Luego, *i* aumentará una unidad y será *3* y la línea *5* imprimirá *palabra[3]* (o sea *r*). Luego, *i* aumentará una unidad y será *4* y la línea *5* imprimirá *palabra[4]* (o sea *o*). Y luego, *i* aumentará una unidad y será *5* y el ciclo finalizará su ejecución.

Codificar y ejecutar

```
1.  palabra = input("Ingrese una palabra: ")
2.
3.  i = 0
4.  while(i < len(palabra)):
5.    print(palabra[i])
6.    i = i+1
```

```
Ingrese una palabra: Carro
C
a
r
r
o
```

Figura 10-9. Ejecución del código anterior ingresando *Carro*.

10.8. Ejercicios

Ejercicios teóricos

E10.1. Diga qué imprime el siguiente programa.

Analizar

```
1.  texto = "Gatubela"
2.  print(texto[0])
3.  print(texto[0:3])
4.  pos1 = texto.find("u")
5.  print(pos1)
```

E10.2. ¿Para qué sirve el segundo argumento que se le puede enviar al método *find* (cuando trabajamos con variables o valores *str*)?

E10.3. ¿Qué imprime el siguiente programa si se ingresa por teclado *Saludo*?

Analizar

```
1. palabra = input()
2. print(palabra[len(palabra)-1])
```

Ejercicios prácticos

E10.4. Implemente un programa en el cual le solicite al usuario que ingrese por teclado un texto. Luego imprima el primer carácter de ese texto concatenado con el último carácter de ese texto.

Ejemplo de entada	**Ejemplo de salida**
Ropa Cara	*Ra*

E10.5. Una bibliotecaria tiene que analizar textos de diferentes libros. Pero la bibliotecaria odia analizar los textos que contienen la letra "z". La bibliotecaria le pide el favor de desarrollar un programa que le informe si un texto contiene alguna letra "z". Implemente un programa en el cual le solicite al usuario que ingrese por teclado un texto. Luego, si el texto contiene por lo menos una letra "z", imprima por pantalla "*Texto contiene letra z*". En caso contrario, imprima "*Texto no contiene letra z*". **Pista:** para este ejercicio puede utilizar *count* o *find*. Si utiliza *find*, recuerde que el método *find* retorna *-1* si no encuentra un carácter o subtexto dentro de un texto.

Ejemplo de entada	**Ejemplo de salida**
Amarillo	*Texto no contiene letra z*

E10.6. Un filólogo (persona que estudia las lenguas) está realizando una investigación para comunicarse con alienígenas. Todos sabemos que los alienígenas no utilizan vocales. Por lo tanto, el filólogo desea contar con un programa, que, al ingresarle una palabra, muestre cada letra de esa palabra en una línea independiente. Y si alguna letra corresponde a una vocal, no la muestre. Implemente un programa en el cual le solicite al usuario que ingrese por teclado una palabra. Luego, imprima por pantalla cada carácter de la palabra en una línea independiente (sin incluir los caracteres que representan vocales).

Ejemplo de entada	**Ejemplo de salida**
Motor	*M* *T* *r*

Pista: complete todas las líneas en blanco del siguiente código.

Analizar

```
1.
2.
3.
4.    if(palabra[i] != "a" and palabra[i] != "e" and palabra[i] != "i" and palabra[i] != "o" and
      palabra[i] != "u"):
5.        print(palabra[i])
6.
```

Resumen

En este capítulo aprendimos a manipular textos. Aprendimos las propiedades principales de los textos: (i) están compuestos por caracteres, (ii) tienen un tamaño y (iii) los caracteres se almacenan en posiciones ascendentes iniciando en cero. Aprendimos diferentes operaciones sobre textos, tales como: (i) obtener el tamaño de un texto con la función *len*, (ii) extraer un carácter de un texto con los corchetes [] y una posición particular, (ii) extraer un subtexto de un texto con los corchetes [] y la posición inicial y final, y (iv) buscar caracteres o subtextos dentro de un texto utilizando el método *find*. Aprendimos que las variables y valores *str* contienen otro grupo de métodos interesantes que podríamos utilizar en nuestros programas. Y finalmente, aprendimos cómo recorrer textos con el uso de ciclos.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una “X” los siguientes trucos: T36, T37, T38, T39, T40, T41, T43, T44, T45 y T46.

En el siguiente capítulo aprenderemos un nuevo tipo de variable llamado lista (*list*) que resulta clave para diseñar programas más complejos.

Capítulo 11 – Listas

Cuando nuestros programas empiezan a contener muchos datos e información, es común observar la definición de decenas o cientos de variables. Pero esto hace que el programa sea más largo, difícil de modificar y evolucionar. Para este problema particular, las listas vienen a nuestro rescate. Una lista permite almacenar múltiples valores en una única variable, y así podemos reducir la cantidad de variables, y hacer nuestros programas más cortos, simples y entendibles.

En este capítulo explicaremos cómo funcionan las listas, algunas de sus propiedades principales y algunas de las operaciones más comunes.

A continuación, desarrollaremos las siguientes secciones:

1. Definición de listas.
2. Propiedades de las listas.
3. Visualización de listas en memoria.
4. Operaciones básicas.
5. Ciclos con listas.
6. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo11. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

11.1. Definición de listas

Antes de definir una lista, veamos el problema que resuelven las listas. Supongamos que nos piden almacenar la edad de 100 trabajadores de una empresa (100 números enteros). Con lo visto hasta el momento, tendríamos que definir 100 variables y almacenar en cada una de ellas el valor de la edad de un trabajador (algo como lo que se presenta en el siguiente código).

Analizar

```
1.    edad1 = 31
2.    edad2 = 25
...   ...
100.  edad100 = 45
```

Pero, con una lista, podemos definir una sola variable que contenga las 100 edades.

Definición de una lista

En Python, las **listas** permiten almacenar múltiples valores asignados a una única variable. En una lista (*list*) podríamos almacenar letras del alfabeto, nombres de personas del grupo familiar, edades, salarios, entre muchos otros. El siguiente código muestra como definir una lista.

Codificar

```
1.  edades = [28, 67, 40, 15]
```

La definición de una lista posee una estructura como la presentada en la Figura 11-1.

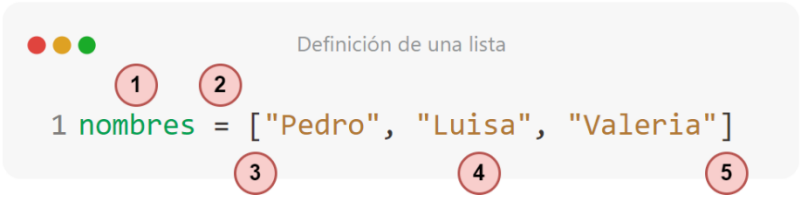


Figura 11-1. Estructura de definición de una lista.

1. Se define el **nombre de una variable** (comúnmente en plural dado que esperamos almacenar múltiples valores).
2. Se coloca el signo igual "**=**".
3. Se abre un corchete "**[**".
4. Se colocan los **valores individuales** que deseamos almacenar en la lista **separados por comas**.
5. Y se cierra un corchete "**]**".

El siguiente código muestra cómo definir una lista que almacena nombres, y luego imprimir su contenido y su tipo. La Figura 11-2 muestra la salida por pantalla de la ejecución del código. Allí vemos que la lista almacena tres valores *str*, y luego vemos que el tipo de la variable *nombres* es *list*.

Codificar y ejecutar

```
1. nombres = ["Pedro", "Luisa", "Valeria"]
2. print(nombres)
3. print(type(nombres))
```

```
['Pedro', 'Luisa', 'Valeria']
<class 'list'>
```

Figura 11-2. Ejecución del código anterior.

Listas que almacenan diferentes tipos de datos

En Python, las **listas** permiten almacenar valores de diferentes tipos. Por ejemplo, podemos almacenar un texto, luego un entero y luego un booleano, todo en una misma lista. Aunque esto es posible, por lo general no se recomienda. Cuando deseemos almacenar valores de distintos tipos, se recomienda utilizar un diccionario (tema que se explica en el siguiente capítulo).

Por ahora veamos cómo definir una lista que almacena diferentes tipos de datos. El siguiente código muestra la definición de una lista que almacena valores *str*, *int* y *float*.

Codificar y ejecutar

```
1. misDatos = ["Luisa", 37, "lmalvarez33@eafit.edu.co", 1.67]
2. print(misDatos)
```

Ahora piense a qué información corresponde cada valor. Por ejemplo, ¿Qué representa el *37*? Muchos dirán que la edad de la persona, pero en realidad podría representar cualquier otro valor, como la temperatura corporal en grados centígrados. Por eso, para estos casos, los diccionarios funcionan mejor (pero lo veremos luego).

11.2. Propiedades de las listas

Toda variable o valor *list* en Python, tiene tres propiedades que exploramos en detalle en esta sección (muy similares a las de los *str*):

1. Las *list* están compuestos por **elementos** (valores separados por comas).
2. Las *list* tienen un **tamaño (*length*)**, el cual corresponde a la cantidad de elementos que contiene la lista.
3. Los elementos en una *list* aparecen en una secuencia ascendente. Lo cual significa que cada elemento tiene un **número de posición** (también llamado **índice**) dentro del *list*. Las posiciones inician desde 0.

Repasando las propiedades de las listas

Analicemos el siguiente código.

Analizar

```
1. nombres = ["Pedro", "Luisa", "Valeria"]
```

Basándose en la variable *nombres* (que se detalla en la Figura 11-3), responda: (i) ¿Cuál es el tamaño de *nombres*?, (ii) ¿Qué elemento se encuentra en la posición 0?, (iii) ¿Qué elemento se encuentra en la posición 2? Y (iv) ¿De qué tipo es el elemento que se encuentra en la posición *0*?

nombres	"Pedro"	"Luisa"	"Valeria"
posición	0	1	2

Figura 11-3. Representación gráfica de la variable *nombres*.

Respuestas: (i) *3*, (ii) *"Pedro"*, (iii) *"Valeria"* y (iv) *str*.

11.3. Visualización de listas en memoria

Retomemos la representación gráfica de cajas que utilizamos en secciones anteriores, para representar la variable que se define en el siguiente código.

Codificar

```
1. misDatos = ["Luisa", 18, 1.67]
```

La Figura 11-4 muestra la representación gráfica de la variable *misDatos* previamente definida. Analicemos varios elementos de esta figura.

- El papel que contiene *misDatos* almacena una referencia con un número (un identificador). Esto se debe a que las variables tipo *list* almacenan **referencias**. Una referencia representa una ubicación en la memoria de la computadora, en la que los datos de la variable están almacenados.
- Para diferenciar referencias de valores, el papel que se coloca dentro de la caja ahora contiene la palabra *REF*, seguida de dos puntos y un identificador.
- Para el ejemplo anterior (y para variables tipo *list* en general), imagínese la referencia *ID: 452* como una caja grande que almacena por dentro tres cajas pequeñas (los elementos de la lista).

- Por último, se almacena en la posición *0* el texto *"Luisa"*, en la posición *1* el número *18* y en la posición *2* el flotante *1.67*.

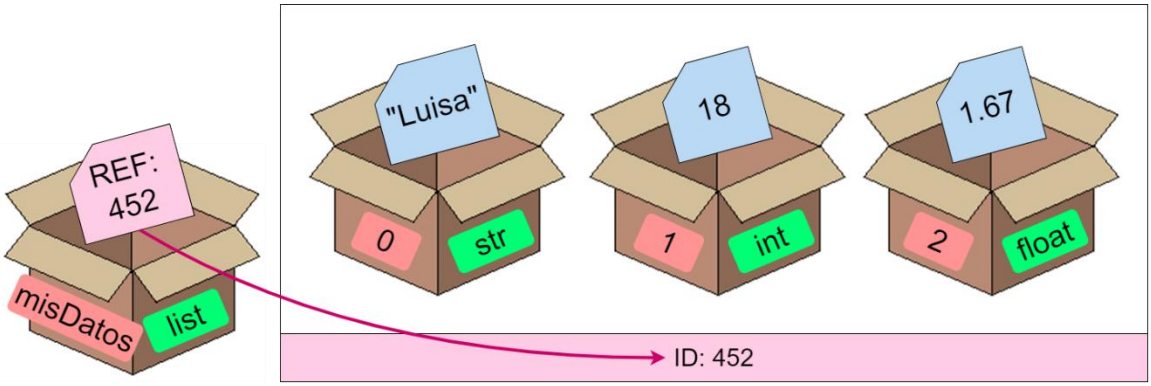


Figura 11-4. Representación gráfica de la variable *misDatos*.

Valores y referencias

Veamos en el siguiente código la definición de dos tipos de variables diferentes y cómo se almacenan en memoria esas variables.

Codificar

```
1. edad = 48
2. hobbies = ["Leer"]
```

La Figura 11-5 muestra la representación gráfica de ambas variables. En la línea 1 definimos la variable *edad*, la cual almacena un número *int*. Hasta el momento para variables *int*, *str*, *float* y *bool* **imaginaremos*** que esas variables almacenan valores. En la línea 2, definimos la variable *hobbies* como una lista que contiene un solo elemento. Para variables *list* imaginaremos que esas variables almacenan referencias.

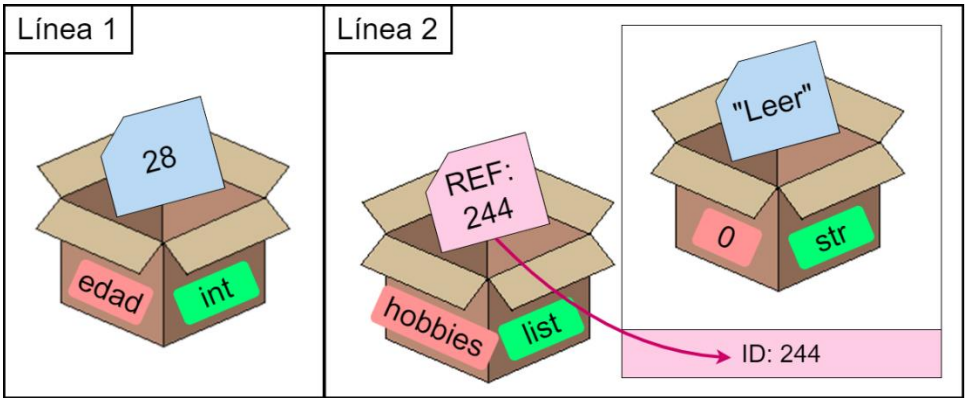


Figura 11-5. Representación gráfica de las variables *edad* y *hobbies*.

***Discusión rápida**: Si se percató, en la explicación anterior para variables *int*, *str*, *float* y *bool* utilizamos **"imaginaremos" con un asterisco**. Esto debido a que esa explicación es una simplificación de lo que realmente pasa en Python. En Python, a diferencia de otros lenguajes, todos los tipos de variables almacenan referencias. Pero en este libro, imaginaremos que las variables *int*, *str*, *float* y *bool* almacenan valores. Para programadores principiantes, esto simplifica la explicación y hace más simple entender lo que pasa en el código. Sin embargo, si desea obtener más información acerca de cómo funcionan las referencias para todas las variables en Python, podría leer este artículo de médium: https://medium.com/geekculture/python-reference-e6458a9a0582 .

11.4. Operaciones básicas

A continuación, analizaremos algunas de las operaciones básicas para trabajar con listas. Las cuales incluyen: (i) acceder a elementos de una lista, (ii) modificar elementos de una lista, (iii) agregar elementos a una lista, y (iv) eliminar elementos de una lista.

Acceder a elementos de una lista

Para acceder a un elemento de una lista debemos seguir el procedimiento presentado a continuación. Primero colocamos el **nombre de la variable** donde se almacenará la lista. Luego abrimos y cerramos corchetes "**[]**", y en el medio de los corchetes colocamos la **posición** del elemento al cual queremos acceder. Este procedimiento nos retornará el elemento a acceder.

El siguiente código muestra cómo acceder a diferentes elementos de una lista (ver ejecución en Figura 11-6).

- En la línea 1 creamos una lista de cuatro cantantes y la asignamos a la variable *cantantes*.

- En la línea 2 accedemos a la cantante en la posición *0* de *cantantes* (o sea a *"Shakira"*) y la asignamos a la variable *cantante1*.
- En la línea 4 imprimimos el tamaño de la lista cantantes (o sea *4*).
- En la línea 5 imprimimos el valor almacenado en *cantante1* (o sea *"Shakira"*).
- En la línea 6 imprimimos el cantante en la posición *2* de la lista (o sea *"J Balvin"*).
- En la línea 7 imprimimos el cantante en la última posición de la lista (o sea *"Maluma"*). Recuerde que para obtener la última posición de una lista o de un texto, se obtiene el tamaño de la lista o texto y se le resta una unidad.

Codificar y ejecutar

```
1. cantantes = ["Shakira", "Karol G", "J Balvin", "Maluma"]
2. cantante1 = cantantes[0]
3.
4. print(len(cantantes))
5. print(cantante1)
6. print(cantantes[2])
7. print(cantantes[len(cantantes)-1])
```

```
4
Shakira
J Balvin
Maluma
```

Figura 11-6. Ejecución del código anterior.

Analice rápidamente el siguiente código y piense qué imprimiría.

Codificar y ejecutar

```
1. cantantes = ["Shakira", "Karol G", "J Balvin", "Maluma"]
2. cantante6 = cantantes[6]
3. print(cantante6)
```

Respuesta: imprimiría un error. La línea 2 está mal definida, ya que la variable *cantantes* no contiene un elemento en la posición *6* (la lista llega hasta la posición 3).

Modificar elementos de una lista

Para modificar un elemento de una lista debemos seguir el procedimiento presentado a continuación. Primero colocamos el **nombre de la variable** en la que se almacenará la lista. Luego abrimos y cerramos corchetes "**[]**". En el medio de los corchetes colocamos la **posición** del elemento que queremos modificar. Luego colocamos un igual "**=**". Y finalmente, colocamos el **nuevo valor** que queremos asignar a esa posición.

El siguiente código muestra cómo modificar un elemento de una lista (ver ejecución en Figura 11-7).

- En la línea 1 creamos una lista de tres marcas de motocicletas.
- En la línea 2 accedemos e imprimimos el valor almacenado en la posición *0* de *motocicletas* (o sea *"Honda"*).
- En la línea 4 modificamos la motocicleta en la posición *0* de la lista y le cambiamos su valor a *"Ducati"*.
- En la línea 5 accedemos e imprimimos el valor almacenado en la posición *0* de *motocicletas* (ahora es *"Ducati"*).

Codificar y ejecutar

```
1. motocicletas = ["Honda", "Yamaha", "Suzuki"]
2. print(motocicletas[0])
3.
4. motocicletas[0] = "Ducati"
5. print(motocicletas[0])
```

```
Honda
Ducati
```

Figura 11-7. Ejecución del código anterior.

Agregar elementos a una lista

Agregar elementos es una tarea común, por ejemplo, puede que quiera agregar un nuevo alien a la lista de aliens (en un videojuego que está programando). Para agregar un elemento a una lista debemos utilizar el método *append* y seguir el procedimiento que se describe a continuación. Primero, colocamos el **nombre de la variable** donde se almacenará la lista. Luego colocamos un punto "**.**". Después colocamos ***append()***, y dentro de los paréntesis colocamos el **nuevo elemento** a agregar. El método *append* permite agregar un elemento al final de una lista específica.

El siguiente código muestra cómo agregar varios elementos a una lista (ver ejecución en Figura 11-8).

- En la línea 1 creamos una lista *aliens* vacía (nótese que no colocamos elementos entre los corchetes).
- En la línea 2 agregamos el valor *"E.T."* al final de *aliens* (en este caso se agregará en la posición 0 dado que la lista estaba vacía).
- En las líneas 3 a 5 continuamos agregando diferentes aliens a la lista.
- En la línea 6 accedemos e imprimimos el valor almacenado en la posición *3* de *aliens*.

Codificar y ejecutar

```
1. aliens = []
2. aliens.append("E.T.")
3. aliens.append("Goku")
4. aliens.append("Marvin")
5. aliens.append("Gazoo")
6. print(aliens[3])
```

```
Gazoo
```

Figura 11-8. Ejecución del código anterior.

Eliminar elementos de una lista

Para eliminar un elemento a una lista debemos utilizar el método *pop* y seguir el procedimiento que se describe a continuación. Primero colocamos el **nombre de la variable** donde se almacenará la lista, luego colocamos un punto "**.**", después colocamos ***pop()***, y dentro de los paréntesis colocamos la **posición del elemento** a eliminar. Una vez eliminado el elemento en esa posición, todos los elementos siguientes se moverán hacia la posición anterior. Por ejemplo, si tenemos una lista con tres elementos (posiciones 0, 1 y 2) y eliminamos el elemento de la posición 1, el que estaba en la posición 2 pasa a la posición 1).

El siguiente código muestra cómo eliminar un elemento de una lista (ver ejecución en Figura 11-9).

- En la línea 1 creamos una lista con cuatro *aliens*.
- En la línea 2 accedemos e imprimimos el valor almacenado en la posición *1* de *aliens* (o sea *"Goku"*).
- En la línea 3 eliminamos el valor almacenado en la posición *1* de la lista (es decir, *"Goku"*).
- En la línea 4 accedemos e imprimimos nuevamente el valor almacenado en la posición *1* de *aliens*. Esta vez saldrá por pantalla *"Marvin"*, dado que antes se encontraba en la

posición *2*, pero al eliminar a *"Goku"* todos los valores siguientes se movieron hacia la posición inmediatamente anterior.

Codificar y ejecutar

```
1. aliens = ["E.T.", "Goku", "Marvin", "Gazoo"]
2. print(aliens[1])
3. aliens.pop(1)
4. print(aliens[1])
```

```
Goku
Marvin
```

Figura 11-9. Ejecución del código anterior.

TIP: Existen muchos otros métodos disponibles para trabajar con listas. Por lo tanto, le recomendamos dar un vistazo rápido a este enlace: https://www.w3schools.com/python/python_ref_list.asp. Si lo prefiere, puede traducir la página a español.

11.5. Ciclos con listas

Una vez definimos listas en nuestros programas, será muy común recorrerlas para mostrar sus elementos. La gran mayoría de programas que utilizamos a diario contienen listas con diversos elementos. Por ejemplo, una tienda en línea es común que agrupe sus productos en una lista, y cuando accedemos a su sitio web, nos va mostrando cada producto almacenado en esa lista. Un periódico en línea podría tener sus noticias en una lista, y nos va mostrando en su página principal algunas de sus últimas noticias. Otros ejemplos incluyen: lista de reproducción de música en Spotify, lista de videos recomendados en YouTube o lista de películas en Netflix.

El siguiente código muestra cómo recorrer y mostrar los elementos de una lista (ver ejecución en Figura 11-10).

- En la línea 1 creamos una lista con cuatro *aliens*.
- En la línea 3 definimos la variable de control *i* inicializada en *0*. La variable de control nos servirá para controlar el ciclo, pero, además, nos indicará en qué posición de la lista estamos en cada iteración. Dado que queremos mostrar todos los elementos de la lista, la variable la inicializamos en *0*.
- En la línea 4 creamos el ciclo y definimos la condición. En este caso el ciclo se ejecutará siempre y cuando *i* sea menor que el tamaño de la lista *aliens*. El tamaño de *aliens* es *4*, o sea que *i* siempre tendrá que ser menor a *4* (lo cual representa posiciones válidas a las que podemos acceder de la lista).

- En la línea 5 inicia el cuerpo del *while*. En este caso, accedemos e imprimimos el elemento almacenado en la lista *aliens*, en la posición *i* (la primera vez que se ejecute accederemos a *aliens[0]*, o sea a *"E.T."*).
- En la línea 6 realizamos el cambio de la variable de control. A *i* le aumentaremos una unidad (dado que queremos ir pasando por todas las posiciones de la lista). En este caso, *i* empezó siendo *0*, y ahora será *1*. Luego el código volverá a ejecutar la línea 4 y verificar si el ciclo se puede repetir o no.
- Cuando *i* sea *1* la línea *5* imprimirá *"Goku"*. Luego, *i* aumentará una unidad y será *2* y la línea *5* imprimirá *"Marvin"*. Luego *i* aumentará una unidad y será *3* y la línea *5* imprimirá *"Gazoo"*. Y luego *i* aumentará una unidad y será *4* y el ciclo finalizará su ejecución (la Figura 11-11 muestra el flujo de ejecución paso a paso).

Codificar y ejecutar

```
1. aliens = ["E.T.", "Goku", "Marvin", "Gazoo"]
2.
3. i = 0
4. while(i < len(aliens)):
5.   print(aliens[i])
6.   i = i+1
```

```
E.T.
Goku
Marvin
Gazoo
```

Figura 11-10. Ejecución del código anterior.

Figura 11-11. Flujo de ejecución paso a paso del código anterior.

11.6. Ejercicios

Ejercicios teóricos

E11.1. Diga qué imprime el siguiente programa.

Analizar

```
1. colores = ["Amarrillo", "Azul", "Rojo"]
2. print(len(colores))
3. print(colores[1])
```

E11.2. Considerando el código del ejercicio anterior, mencione qué instrucción debería codificar si quiere eliminar el valor *"Azul"* de la lista *colores*.

E11.3. ¿Qué imprime el siguiente programa?

Analizar

```
1. colores = ["Amarrillo", "Azul", "Rojo"]
2. print(colores[len(colores)])
```

Ejercicios prácticos

E11.4. Implemente un programa en el cual le solicite al usuario que ingrese por teclado cuatro nombres de sus películas favoritas. Luego añada cada uno de esos nombres a una lista, y finalmente imprima la lista.

Ejemplo de entada	**Ejemplo de salida**
Duna *Interestelar* *Divergente* *Tron*	*['Duna', 'Interestelar', 'Divergente', 'Tron']*

E11.5. Unos alienígenas de otra galaxia dejaron un mensaje secreto en una pieza de código de Python que se muestra a continuación:

Analizar

```
1. mensajeSecreto = ["Planeta", "Más", "Secreto", "Alienigena", "Zombie", "Serás", "Nuclear",
   "Vos"]
```

Ahora tenemos que descifrar ese mensaje. Para eso, tome la lista anterior, y luego recorra la lista con un ciclo e imprima por pantalla únicamente los valores almacenados en las posiciones impares de la lista (o sea la 1, 3, 5 y 7). **Pista:** analice bien, con qué valor debería iniciar la variable de control, y con qué valor debe cambiarla.

Resumen

En este capítulo aprendimos a trabajar con listas. Aprendimos las propiedades principales de las listas: (i) que están compuestas por elementos, (ii) que tienen un tamaño y (iii) que los elementos se almacenan en posiciones ascendentes iniciando en cero. Aprendimos cómo se visualizan listas en memoria, y la diferencia entre valores y referencias. Aprendimos cuatro operaciones básicas cuando trabajamos con listas: (i) acceder a un elemento particular de una lista, (ii) modificar un elemento particular de una lista, (iii) agregar un elemento a una lista (utilizando el método *append*) y (iv) eliminar un elemento de una lista (utilizando el método *pop*). Y finalmente, aprendimos cómo recorrer listas con el uso de ciclos.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una “X” los siguientes trucos: T05, T47, T48, T49, T50, T51, T52, T53 y T54.

En el siguiente capítulo aprenderemos un nuevo tipo de variable llamado diccionario (*dict*), el cual es similar a las listas.

Capítulo 12 – Diccionarios

Los diccionarios, tal como las listas, permiten almacenar múltiples valores asignados a una única variable. La gran diferencia es que a los diccionarios podemos definirle las posiciones o índices con textos. Textos que nos ayudarán a recordar fácilmente qué estamos almacenando en cada posición.

En este capítulo explicaremos cómo funcionan los diccionarios, algunas de sus propiedades principales y algunas de las operaciones más comunes.

A continuación, desarrollaremos las siguientes secciones:

1. Definición de diccionarios.
2. Propiedades de los diccionarios.
3. Visualización de diccionarios en memoria.
4. Operaciones básicas.
5. Ciclos con listas que contienen diccionarios.
6. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo12. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

12.1. Definición de diccionarios

En Python, los **diccionarios** permiten almacenar múltiples valores asignados a una única variable (tal como las listas). En un diccionario (*dict*) podríamos almacenar todos los datos de una persona, de un vehículo, de un producto, entre otros. La gran diferencia entre diccionarios y listas es que los diccionarios contienen una estructura de *clave/valor* que hace que organizar y acceder a los datos del diccionario sea fácil y flexible.

El siguiente código muestra cómo definir un diccionario.

Codificar

```
1.  misDatos = {
2.    "nombre":"Luisa",
3.    "edad":18,
4.    "estatura":1.67
5.  }
```

La definición de un diccionario posee una estructura como la presentada en la Figura 12-1.

Figura 12-1. Estructura de definición de un diccionario.

1. Se define el **nombre de una variable** (que contendrá los datos del diccionario).
2. Se coloca el operador igual "**=**".
3. Se abre llave "**{**".
4. Se colocan parejas **key:value** (clave:valor) con los datos que queremos almacenar separados por comas.
5. Y se cierra llave "**}**".

El siguiente código muestra cómo definir un diccionario que almacena los datos de un producto, y luego cómo imprimir su contenido y su tipo. La Figura 12-2 muestra la salida por pantalla de la ejecución del código. Allí vemos que el diccionario almacena dos valores *str* y un valor *int*, y luego vemos que el tipo de la variable *producto* es *dict*.

Codificar y ejecutar

```
producto = {
  "nombre":"iPhone 14",
  "precio":799,
  "color":"plata"
}
print(producto)
print(type(producto))
```

```
{'nombre': 'iPhone 14', 'precio': 799, 'color': 'plata'}
<class 'dict'>
```

Figura 12-2. Ejecución del código anterior.

12.2. Propiedades de los diccionarios

Toda variable o valor *dict* en Python, tiene tres propiedades que exploramos en detalle en esta sección:

1. Los *dict* están compuestos por un conjunto de parejas **key:value** (clave:valor).
2. Los *dict* tienen un **tamaño (*length*)**, el cual corresponde a la cantidad de parejas clave/valor que contiene el diccionario.
3. Las parejas clave/valor no aparecen en un orden determinado. Las claves comúnmente se definen como textos para que sea fácil recordar lo que estamos almacenando allí. Pero las claves también se podrían definir como números.

Repasando las propiedades de los diccionarios

Analicemos el siguiente código.

Analizar

```
artista = {"nombre":"Fernando", "apellido":"Botero", "edad":90}
```

Basado en la variable (diccionario) *artista*, responda: (i) ¿Cuál es el tamaño de *artista*?, (ii) ¿Qué valor se encuentra en la clave *"nombre"*?, (iii) ¿Cuántas claves se definen en *artista*? Y (iv) ¿De qué tipo es el valor que se almacena en la clave *"edad"*?

Respuestas: (i) *3*, (ii) *"Fernando"*, (iii) *3* y (iv) *int*.

12.3. Visualización de diccionarios en memoria

Retomemos la representación gráfica de cajas que utilizamos en capítulos anteriores, para representar la variable que se define en el siguiente código.

Codificar

```
1.  misDatos = {"nombre":"Luisa", "edad":18, "estatura":1.67}
```

La Figura 12-3 muestra la representación gráfica de la variable *misDatos* previamente definida. Analicemos varios elementos de esta figura.

- El papel que contiene *misDatos* almacena una referencia con un número (un identificador). Esto se debe a que las variables tipo *dict* almacenan **referencias**.
- En la clave *"nombre"* se almacena el texto *"Luisa"*.
- En la clave *"edad"* se almacena el número entero *18*.
- En la clave *"estatura"* se almacena el número flotante *1.67*.

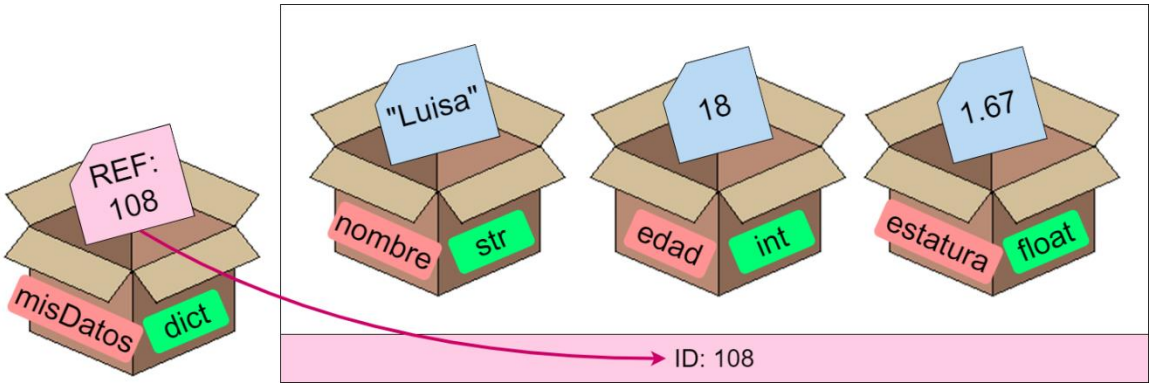


Figura 12-3. Representación gráfica de la variable *misDatos*.

¿Cuándo utilizar una lista y cuándo un diccionario?

Recomendamos utilizar **una lista** para almacenar datos de manera ordenada, cuando estos datos son del mismo tipo, y cualquier dato en cualquier posición tiene el mismo significado. Por ejemplo: edades, días de la semana, calificaciones, entre otros. El siguiente código muestra tres ejemplos de listas. Si tomamos la lista definida en la línea 2, nos percatamos que todos los elementos son tipo *str*, y que si analizamos los valores en cualquier posición todos tienen el mismo significado (días de la semana).

Codificar

```
1. notas = [4.5, 3.0, 5.0]
2. dias = ["Lunes", "Martes", "Miércoles"]
3. edades = [40, 12, 29, 55]
```

Recomendamos utilizar **un diccionario** para almacenar datos en los que el orden no importa, y, además, quieres que estos datos queden asociados a claves para identificarlos fácilmente. En este caso, los datos pueden ser de tipos diferentes y pueden tener diferentes significados. Por ejemplo: los datos de una persona, de un vehículo, o de un personaje de un videojuego, entre otros. El siguiente código muestra tres ejemplos de diccionarios. Si tomamos el diccionario definido en la línea 2, nos damos cuenta de que contiene un valor *str* y un valor *int*, y que ambos valores representan cosas diferentes (el nombre y el ataque del personaje).

Codificar

```
1. computador = {"marca":"Dell", "procesador":"i7"}
2. personaje = {"nombre":"mago", "ataque":20}
3. pais = {"nombre":"Colombia", "capital":"Bogotá"}
```

12.4. Operaciones básicas

A continuación, analizaremos algunas de las operaciones básicas cuando trabajamos con diccionarios: (i) acceder a valores de un diccionario, (ii) modificar valores de un diccionario, (iii) agregar valores a un diccionario, y (iv) eliminar valores de un diccionario.

Para ejemplificar estas operaciones, nos imaginaremos que estamos en un videojuego donde personalizamos un carro.

Acceder a valores de un diccionario

Para acceder a un valor de un diccionario debemos seguir el procedimiento descrito a continuación. Primero colocamos el **nombre de la variable** donde se almacenará el diccionario, luego abrimos y cerramos corchetes "**[]**", y en el medio de los corchetes colocamos la **clave** a la cual está asociada el valor al que queremos acceder. Este procedimiento nos retornará el valor al cual accedimos.

El siguiente código muestra cómo acceder a diferentes valores de un diccionario (ver ejecución en Figura 12-4).

- En las líneas 1 a 4 creamos un diccionario con dos parejas clave/valor y la asignamos a *carro*.
- En la línea 6 imprimimos el tamaño del diccionario *carro* (o sea *2*).
- En la línea 7 accedemos e imprimimos el valor almacenado en la clave *"marca"* de *carro* (o sea *"Toyota"*).
- En la línea 8 accedemos e imprimimos el valor almacenado en la clave *"color"* de *carro* (o sea *"negro"*).

Codificar y ejecutar

```
1. carro = {
2.   "marca":"Toyota",
3.   "color":"negro"
4. }
5.
6. print(len(carro))
7. print(carro["marca"])
8. print(carro["color"])
```

```
2
Toyota
negro
```

Figura 12-4. Ejecución del código anterior.

Modificar valores de un diccionario

Para modificar un valor de un diccionario debemos seguir el procedimiento descrito a continuación. Primero colocamos el **nombre de la variable** donde se almacena el diccionario. Luego abrimos y cerramos corchetes "**[]**". En el medio de los corchetes colocamos la **clave** a la cual está asociada el valor que queremos modificar. Luego colocamos un igual "**=**". Y finalmente colocamos el **nuevo valor** que queremos asignar a esa clave.

El siguiente código muestra cómo modificar un valor de un diccionario (ver ejecución en Figura 12-5).

- En las líneas 1 a 4 creamos un diccionario con dos parejas clave/valor y la asignamos a *carro*.
- En la línea 6 accedemos e imprimimos el valor almacenado en la clave *"color"* de *carro* (o sea *"negro"*).

- En la línea 7 modificamos el valor almacenado en la clave *"color"* de la variable *carro*, ahora su valor será *"rojo"*.
- En la línea 8 accedemos e imprimimos nuevamente el valor almacenado en la clave *"color"* de *carro* (esta vez es *"rojo"*).

Codificar y ejecutar

```
1. carro = {
2.   "marca":"Toyota",
3.   "color":"negro"
4. }
5.
6. print(carro["color"])
7. carro["color"] = "rojo"
8. print(carro["color"])
```

```
negro
rojo
```

Figura 12-5. Ejecución del código anterior.

Agregar valores a un diccionario

Para agregar un valor a un diccionario debemos seguir el procedimiento descrito a continuación. Primero colocamos el **nombre de la variable** donde se almacena el diccionario. Luego abrimos y cerramos corchetes "**[]**". En el medio de los corchetes colocamos la nueva **clave** a la cual estará asociada el nuevo valor que queremos agregar. Luego colocamos un igual "**=**". Y finalmente colocamos el **nuevo valor** que queremos agregar.

El siguiente código muestra cómo agregar un par de claves y valores a un diccionario (ver ejecución en Figura 12-6).

- En las líneas 1 a 4 creamos un diccionario con dos parejas clave/valor y lo asignamos a *carro*.
- En la línea 6 agregamos a *carro* la nueva clave *"maletero"* con valor *"SkyBox 16"*.
- En la línea 7 agregamos a *carro* la nueva clave *"guardabarro"* con valor *"Pocket Style"*.
- En la línea 8 accedemos e imprimimos el valor almacenado en la clave *"maletero"* de *carro* (o sea *"SkyBox 16"*).
- En la línea 9 imprimimos el diccionario *carro* completo.

Codificar y ejecutar

```
1.  carro = {
2.    "marca":"Toyota",
3.    "color":"negro"
4.  }
5.
6.  carro["maletero"] = "SkyBox 16"
7.  carro["guardabarro"] = "Pocket Style"
8.  print(carro["maletero"])
9.  print(carro)
```

```
SkyBox 16
{'marca': 'Toyota', 'color': 'negro', 'maletero': 'SkyBox 16', 'guardabarro': 'Pocket Style'}
```

Figura 12-6. Ejecución del código anterior.

Eliminar valores de un diccionario

Para eliminar un valor de un diccionario debemos utilizar el método *pop* y seguir el procedimiento descrito a continuación. Primero colocamos el **nombre de la variable** donde se almacena el diccionario, luego colocamos un punto "**.**", luego colocamos ***pop()***, y dentro de los paréntesis colocamos la **clave** a eliminar. Este procedimiento eliminará tanto la clave como el valor asociado a esa clave.

El siguiente código muestra cómo eliminar un elemento de un diccionario (ver ejecución en Figura 12-7).

- En las líneas 1 a 5 creamos un diccionario con tres parejas clave/valor y lo asignamos a *carro*.
- En la línea 7 imprimimos el diccionario *carro* que incluirá las tres parejas clave/valor.
- En la línea 8 eliminamos del diccionario *carro* la clave *"maletero"* la cual estaba asociada al valor *"SkyBox 16"*.
- En la línea 9 imprimimos el diccionario *carro* que incluirá esta vez dos parejas clave/valor.

Codificar y ejecutar

```
1.  carro = {
2.    "marca":"Toyota",
3.    "color":"negro",
4.    "maletero":"SkyBox 16"
5.  }
6.
7.  print(carro)
8.  carro.pop("maletero")
9.  print(carro)
```

```
{'marca': 'Toyota', 'color': 'negro', 'maletero': 'SkyBox 16'}
{'marca': 'Toyota', 'color': 'negro'}
```

Figura 12-7. Ejecución del código anterior.

TIP: Existen otros métodos disponibles para trabajar con diccionarios. Por lo tanto, le recomendamos dar un vistazo rápido a este enlace: https://www.w3schools.com/python/python_ref_dictionary.asp. Si lo prefiere, puede traducir la página a español.

12.5. Ciclos con listas que contienen diccionarios

En las aplicaciones que utilizamos a diario, resulta común que los programadores creen listas que contienen datos más complejos. Por ejemplo, una lista que contiene un conjunto de productos o usuarios representados como diccionarios. Si accede a su tienda online favorita, es posible que esos productos que esté viendo, estén internamente contenidos en una lista, y sean recorridos para ir mostrándolos uno a uno. A continuación, vamos a explicar cómo definir listas que contienen diccionarios, y luego como recorrerlas.

Listas que contienen diccionarios

El siguiente código muestra cómo definir una lista compuesta por diccionarios (ver ejecución en Figura 12-8).

- En las líneas 1 y 2 creamos dos diccionarios con la información de dos productos.
- En la línea 4 creamos una lista vacía llamada *listaProductos*.
- En las líneas 5 y 6 añadimos los dos diccionarios a *listaProductos*. El primer diccionario quedará asignado a la posición 0, y el segundo a la posición 1.

- En la línea 8 imprimimos la lista *listaProductos* que mostrará la lista con los dos elementos tipo diccionarios que contiene.
- En la línea 9 imprimimos el primer elemento (posición *0*) de la lista *listaProductos*. En este caso mostrará los datos asociados al primer producto.
- En la línea 10 imprimimos el valor almacenado en la clave *"nombre"* del primer elemento (posición *0*) de la lista *listaProductos*. En este caso mostrará *"Monitor"*.

Codificar y ejecutar

```
1.  producto1 = {"nombre":"Monitor", "precio":119}
2.  producto2 = {"nombre":"Mouse", "precio":6}
3.
4.  listaProductos = []
5.  listaProductos.append(producto1)
6.  listaProductos.append(producto2)
7.
8.  print(listaProductos)
9.  print(listaProductos[0])
10. print(listaProductos[0]["nombre"])
```

```
[{'nombre': 'Monitor', 'precio': 119}, {'nombre': 'Mouse', 'precio': 6}]
{'nombre': 'Monitor', 'precio': 119}
Monitor
```

Figura 12-8. Ejecución del código anterior.

Las Figuras 12-9 y 12-10 muestran la representación gráfica en memoria de los diccionarios y la lista creados en el código anterior. La Figura 12-9 muestra la representación gráfica en memoria de las líneas 1 y 2. En este caso se crean dos variables tipo diccionarios (*producto1* y *producto2*) y se asocian a ubicaciones en memoria que contienen los datos de los dos diccionarios.

La Figura 12-10 muestra la representación gráfica en memoria del código completo.

1. En la línea 5 se añade el *producto1* a la posición 0 de *listaProductos*. Aquí, lo que realmente sucede, es que se duplica el papel de *producto1* (la referencia), y se coloca en la caja con posición *0* de *listaProductos*.
2. El mismo procedimiento ocurre en la línea 6, pero aplicado a *producto2* y a la posición *1* de *listaProductos*.

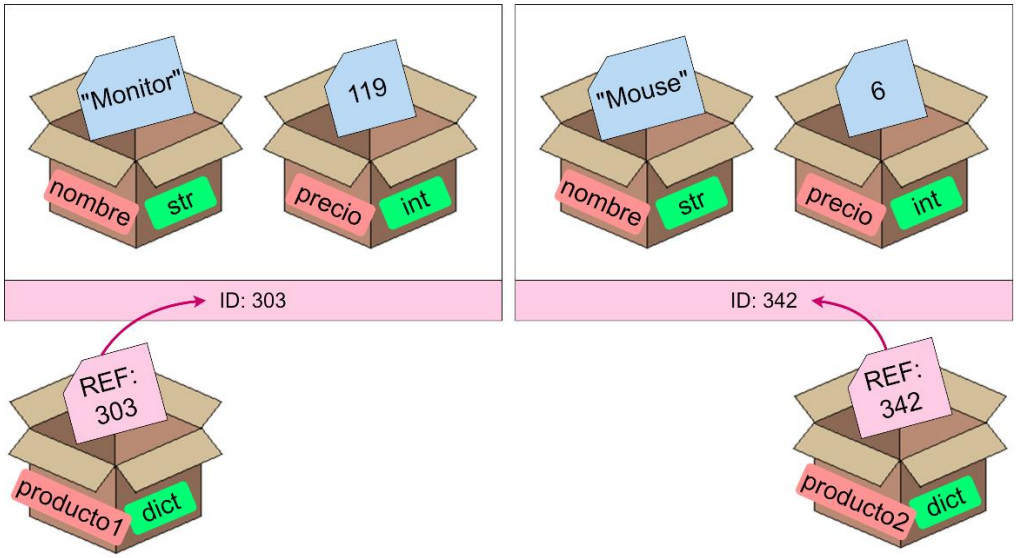


Figura 12-9. Representación gráfica de las variables *producto1* y *producto2* (líneas 1 y 2).

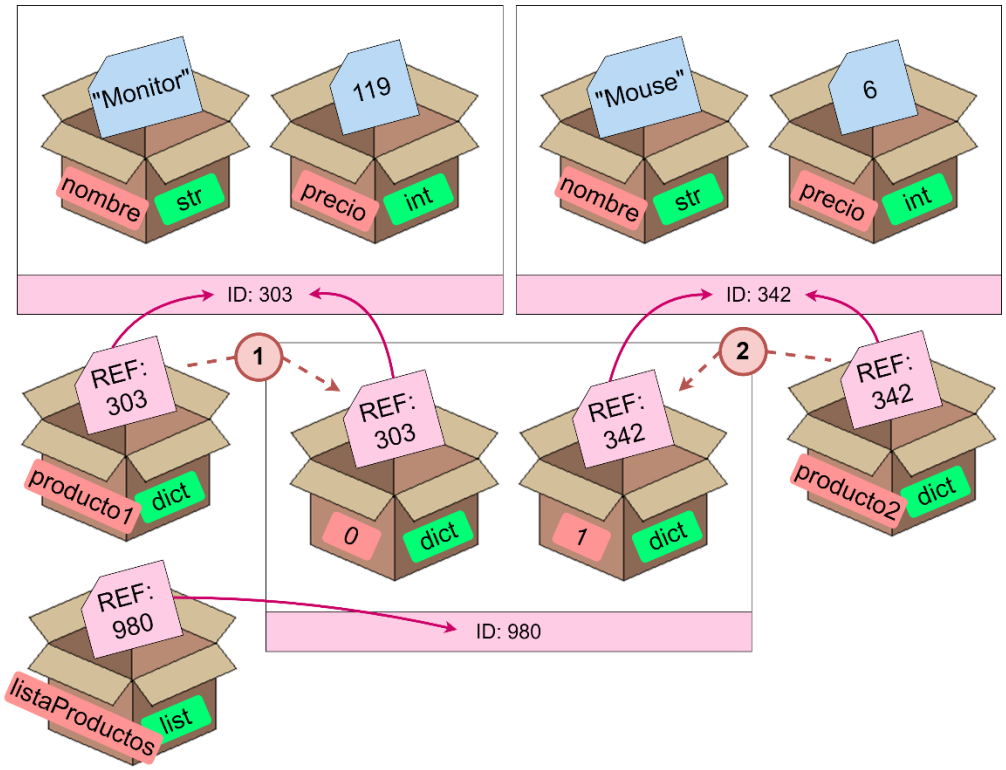


Figura 12-10. Representación gráfica de todas las variables del código anterior.

Basado en el código y la gráfica anterior responda: (i) ¿Cuál es el tamaño de *listaProductos*?, (ii) ¿Qué valor se encuentra en *listaProductos[1]["precio"]*?, (iii) ¿Qué valor se encuentra en *producto2["precio"]*?, (iv) ¿De qué tipo es la variable *listaProductos*?, (v) ¿De qué tipo es *listaProductos[0]*? Y (vi) ¿De qué tipo es el valor que se almacena en *listaProductos[1]["nombre"]*?

Respuestas: (i) *2*, (ii) *6*, (iii) *6*, (iv) *list*, (v) *dict* y (vi) *str*.

Ciclos con listas que contienen diccionarios

Una vez definidas nuestras listas que contienen diccionarios, podemos recorrerlas y mostrar su contenido. El siguiente código muestra cómo recorrer una lista compuesta por diccionarios (ver ejecución en Figura 12-11).

- En las líneas 1 y 2 creamos dos diccionarios con la información de dos productos.
- En la línea 4 creamos una lista llamada *listaProductos* y le asignamos los dos productos anteriores.
- En la línea 6 definimos la variable de control *i* inicializada en *0* (debido a que queremos pasar por todas las posiciones de *listaProductos*).
- En la línea 7 creamos el ciclo y definimos la condición. En este caso el ciclo se ejecutará siempre y cuando *i* sea menor que el tamaño de *listaProductos* (menor que *2*).
- En la línea 8 y 9 imprimimos el nombre y precio del producto sobre el cual estamos iterando. La primera vez *i* será igual a *0*, por lo tanto, accederemos a los datos de *listaProductos[0]* que representa el primer producto.
- En la línea 10 realizamos el cambio de la variable de control. A *i* le aumentaremos una unidad (dado que queremos ir pasando por todas las posiciones de *listaProductos*).

Codificar y ejecutar

```
1.  producto1 = {"nombre":"Monitor", "precio":119}
2.  producto2 = {"nombre":"Mouse", "precio":6}
3.
4.  listaProductos = [producto1, producto2]
5.
6.  i = 0
7.  while(i < len(listaProductos)):
8.    print("Nombre producto: "+listaProductos[i]["nombre"])
9.    print("Precio producto: "+str(listaProductos[i]["precio"]))
10.   i = i+1
```

```
Nombre producto: Monitor
Precio producto: 119
Nombre producto: Mouse
Precio producto: 6
```

Figura 12-11. Ejecución del código anterior.

12.6. Ejercicios

Ejercicios teóricos

E12.1. Diga qué imprime el siguiente programa.

Analizar

```
1. personaje = {"nombre":"Mago", "fuerza":30.5, "vidas":2}
2. print(personaje["nombre"])
3. print(type(personaje["fuerza"]))
```

E12.2. ¿Cuándo deberíamos utilizar un diccionario en lugar de una lista?

E12.3. Basado en la siguiente variable: *juguete = {"nombre": "Tren", "precio":99}*, responda: (i) ¿De qué tipo es la variable *juguete*?, (ii) ¿De qué tipo es el valor almacenado en *juguete["precio"]*?, (iii) ¿Cuántas claves posee *juguete*? Y (iv) ¿Qué valor se encuentra almacenado en *juguete["marca"]*?.

Ejercicios prácticos

E12.4. Implemente un programa en el cual le solicite al usuario que ingrese por teclado el nombre de una mascota, su edad y su raza. Luego cree un diccionario con tres claves: *"nombre"*, *"edad"* y *"raza"*, y asocie cada uno de los valores recibidos por teclado a esas claves. Finalmente imprima el diccionario.

Ejemplo de entada	Ejemplo de salida
Oreo *4* *Persa*	*{'nombre': 'Oreo', 'edad': 4, 'raza': 'Persa'}*

E12.5. Implemente un programa en Python en el que le solicite al usuario que ingrese por teclado un número entero. Ese número representará la cantidad de equipos de futbol de los que es fan. Luego cree una lista vacía llamada *equipos*. Luego cree un ciclo que se repita tantas veces como

el número ingresado (por ejemplo, si ingresa 3, se deberá crear un ciclo que se ejecute 3 veces). Ahora, dentro del cuerpo del ciclo solicite al usuario que ingrese por teclado el nombre de un equipo y su país de origen. Luego cree un diccionario con las claves *"nombre"* y *"pais"*, y asocie los valores recibidos por teclado a esas claves. Y antes de finalizar el cuerpo del ciclo, añada el diccionario a la lista *equipos*. Por último, por fuera del ciclo, imprima la lista *equipos*.

Ejemplo de entada	**Ejemplo de salida**
2 *Real Madrid* *España* *PSG* *Francia*	*[{'nombre': 'Real Madrid', 'pais': 'España'}, {'nombre': 'PSG', 'pais': 'Francia'}]*

Resumen

En este capítulo aprendimos a trabajar con diccionarios. Aprendimos las propiedades principales de los diccionarios: (i) que están compuestos por parejas clave/valor, (ii) que tienen un tamaño y (iii) que estas parejas clave/valor no aparecen en un orden determinado. Aprendimos cómo visualizar diccionarios en memoria, e incluso listas que contienen diccionarios. Aprendimos cuatro operaciones básicas cuando trabajamos con diccionarios: (i) acceder a un valor particular, (ii) modificar un valor particular, (iii) agregar una pareja clave/valor y (iv) eliminar una pareja clave/valor (utilizando el método *pop*). Y finalmente, aprendimos cómo usar ciclos para recorrer listas que contienen diccionarios.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una “X” los siguientes trucos: T06, T55, T56, T57, T58, T59, T60 y T61.

En el siguiente capítulo retomaremos el tema de ciclos y aprenderemos otra forma de recorrer diversos datos mediante el ciclo *for*.

Capítulo 13 – Ciclo for

Dado que ya analizamos cómo funcionan los textos, las listas, los diccionarios, y algunas de sus propiedades principales, ahora podemos dar paso a entender otro tipo de ciclo en Python, el ciclo *for*. El ciclo *for*, es una alternativa al ciclo *while*. Con este tipo de ciclo también podemos recorrer diversos tipos de variables, y en algunos casos el código se hace más compacto.

En este capítulo explicaremos cómo funciona el ciclo *for* y cómo utilizarlo para recorrer algunos de los tipos de variables ya vistos.

A continuación, desarrollaremos las siguientes secciones:

1. Estructura clásica de un ciclo for.
2. Uso de for para recorrer algunos tipos de variables.
3. Función range.
4. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo13. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

13.1. Estructura clásica de un ciclo for

Antes de presentar la estructura clásica de un ciclo *for* en Python. Comparemos las definiciones de los ciclos *while* y *for*.

Un ciclo ***while*** ejecuta un conjunto de instrucciones **mientras** una condición sea verdadera (*True*). Por su parte, un ciclo ***for*** ejecuta un conjunto de instrucciones **para** cada elemento de una colección (ya sea una lista, diccionario, texto, u otra). Por lo tanto, a un ciclo *for* comúnmente **se le debe especificar la colección a recorrer**.

Un ciclo *for* clásico posee una estructura como la presentada en la Figura 13-1.

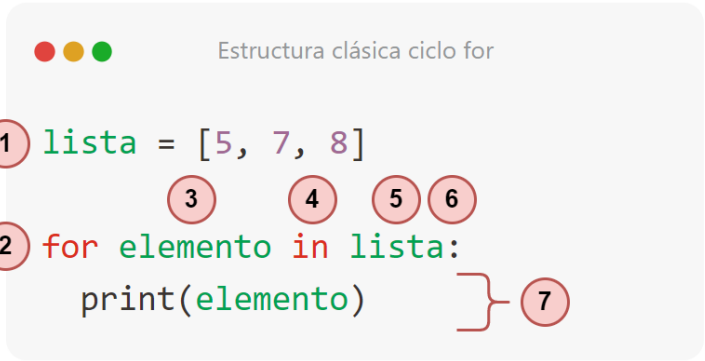


Figura 13-1. Estructura clásica de un ciclo *for*.

1. Se define la **colección** a recorrer (ya sea una lista, diccionario, texto, entre otros).
2. En la línea siguiente se utiliza la palabra reservada de Python ***for*** para empezar a definir el ciclo.
3. Se define el **nombre de la variable de control**. Esta variable va tomando uno a uno los valores almacenados en la colección. Cuando se itera sobre todos los valores de la colección, el ciclo finaliza.
4. Se coloca la palabra reservada de Python "**in**".
5. Se coloca la **variable** que representa la colección a recorrer.
6. Se agregan dos puntos "**:**".
7. Se define el **cuerpo del *for***. Allí se coloca la instrucción (o instrucciones) que se ejecutará mientras se itera por cada elemento de la colección. Esa instrucción (o instrucciones) se deben colocar como bloque de código indentado, es decir, mover de izquierda a derecha (con respecto al *for*) mediante una tabulación.

Programando nuestro primer ciclo *for*

Supongamos que tenemos una lista que contiene las edades de algunos familiares, y queremos recorrer e imprimir los valores almacenados en esa lista. El siguiente código muestra cómo utilizar un ciclo *for* para recorrer listas (ver ejecución en Figura 13-2).

- En la línea 1 definimos la colección a recorrer. En este caso es una lista de edades que contiene tres valores.
- En la línea 3 creamos el ciclo *for*. En este caso, nuestra variable de control la llamaremos *edad*. Esto debido a que, dentro de la lista *edades*, cada valor representa una edad. Además, definimos que *edad* iterará sobre los valores almacenados en la lista *edades* (eso es lo que significa la porción del código "*in edades*").
- La línea 4 contiene el cuerpo del *for*. En este caso, imprimimos el valor almacenado en la variable de control *edad* (la primera vez que se ejecute el ciclo *for*, *edad* contendrá el primer elemento almacenado en *edades*, o sea, a *25*).

- Luego el ciclo se volverá a ejecutar, y esta vez *edad* tomará el valor del segundo elemento almacenado en *edades* (o sea *7*) y lo imprimirá por pantalla.
- Luego el ciclo se volverá a ejecutar, y esta vez *edad* tomará el valor del tercer elemento almacenado en *edades* (o sea *39*) y lo imprimirá por pantalla.
- Dado que no quedan más elementos por recorrer en *edades*. El ciclo finalizará su ejecución y se imprimirá el mensaje de la línea 6 (la Figura 13-3 muestra el flujo de ejecución paso a paso).

Codificar y ejecutar

```
1. edades = [25, 7, 39]
2.
3. for edad in edades:
4.   print(edad)
5.
6. print("Fin programa")
```

```
25
7
39
Fin programa
```

Figura 13-2. Ejecución del código anterior.

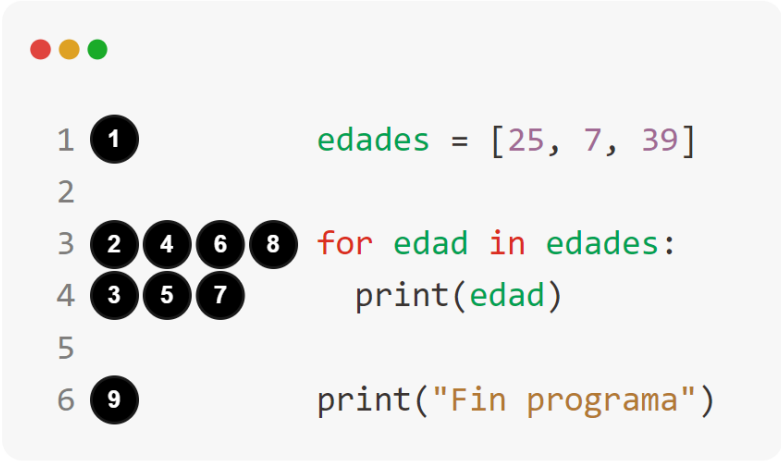


Figura 13-3. Flujo de ejecución paso a paso del código anterior.

13.2. Uso de for para recorrer algunos tipos de variables

En el código desarrollado en la sección anterior vimos cómo recorrer listas con el uso del ciclo *for*. Ahora veamos cómo recorrer textos, diccionarios, y listas que contienen diccionarios.

Ciclo *for* con textos

El siguiente código muestra cómo recorrer una variable *str* con el uso de un ciclo *for*. En la línea 1 definimos el texto a recorrer (almacenado en la variable *mensaje*). Luego en la línea 3 definimos el ciclo *for*. En este caso, la variable de control *caracter* recorrerá y tomará el valor que hay en cada posición del texto almacenado en *mensaje* (o sea cada carácter). Y finalmente en la línea 4 imprimimos cada carácter (ver ejecución en Figura 13-4).

Codificar y ejecutar

```
1. mensaje = "Hola Ana"
2.
3. for caracter in mensaje:
4.   print(caracter)
```

```
H
o
l
a

A
n
a
```

Figura 13-4. Ejecución del código anterior.

TIP: La variable de control del ciclo *for* se puede definir con cualquier nombre. Nosotros sugerimos que le asigne un nombre que represente lo que realmente se almacena en cada posición de la colección a recorrer. Por ejemplo, si está iterando sobre una lista de salarios, llame a la variable de control *salario*. O si está iterando sobre una lista de estudiantes, llame a la variable de control *estudiante*.

Ciclo *for* con diccionarios

El siguiente código muestra cómo recorrer una variable *dict* con el uso de un ciclo *for*. En la línea 1 definimos el diccionario a recorrer (almacenado en la variable *ciclista*). Luego, en la línea 3

definimos el ciclo *for*. En este caso, la variable de control *clave* tomará cada clave que existe en el diccionario *ciclista* (así funciona el ciclo *for* con *dict*, por defecto toma las claves del diccionario). Y finalmente, en la línea 4 imprimimos cada clave, y el valor almacenado en cada clave del diccionario *ciclista* (ver ejecución en Figura 13-5).

Codificar y ejecutar

```
1. ciclista = {"nombre":"Egan", "apellido":"Bernal", "tipo":"Escalador"}
2.
3. for clave in ciclista:
4.   print(clave+": "+ciclista[clave])
```

```
nombre: Egan
apellido: Bernal
tipo: Escalador
```

Figura 13-5. Ejecución del código anterior.

Ciclo *for* con listas que contienen diccionarios

El siguiente código muestra cómo usar un ciclo *for* para recorrer una variable *list* que contiene diccionarios. En las líneas 1 y 2 definimos un par de diccionarios (con información de discotecas). Luego, en la línea 4 definimos una lista llamada *discotecas* y le asignamos las dos discotecas definidas anteriormente. Luego en la línea 6 definimos el ciclo *for*. En este caso, la variable de control *discoteca* tomará el valor almacenado en cada posición de la lista *discotecas* (tomará cada diccionario discoteca). Y finalmente, en la línea 7 imprimimos el *nombre* y *ubicación* de cada discoteca que está siendo iterada (ver ejecución en Figura 13-6).

Codificar y ejecutar

```
1. discoteca1 = {"nombre":"Perro negro", "ubicación":"Poblado"}
2. discoteca2 = {"nombre":"Teatro victoria", "ubicación":"Poblado"}
3.
4. discotecas = [discoteca1, discoteca2]
5.
6. for discoteca in discotecas:
7.   print(discoteca["nombre"]+" - "+discoteca["ubicación"])
```

```
Perro negro - Poblado
Teatro victoria - Poblado
```

Figura 13-6. Ejecución del código anterior.

13.3. Función range

En las secciones anteriores vimos cómo recorrer diferentes tipos de variables y valores. Pero, ¿qué tal si queremos utilizar el ciclo *for* para imprimir números en un orden particular?

Por ejemplo, el siguiente código muestra cómo utilizar el ciclo *for* para imprimir los números del 1 al 5. En este caso, debemos crear una lista que contenga los números del 1 al 5 y luego recorrerla e imprimir sus valores almacenados (ver ejecución en Figura 13-7).

Codificar y ejecutar

```
1. numeros = [1, 2, 3, 4, 5]
2.
3. for numero in numeros:
4.   print(numero)
```

```
1
2
3
4
5
```

Figura 13-7. Ejecución del código anterior.

Y ahora, ¿qué tal si queremos imprimir los números del 1 al 100? Si seguimos la lógica del ejemplo anterior, deberíamos crear una lista que contenga los números del 1 al 100. Sin embargo, el código quedaría muy extenso. Afortunadamente, Python provee una función llamada ***range***, que permite crear secuencias de números.

Función *range* con un solo argumento

En su forma más simple, la función *range* recibe un solo argumento (un número entero). Supongamos que codificamos ***range(n)***, *range* devolverá una secuencia de números según las siguientes características:

- Por defecto, la secuencia de números iniciará desde el número 0.
- Por defecto, la secuencia de números irá incrementando de a una unidad.

- La secuencia de números terminará antes de llegar al número *n* recibido como argumento (es decir, *n* no se incluye).

El siguiente código muestra cómo utilizar la función *range* (con un argumento) para crear una secuencia de números e iterar sobre esta. En la línea 1 definimos la variable de control *numero*, esta variable iterará sobre la secuencia creada por *range(6)*. *range(6)* creará una secuencia de números iniciando en el número *0*, incrementando de a *1*, y llegando hasta una unidad antes del número *6* (o sea hasta *5*). La Figura 13-8 muestra la ejecución por pantalla.

Codificar y ejecutar

```
1. for numero in range(6):
2.     print(numero)
```

```
0
1
2
3
4
5
```

Figura 13-8. Ejecución del código anterior.

Pero ¿qué tal si queremos iniciar desde un número particular, o incrementar de a dos unidades? Veamos cómo funciona la función *range* con múltiples argumentos.

Función *range* con múltiples argumentos

La función range puede recibir hasta tres argumentos. Supongamos que codificamos ***range(inicio, parada, paso)***, *range* devolverá una secuencia de números basados en *inicio*, *parada* y *paso*, y en las siguientes características:

- **inicio (opcional):** indica el número entero desde el cual inicia la secuencia. Si no se coloca nada iniciará desde 0.
- **parada (requerido):** indica el número entero en el cual la secuencia termina (ese número no se incluye en la secuencia generada).
- **paso (opcional):** indica el número entero que define el incremento o decremento de la secuencia. Si no se coloca nada definirá un incremento de 1 unidad.

El siguiente código muestra cómo utilizar la función *range* (con múltiples argumentos) para crear una secuencia de números e iterar sobre esta. En la línea 1 definimos la variable de control *numero*, esta variable iterará sobre la secuencia creada por *range(1, 7, 2)*. La instrucción *range(1,*

7, 2) creará una secuencia de números iniciando en el número *1*, incrementando de a *2* unidades, y llegando hasta antes del número *7* (o sea hasta *5*). La Figura 13-9 muestra la ejecución por pantalla.

Codificar y ejecutar

```
1. for numero in range(1, 7, 2):
2.   print(numero)
```

```
1
3
5
```

Figura 13-9. Ejecución del código anterior.

Basado en el funcionamiento de *range*, responda: (i) ¿Qué secuencia genera *range(0, 11, 3)*?, (ii) ¿Qué secuencia genera *range(5, 21, 5)*?, (iii) ¿Qué secuencia genera *range(10, 0, -1)*? Y (iv) ¿Cómo debo definir la función *range* para que genere los siguientes números: *2,4,6,8*.

Respuestas: (i) *0,3,6,9*, (ii) *5,10,15,20,* (iii) *10,9,8,7,6,5,4,3,2,1* y (iv) *range(2, 9, 2)*.

13.4. Ejercicios

Ejercicios teóricos

E13.1. ¿Qué imprime el siguiente programa si se ingresa *4* por teclado?

Analizar

```
1. numero = int(input())
2. for i in range(numero):
3.   print("Rocas")
```

E13.2. ¿Cuáles son los tres argumentos que puede recibir la función *range*?

E13.3. Analice el siguiente código y mencione cuáles líneas de código se ejecutan.

Analizar

```
1. mercado = ["Pan", "Harina", "Huevos"]
2. for articulo in mercado:
3.     print(articulo)
```

Ejercicios prácticos

E13.4. Implemente lo siguiente: (i) un ciclo *for* que imprima los números de uno en uno, desde el 1 hasta el 80. (ii) un ciclo *for* que imprima los números del 1000 al 0 (decrementando de a 100). Y (iii) un ciclo *for* que imprima los números pares del 2 al 66.

E13.5. Una mujer está buscando un nombre para su hijo. Una amiga le envía una lista de opciones con los siguientes nombres: *"Luis"*, *"David"*, *"Leon"* y *"Martin"*. Sin embargo, la mujer ha tenido malas experiencias con hombres cuyos nombres empiezan por *"L"*. Implemente un programa en el que reciba los cuatro nombres por teclado. Luego, añada los cuatro nombres a una lista, y luego recorra la lista de nombres con un ciclo *for*. Dentro del cuerpo de ciclo, usted deberá verificar si el nombre sobre el que se está iterando inicia por la letra *"L"*. En caso afirmativo, debe imprimir un mensaje diciendo *"Nombre descartado"*, y en caso contrario, un mensaje diciendo *"Nombre posible"*. **Pista:** recuerde cómo extraer un carácter de un texto.

Ejemplo de entada	**Ejemplo de salida**
Luis	*Nombre descartado*
David	*Nombre posible*
Leon	*Nombre descartado*
Martin	*Nombre posible*

Resumen

En este capítulo aprendimos a trabajar con el ciclo *for*. Aprendimos la estructura clásica de un ciclo *for*. Aprendimos a usar el ciclo *for* para recorrer textos, listas, diccionarios y listas que contienen diccionarios. Y aprendimos a utilizar la función *range* para generar secuencias de números que podemos utilizar en el ciclo *for*.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una “X” los siguientes trucos: T62, T63, T64, T65, T66 y T67.

En el siguiente capítulo aprenderemos las propiedades de las funciones en Python y cómo definir nuestras propias funciones.

Capítulo 14 – Funciones

En programación, las funciones ayudan a dividir mejor el código de nuestros programas. A medida que nuestro programa va creciendo se van creando decenas, cientos o incluso miles de líneas de código. Con el uso de funciones, podemos organizar mejor nuestros programas y hacer que diferentes piezas de código sean reutilizables y más manejables.

En este capítulo explicaremos cómo funcionan las funciones en Python y cómo crear nuestras propias funciones.

A continuación, desarrollaremos las siguientes secciones:

1. Valores y referencias.
2. Definición de funciones.
3. Funciones con parámetros.
4. Ámbito de las variables.
5. Funciones con retorno.
6. Calculadora.
7. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo14. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

14.1. Valores y referencias

Antes de presentar la temática de funciones, hagamos un repaso a la forma simplificada que usamos para imaginarnos cómo se manejan los valores y referencias en la memoria de la computadora. Esto luego nos servirá para entender mejor qué sucede cuando le pasamos valores o variables a las funciones en Python.

Valores

Como vimos en capítulos pasados, en este libro imaginamos que las variables de tipo *int*, *str*, *float*, y *bool* almacenan valores. Esto quiere decir, que cuando a una variable, le asignamos otra variable (de los tipos anteriores), se **copiará el valor** de la variable de la derecha en la variable de la

izquierda. Y las variables quedarán "**independientes**" entre ellas. Cualquier cambio que se haga a alguna de esas variables no modificará la otra variable.

El siguiente código muestra ese funcionamiento en acción (ver ejecución en Figura 14-1).

- En la línea 1 definimos la variable *precio1* con un valor *int* de *100*.
- En la línea 2 definimos la variable *precio2* a la cual le asignamos la variable *precio1*. En este punto, **se copiará el valor** de *precio1* en *precio2* (ver Figura 14-2 - Línea 2).
- Luego, en la línea 3 imprimimos *precio1*, y según la "Figura 14-2 - Línea 2" en ese punto *precio1* tiene un valor de *100*.
- Luego, en la línea 4 modificamos el valor de *precio2* y le asignamos *200*. Este cambio solo afecta a *precio2*.
- Finalmente, en la línea 5 imprimimos *precio1*, y según la "Figura 14-2 - Línea 4", *precio1* continúa con un valor de *100* (ya que las variables quedaron **independientes** entre ellas).

Codificar y ejecutar

```
1. precio1 = 100
2. precio2 = precio1
3. print(precio1)
4. precio2 = 200
5. print(precio1)
```

```
100
100
```

Figura 14-1. Ejecución del código anterior.

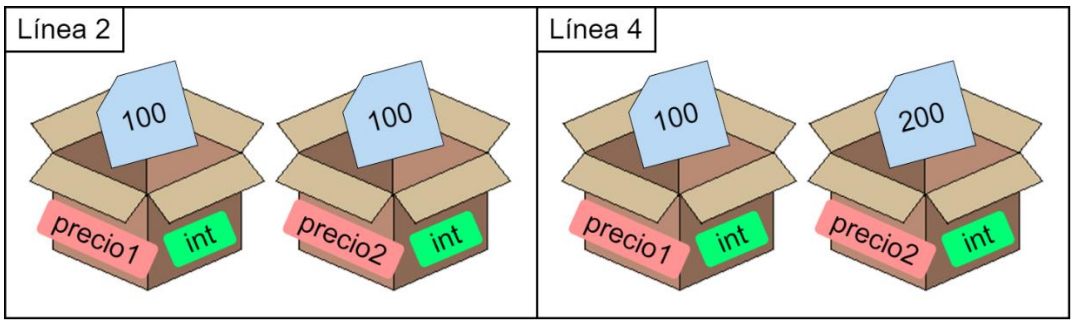


Figura 14-2. Representación gráfica de las variables *precio1* y *precio2*.

Referencias

Como vimos en capítulos pasados, las variables de tipo *list* y *dict* almacenan referencias. Esto quiere decir, que cuando a una variable, le asignamos otra variable (de los tipos anteriores), se **copiará la referencia** de la variable de la derecha en la variable de la izquierda. Y las variables quedarán "**dependientes**" entre ellas. Cualquier cambio que se haga a alguna de esas variables modificará la otra variable (ya que ambas apuntan a la misma referencia).

El siguiente código muestra ese funcionamiento en acción (ver ejecución en Figura 14-3).

- En la línea 1 definimos la variable tipo lista *motos1* con el elemento *"Yamaha"*.
- En la línea 2 definimos la variable *motos2*, a la cual le asignamos la variable *motos1*. En este punto, **se copiará la referencia** almacenada dentro de *motos1* en *motos2* (ver Figura 14-4 - Línea 2).
- Luego en la línea 3 imprimimos *motos1* en la posición *0*, y según la "Figura 14-4 - Línea 2", *motos1[0]* contiene el valor *"Yamaha"*.
- Luego, en la línea 4 modificamos a *motos2[0]* y le asignamos *"Ducati"*. Este cambio afectará tanto a *motos2* como a *motos1* (dado que ambas variables apuntan a la misma referencia).
- Finalmente, en la línea 5 imprimimos *motos1* en la posición *0*, y según la "Figura 14-4 - Línea 4", *motos1[0]* contiene el valor *"Ducati"* (ya que las variables quedaron **dependientes** entre ellas).

Codificar y ejecutar

```
1. motos1 = ["Yamaha"]
2. motos2 = motos1
3. print(motos1[0])
4. motos2[0] = "Ducati"
5. print(motos1[0])
```

```
Yamaha
Ducati
```

Figura 14-3. Ejecución del código anterior.

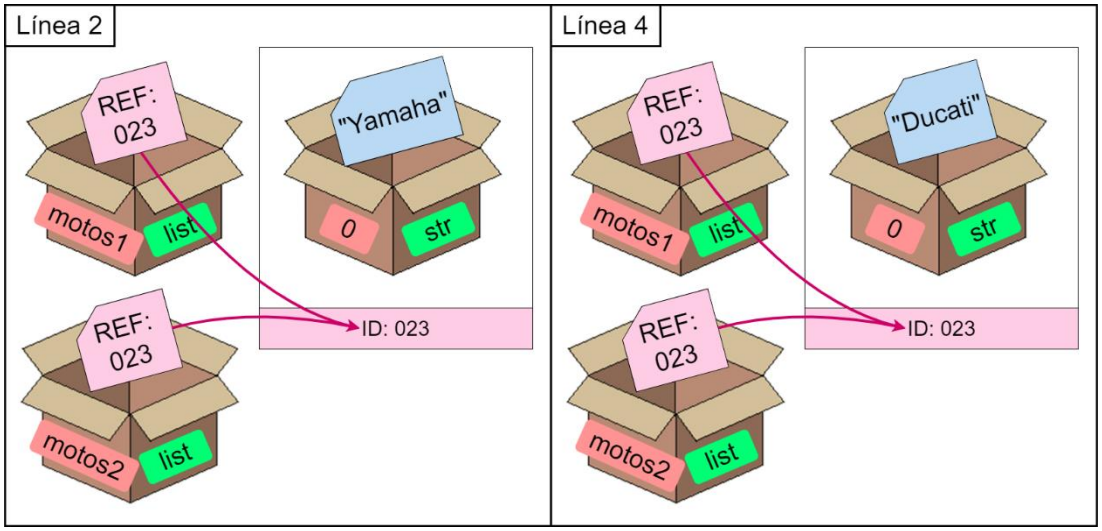


Figura 14-4. Representación gráfica de las variables *motos1* y *motos2*.

14.2. Definición de funciones

En este punto del libro, esperamos que ya esté familiarizado con algunas funciones que provee Python, tales como: *print*, *input* y *len*.

Una **función** es un bloque de código que está diseñado para hacer una tarea específica. Por ejemplo, la función *input* está diseñada para recoger datos que ingresa el usuario por teclado. La función *print* está diseñada para imprimir información por pantalla. Una función también puede verse como un "mini programa" que puede usarse dentro de un programa.

El propósito de usar funciones es dividir un programa grande en pequeñas partes. Y de esta manera hacer que el código sea más modular y reutilizable.

En Python podemos definir nuestras propias funciones, para ello veamos la estructura de una función.

Estructura de una función

Una función posee una estructura como la presentada en la Figura 14-5.

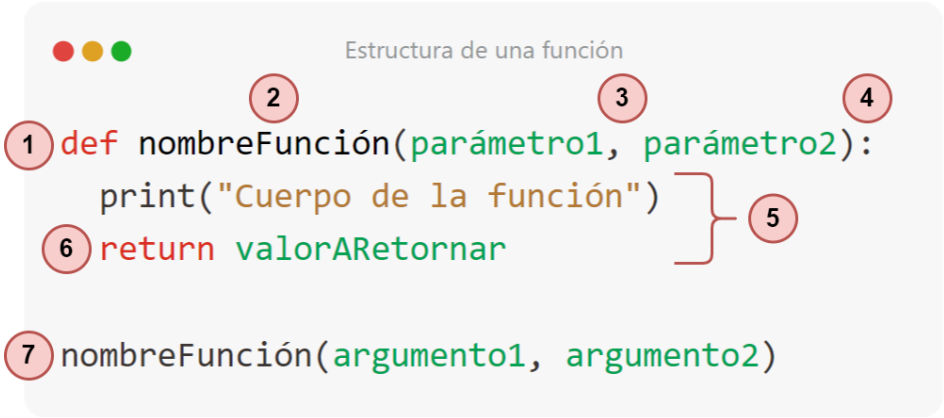


Figura 14-5. Estructura de una función.

1. Se utiliza la palabra reservada de Python ***def*** (que indica que vamos a definir una función).
2. Se define el **nombre de la función**. Se recomienda colocar un nombre alusivo a la acción específica que realiza la función (recuerde no utilizar como nombre de la función, nombres de palabras reservadas de Python).
3. Se agregan paréntesis "**()**" y dentro de los paréntesis se definen los **parámetros** que recibe la función. Los parámetros son opcionales. En la siguiente sección se explicará cómo se comportan las funciones con parámetros.
4. Se colocan dos puntos "**:**".
5. Se define el **cuerpo de la función**. Allí se coloca la instrucción (o instrucciones) propias de la función. Esa instrucción (o instrucciones) se deben colocar como bloques de código indentados. Es decir, mover de izquierda a derecha (con respecto a *def*) mediante una tabulación.
6. Si la función tiene retorno, se debe colocar la palabra reservada de Python ***return***, seguida del **valor a retornar**. El retorno es opcional y se explicará en una sección posterior.
7. Finalmente, cuando se desee ejecutar el código de la función, esta se debe llamar o "invocar". Para invocar una función se debe colocar, por fuera del cuerpo de la función, el **nombre de la función**, seguida de **paréntesis** y se deben enviar tantos **argumentos** como parámetros tenga la función.

Definiendo nuestra primera función

Teniendo en cuenta la estructura anterior es hora de definir nuestra primera función. Inicialmente, definiremos una función simple sin parámetros ni retorno. El siguiente código muestra cómo definir una función llamada *saludar*, que se encarga de imprimir por pantalla un texto de saludo (ver ejecución en Figura 14-6).

- En la línea 1 definimos la función *saludar* que no recibe parámetros.
- En las líneas 2 y 3 definimos el cuerpo de la función. En este caso la función se encarga de imprimir dos textos por pantalla.
- Luego, en la línea 5 invocamos la función, esto implica que se ejecutará el cuerpo de la función y se imprimirán los dos textos por pantalla.
- Al terminar la ejecución de la función, se ejecuta la línea 6 y se imprime por pantalla el texto *"Fin programa"*.

Codificar y ejecutar

```
1. def saludar():
2.   print("Bienvenido")
3.   print("A mi programa")
4.
5. saludar()
6. print("Fin programa")
```

```
Bienvenido
A mi programa
Fin programa
```

Figura 14-6. Ejecución del código anterior.

Flujo de ejecución

El flujo de ejecución de un programa que contiene la definición de una función propia es un poco diferente a los flujos de ejecución que hemos visto en este libro. Por lo tanto, hagamos seguimiento, paso a paso, de las líneas de código (del código anterior) que se ejecutan y en qué orden lo hacen (ver Figura 14-7):

- **Paso 1:** Se ejecuta la línea 5. Allí invocamos la función *saludar*. Las líneas 1 a 3 son omitidas por Python (dado que allí se encuentra el código de la función *saludar*). Esto quiere decir que **el código que define una función no se ejecuta por defecto**.
- **Paso 2:** Luego de la invocación anterior, se ejecuta la línea 1.
- **Pasos 3 y 4:** Se ejecuta el cuerpo de la función *saludar* y se imprimen los dos textos allí definidos.
- **Paso 5:** El cuerpo de la función termina, y ahora el programa retorna a su ejecución principal (a la línea 6, línea posterior a la invocación de la función). Allí se imprime por pantalla el texto *"Fin programa"*.

Figura 14-7. Flujo de ejecución paso a paso del código anterior.

Ejercicio

Basado en el siguiente código, responda: (i) ¿Qué líneas y en qué orden se ejecutan? Y (ii) ¿Cuál será la salida por pantalla?

Codificar y ejecutar

```
1. def saludar():
2.   print("Bienvenido")
3.   print("A mi programa")
4.
5. saludar()
6. saludar()
```

Respuestas: (i) se ejecutan las líneas *5,1,2,3,6,1,2,3*, y (ii) *Bienvenido A mi programa Bienvenido A mi programa*.

14.3. Funciones con parámetros

Algunas funciones requieren que se les pasen argumentos o información para poder funcionar. Por ejemplo, la función *range* requiere que le enviemos un argumento (un *int* que indica el número hasta el cual se creará una secuencia). Sin embargo, la función *saludar* de la sección anterior no requería el envío de argumentos.

En esta sección aprenderemos a definir **funciones con parámetros**, o sea, funciones que reciben argumentos cuando son invocadas.

El siguiente código muestra cómo definir una función con parámetros (ver ejecución en Figura 14-8).

- En la línea 1 definimos la función *saludoPersonalizado* que recibe dos parámetros (*nombre* y *apellido*).
- En la línea 2 definimos el cuerpo de la función. En este caso la función se encarga de imprimir un mensaje con el texto *"Hola"* concatenado con los valores que se almacenarán en los dos parámetros que recibe la función (en *nombre* y *apellido*).
- Luego, en la línea 4 invocamos la función, y enviamos dos argumentos. El primer argumento es el valor *"Juliana"* que se asignará al parámetro *nombre*. Y el segundo argumento es el valor *"Correa"* que se asignará al parámetro *apellido*. Luego se ejecutará el cuerpo de la función imprimiendo el saludo personalizado.
- Luego, en la línea 5 volvemos a invocar la función, y enviamos dos argumentos. El primer argumento es el valor *"Gonzalo"* que se asignará al parámetro *nombre*. Y el segundo argumento es el valor *"Velasquez"* que se asignará al parámetro *apellido*. Y luego se ejecutará el cuerpo de la función imprimiendo el saludo personalizado.

Codificar y ejecutar

```
1. def saludoPersonalizado(nombre, apellido):
2.   print("Hola "+nombre+" "+apellido)
3.
4. saludoPersonalizado("Juliana", "Correa")
5. saludoPersonalizado("Gonzalo", "Velasquez")
```

```
Hola Juliana Correa
Hola Gonzalo Velasquez
```

Figura 14-8. Ejecución del código anterior.

Error en invocación

El siguiente código muestra un error al definir la invocación de una función. En la línea 1 definimos una función llamada *sumar* que recibe dos parámetros (*num1* y *num2*). Por dentro, esta función suma los dos parámetros e imprime el resultado de esa suma. En la línea 5 invocamos la función sumar. El problema es que, al invocar esta función, no estamos pasando los dos argumentos que se requieren, y, por lo tanto, obtenemos un error como el que se presenta en la Figura 14-9. **Nota:** intente modificar el código para que se ejecute sin problema (envíe dos argumentos cualesquiera).

Codificar y ejecutar

```
1.  def sumar(num1, num2):
2.    suma = num1+num2
3.    print(suma)
4.
5.  sumar()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-e2b8f64b0da6> in <module>
      3   print(suma)
      4
----> 5 sumar()

TypeError: sumar() missing 2 required positional arguments: 'num1' and 'num2'
```

Figura 14-9. Ejecución del código anterior.

14.4. Ámbito de las variables

Las variables que definimos a lo largo de nuestros programas tienen un ámbito. El **ámbito de una variable** es la zona o sección del programa donde podemos manipular y acceder a esa variable. En Python, existen dos ámbitos: global y local.

Una variable o parámetro que sea asignado dentro de una función solo existe en el ámbito **local** de esa función. Mientras que una variable que sea asignada por fuera de todas las funciones, existe en el ámbito **global**.

Ámbito local

Para entender mejor el significado de ámbito local, analicemos el siguiente código.

Codificar y ejecutar

```
1.  def sumar(num1, num2):
2.    suma = num1+num2
3.    print(suma)
4.
5.  sumar(4, 6)
6.  print(suma)
```

La variable *suma*, y los parámetros *num1* y *num2* **solo existen en el ámbito local** de la función *sumar*. Esto quiere decir que solo podemos manipular estos tres elementos dentro del cuerpo de la función *sumar* (es decir, *suma*, *num1* y *num2* solo existen como variables dentro de su función). En la línea 6 tratamos de acceder a la variable *suma*, sin embargo, si ejecutamos el código (ver Figura 14-10) veremos un error por pantalla indicando que la variable *suma* no está definida (no existe).

```
10
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-0d0fa9a73226> in <module>
      4
      5 sumar(4, 6)
----> 6 print(suma)

NameError: name 'suma' is not defined
```

Figura 14-10. Ejecución del código anterior.

Ámbito global

Para entender mejor cómo funciona el ámbito global, analicemos el siguiente código que calcula el precio final de un producto, al sumarle a su valor los impuestos del país.

Codificar y ejecutar

```
1.  def calcularPrecioFinal(precio):
2.    precioFinal = precio+impuestos
3.    print(precioFinal)
4.
5.  impuestos = 10
6.  calcularPrecioFinal(100)
```

En este caso, la variable *impuestos* (definida en la línea 5) **existe en el ámbito global**. Esto quiere decir que esta variable se puede manipular y acceder desde cualquier lugar del programa. En la línea 2 accedemos al valor de la variable *impuestos* sin inconveniente, aunque no esté definida dentro de la función (por ser de ámbito global).

La Figura 14-11 muestra una posible representación gráfica de las variables del código anterior, con sus ámbitos respectivos. Es importante resaltar, que las variables *precio* y *precioFinal* de la función *calcularPrecioFinal* solo existen mientras la función es invocada y su cuerpo se está ejecutando. Una vez finalice la ejecución del cuerpo de la función, las variables desaparecen.

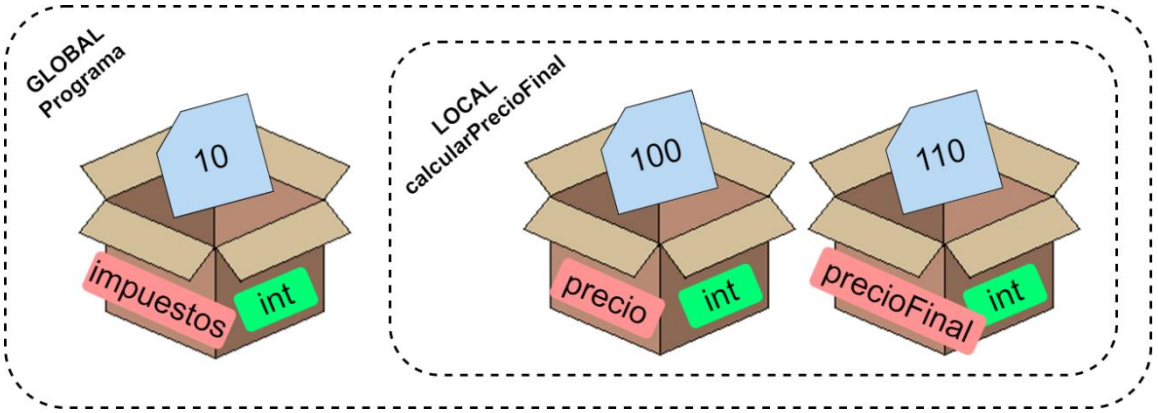


Figura 14-11. Representación gráfica de las variables del código anterior, incluyendo su ámbito.

14.5. Funciones con retorno

Las funciones, además de recibir información (asociada a parámetros), pueden retornar información. Esto se conoce como una **función con retorno**. Por ejemplo, la función *input* retorna un texto (la información que ingresa el usuario por teclado). La función *len* retorna un número *int* (el número de elementos que tiene una lista, o el número de caracteres que hay en un texto).

Si deseamos crear una función que retorne algún valor, debemos colocar la palabra reservada de Python ***return***, seguida del valor a retornar, todo esto dentro del cuerpo de la función.

Por ejemplo, supongamos que trabajamos en una biblioteca, y nos piden crear un programa que calcule cuánto dinero debe pagar alguien que no haya regresado un libro a tiempo. El siguiente código muestra cómo diseñar una función para realizar ese cálculo, basado en un escenario ficticio (ver ejecución en Figura 14-12).

- En la línea 1 definimos la función *calcularMulta* que recibe un parámetro (los días de retraso de la entrega del libro).

- En las líneas 2 a 5 definimos el cuerpo de la función. En este caso la función verifica si los *diasDeRetraso* son mayores a *7*. En caso afirmativo, se genera una multa de *2* dólares por cada uno de los *diasDeRetraso* y se retorna ese resultado. De lo contrario, se genera una multa de *3* dólares y se retorna ese valor.
- Luego, en la línea 7 solicitamos al usuario por teclado que ingrese los días de retraso de la entrega del libro.
- Luego, en la línea 8 invocamos la función, y enviamos un argumento (los *dias* ingresados por teclado, que serán asignamos al parámetro *diasDeRetraso*). Y el resultado retornado por la función, será asignado a la variable *multa*.
- Finalmente, en la línea 9 imprimimos el mensaje con el valor de multa a pagar.

Codificar y ejecutar

```
1. def calcularMulta(diasDeRetraso):
2.   if(diasDeRetraso > 7):
3.     return diasDeRetraso*2
4.   else:
5.     return 3
6.
7. dias = int(input("Ingrese los días de retraso: "))
8. multa = calcularMulta(dias)
9. print("Debe pagar "+str(multa)+" dólares de multa")
```

```
Ingrese los días de retraso: 10
Debe pagar 20 dólares de multa
```

Figura 14-12. Ejecución del código anterior ingresando *10*.

14.6. Calculadora

Dado que hemos adquirido múltiples conocimientos sobre diversos elementos de Python, desarrollemos un ejercicio más complejo.

El siguiente código muestra una calculadora que solo suma (ver ejecución en Figura 14-13).

- En las líneas 1 a 5 definimos la función *sumar*. Esta será la primera opción que soporta nuestra calculadora. Esta función se encarga de pedir dos números al usuario por teclado, y luego los suma y muestra el resultado por pantalla.
- En la línea 7 definimos un ciclo infinito.

- En las líneas 8 a 10 mostramos las diferentes opciones que soporta nuestra calculadora. Por el momento solo soportamos dos opciones: *1* para sumar y *0* para salir.
- En la línea 11 le pedimos al usuario que ingrese la opción que desea ejecutar.
- Si el usuario ingresa 0, se ejecutarán las líneas 15 y 16, así el programa mostrará un mensaje de despedida y finalizará su ejecución. Pero si el usuario ingresa 1, se invocará la función *sumar*, y una vez finalice su ejecución se volverá a ejecutar el ciclo infinito.

Codificar y ejecutar

```
1.  def sumar():
2.    num1 = float(input("Ingrese el primer número: "))
3.    num2 = float(input("Ingrese el segundo número: "))
4.    resultado = num1+num2
5.    print("La suma es: "+str(resultado))
6.
7.  while(True):
8.    print("----Ingrese----")
9.    print("1) Para sumar")
10.   print("0) Para salir")
11.   opcion = int(input())
12.   if(opcion == 1):
13.     sumar()
14.   elif (opcion == 0):
15.     print("Chao")
16.     break
```

```
----Ingrese----
1) Para sumar
0) Para salir
1
Ingrese el primer número: 6
Ingrese el segundo número: 4
La suma es: 10.0
----Ingrese----
1) Para sumar
0) Para salir
0
Chao
```

Figura 14-13. Ejecución del código anterior ingresando *1, 6, 4* y *0* respectivamente.

Ejecutemos el programa anterior un par de veces para entender su funcionamiento. Y luego continuemos añadiendo una nueva opción.

Opción restar

Modifiquemos el código anterior para que nuestra calculadora también soporte las restas. El siguiente código muestra resaltado en **negrilla** las líneas de código que agregamos para soportar la resta (ver ejecución en Figura 14-14).

- En las líneas 7 a 11 añadimos la función *restar*. Esta será una nueva opción que soportará nuestra calculadora. Esta función se encarga de pedir dos números al usuario por teclado, y luego los resta y muestra el resultado por pantalla.
- En la línea 16 adicionamos el texto correspondiente a la nueva opción para que sea mostrado por pantalla (*2* para restar).
- Y en las líneas 21 y 22 añadimos la condición para que se ejecute la función *restar* cuando el usuario ingrese la opción *2*.

Modificar y ejecutar

```
1.  def sumar():
2.    num1 = float(input("Ingrese el primer número: "))
3.    num2 = float(input("Ingrese el segundo número: "))
4.    resultado = num1+num2
5.    print("La suma es: "+str(resultado))
6.
7.  def restar():
8.    num1 = float(input("Ingrese el primer número: "))
9.    num2 = float(input("Ingrese el segundo número: "))
10.   resultado = num1-num2
11.   print("La resta es: "+str(resultado))
12.
13. while(True):
14.   print("----Ingrese----")
15.   print("1) Para sumar")
16.   print("2) Para restar")
17.   print("0) Para salir")
18.   opcion = int(input())
19.   if(opcion == 1):
20.     sumar()
21.   elif(opcion == 2):
22.     restar()
23.   elif(opcion == 0):
24.     print("Chao")
25.     break
```

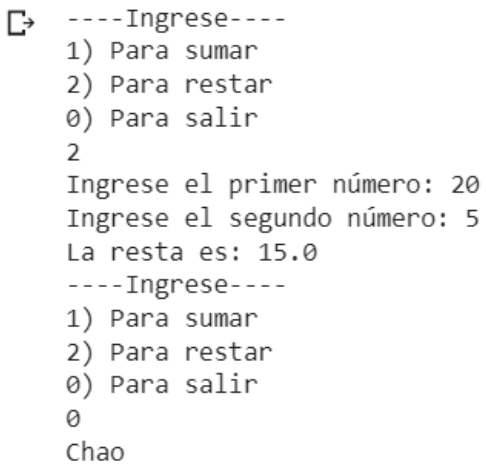


Figura 14-14. Ejecución del código anterior ingresando *2, 20, 5* y *0* respectivamente.

Nota: en la sección de ejercicios tendremos algunos retos para agregar más opciones a la calculadora.

14.7. Ejercicios

Ejercicios teóricos

E14.1. Analice el siguiente código y responda: (i) ¿cuál es el nombre de la función definida?, (ii) ¿cuántos argumentos recibe la función?, (iii) ¿la función tiene retorno?, (iv) ¿cuántas veces se invoca la función? Y (v) ¿cuál es la salida por pantalla?

Analizar

```
1. def mensajeRaro(mensaje):
2.   print("hola"+mensaje+"hola")
3.
4. mensajeRaro("Edificio")
```

E14.2. ¿Qué palabra reservada de Python se utiliza para retonar datos de una función?

E14.3. Analice el siguiente código y mencione cuáles líneas de código se ejecutan.

Analizar

```
1. def loteria(numero):
2.   if(numero == 88):
3.     print("Ganador")
4.
5. loteria(67)
6. loteria(88)
```

Ejercicios prácticos

E14.4. Implemente una función que reciba una lista de nombres y la función deberá imprimir el primer nombre que se encuentra en la lista. **Nota:** complete el siguiente código.

Analizar

```
1. def primerNombre(nombres):
2.
3.
4. nombres = ["Zeus", "Poseidon", "Ares"]
5. primerNombre(nombres)
```

E14.5. Implemente las siguientes opciones adicionales al programa de la calculadora.

- Implemente una opción que permita multiplicar dos números.
- Implemente una opción que permita dividir dos números. **Nota:** verifique que el segundo número es diferente de cero para poder realizar la división. Si el segundo número es diferente de cero, imprima el resultado de la división. De lo contrario, imprima el mensaje *"Denominador inválido"*.
- Implemente una opción que permita elevar un número al cuadrado.

Resumen

En este capítulo aprendimos cómo se comportan las funciones en Python. Aprendimos a definir funciones y su estructura interna. Aprendimos a definir funciones con parámetros y con retorno. Aprendimos los dos tipos de ámbitos de las variables en Python (global y local). Y desarrollamos una calculadora sobre la cual aplicamos muchos de los conceptos vistos en este libro.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una "X" los siguientes trucos: T68, T69, T70, T71, T72 y T73.

En el siguiente capítulo aprenderemos tres tipos de operaciones muy comunes en programación, las cuales las abordaremos a través de contadores, acumuladores y banderas.

Capítulo 15 – Contadores, acumuladores y banderas

Existen tres tipos de operaciones muy comunes cuando desarrollamos diferentes programas. Contar cosas, acumular valores, y verificar si cierta situación ha ocurrido. Por ejemplo, contar cuántos estudiantes perdieron un curso (para un programa universitario). Acumular el salario que devenga cada persona en una empresa durante el mes para pagar la nómina (para un programa empresarial). Y verificar si existe alguna persona con tipo de sangre O+ (para un programa hospitalario).

En este capítulo explicaremos cómo funcionan los contadores, acumuladores y banderas.

A continuación, desarrollaremos las siguientes secciones:

1. Contadores.
2. Acumuladores.
3. Banderas.
4. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo15. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

15.1. Contadores

En programación, un **contador** es una variable que se utiliza para llevar la cuenta de algo. Normalmente, realizamos un conteo cuando tenemos que analizar múltiples valores similares. Por ejemplo, varias edades, salarios, precios de productos, entre otros.

Para utilizar un contador, normalmente definimos una variable que represente el contador y la inicializamos en cero. Luego, recorremos los valores a analizar mediante el uso de un ciclo. Y dentro de ese ciclo cambiamos el valor del contador sumándole o restándole un **valor constante**. El caso más utilizado es **incrementar el contador en una unidad**.

Por ejemplo, podríamos utilizar contadores para contar: (i) cuántas personas ganaron un examen, (ii) cuántos votos son válidos, (iii) cuántas computadoras vendimos en el mes, entre otros.

Contadores con listas

El siguiente código muestra cómo definir y utilizar un contador sobre los valores de una lista. La lista está compuesta de diferentes edades de personas que trabajan en una empresa, y deseamos contar cuántas de esas personas tienen más de 30 años (ver ejecución en Figura 15-1).

- En la línea 1 definimos la lista *edades* que contiene las edades de las personas de la empresa.
- En la línea 3 definimos el contador llamado *contadorMayores30* (aquí llevaremos el conteo de la cantidad de personas mayores de 30 años). Inicializamos el contador en *0* dado que todavía no hemos analizado ninguna edad.
- En las líneas 4 a 8 definimos los elementos del ciclo para recorrer la lista de edades.
- En la línea 6 definimos un condicional para verificar si la edad sobre la que estamos iterando es mayor a *30*. En caso afirmativo, en la línea 7, modificamos el valor de *contadorMayores30* (tomamos su valor actual y le aumentamos una unidad).
- Al terminar la ejecución del ciclo, se ejecuta la línea 10 y se imprime por pantalla la cantidad de personas mayores de 30 años que encontramos en la lista (se imprime el valor de *contadorMayores30*).

Codificar y ejecutar

```
1.  edades = [34, 50, 28, 20, 44]
2.
3.  contadorMayores30 = 0
4.  i = 0
5.  while(i < len(edades)):
6.    if(edades[i] > 30):
7.      contadorMayores30 = contadorMayores30+1
8.    i = i+1
9.
10. print("Total mayores de 30: "+str(contadorMayores30))
```

```
Total mayores de 30: 3
```

Figura 15-1. Ejecución del código anterior.

Contadores con listas que contienen diccionarios

El siguiente código muestra cómo definir y utilizar un contador sobre los valores de una lista que contiene diccionarios. La lista está compuesta de tres diccionarios que almacenan datos de ciudades, y deseamos contar cuántas ciudades tienen más de 90000 habitantes (ver ejecución en Figura 15-2).

- En las líneas 1 a 3 definimos tres diccionarios, cada uno con los datos de una ciudad.
- En la línea 5 agrupamos las ciudades en la lista *ciudades*.
- En la línea 6 definimos el contador con valor de *0*.
- En la línea 8 definimos el ciclo para recorrer las ciudades.
- En la línea 9 definimos un condicional para verificar si la cantidad de habitantes de la ciudad sobre la que estamos iterando es mayor a *90000*. En caso afirmativo, en la línea 10, modificamos el contador aumentándole una unidad a su valor actual.
- Al terminar la ejecución del ciclo, se ejecuta la línea 12, en la que se imprime por pantalla la cantidad de ciudades que tienen más de 90000 habitantes.

Codificar y ejecutar

```
1.  ciudad1 = {"nombre":"Envigado", "habitantes": 249800}
2.  ciudad2 = {"nombre":"Sabaneta", "habitantes": 87981}
3.  ciudad3 = {"nombre":"Rionegro", "habitantes": 147484}
4.
5.  ciudades = [ciudad1, ciudad2, ciudad3]
6.  contador = 0
7.
8.  for ciudad in ciudades:
9.    if(ciudad["habitantes"] > 90000):
10.     contador = contador+1
11.
12. print("Más de 90000 habitantes: "+str(contador))
```

```
Más de 90000 habitantes: 2
```

Figura 15-2. Ejecución del código anterior.

15.2. Acumuladores

En programación, un **acumulador** es una variable que se utiliza para acumular valores.

Para utilizar un acumulador, normalmente definimos una variable que represente el acumulador y la inicializamos en cero. Luego, recorremos los valores a analizar mediante el uso de un ciclo. Y dentro de ese ciclo, cambiando el valor del acumulador sumándole o restándole el **valor de una variable**. Es decir, no siempre se le suma o resta la misma cantidad.

Por ejemplo, podríamos utilizar acumuladores cuando queremos: (i) sumar el precio de todos los productos de una compra (para generar la factura), (ii) sumar el salario de cada trabajador (para calcular la nómina total de la empresa), (iii) sumar los ahorros acumulados en una alcancía o cuenta de ahorros, entre otros.

Acumuladores con listas

El siguiente código muestra cómo definir y utilizar un acumulador sobre los valores de una lista. La lista está compuesta de diferentes salarios de personas de una empresa, y deseamos sumar todos los salarios para calcular la nómina de la empresa (ver ejecución en Figura 15-3).

- En la línea 1 definimos la lista *salarios* que contiene los salarios de las personas.
- En la línea 3 definimos el acumulador llamado *acumuladorSalarios* (aquí acumularemos los salarios de las personas). Inicializamos el acumulador en *0* dado que todavía no hemos acumulado ningún salario.
- En las líneas 4 a 7 definimos los elementos del ciclo para recorrer la lista que contiene los salarios.
- En la línea 6, modificamos el valor de *acumuladorSalarios* (tomamos su valor actual y le aumentamos el valor del salario sobre el cual estamos iterando).
- Al terminar la ejecución del ciclo, se ejecuta la línea 9, la cual imprime por pantalla la suma de los salarios.

Codificar y ejecutar

```
1.  salarios = [3000, 1500, 800]
2.
3.  acumuladorSalarios = 0
4.  i = 0
5.  while(i < len(salarios)):
6.    acumuladorSalarios = acumuladorSalarios+salarios[i]
7.    i = i+1
8.
9.  print("Total suma de salarios: "+str(acumuladorSalarios))
```

```
Total suma de salarios: 5300
```

Figura 15-3. Ejecución del código anterior.

Acumuladores con listas que contienen diccionarios

El siguiente código muestra cómo definir y utilizar un acumulador sobre los valores de una lista que contiene diccionarios. La lista está compuesta de tres diccionarios que almacenan datos de ciudades, y deseamos encontrar en promedio cuántos habitantes hay en las tres ciudades (ver ejecución en Figura 15-4).

- En las líneas 1 a 3 definimos tres diccionarios, cada uno con los datos de una ciudad.
- En la línea 5 agrupamos las ciudades en la lista *ciudades*.
- En la línea 6 definimos el *acumulador* con valor de *0*.
- En la línea 8 definimos el ciclo para recorrer las ciudades de la lista.
- En la línea 9 modificamos el *acumulador* tomando su valor actual y sumándole la cantidad de habitantes de la ciudad sobre la que estamos iterando.
- En la línea 11 calculamos el promedio de habitantes de las ciudades (tomamos *acumulador* y dividimos su valor entre la cantidad de ciudades).
- Finalmente, en la línea 12 imprimimos por pantalla el promedio de habitantes.

Codificar y ejecutar

```
1.  ciudad1 = {"nombre":"Envigado", "habitantes":249800}
2.  ciudad2 = {"nombre":"Sabaneta", "habitantes":87981}
3.  ciudad3 = {"nombre":"Rionegro", "habitantes":147484}
4.
5.  ciudades = [ciudad1, ciudad2, ciudad3]
6.  acumulador = 0
7.
8.  for ciudad in ciudades:
9.    acumulador = acumulador+ciudad["habitantes"]
10.
11. promedio = acumulador/len(ciudades)
12. print("Promedio de habitantes: "+str(promedio))
```

```
Promedio de habitantes: 161755.0
```

Figura 15-4. Ejecución del código anterior.

15.3. Banderas

En programación, una **bandera** es una variable booleana que se utiliza para verificar si cierta situación ha ocurrido.

Para utilizar una bandera, normalmente definimos una variable que representa la bandera y la inicializamos en *False*. Luego, recorremos los valores a analizar mediante el uso de un ciclo. Y dentro de ese ciclo, si ocurre cierta situación o se cumple cierta condición, modificamos el valor de la bandera a *True*.

Por ejemplo, podríamos utilizar banderas para: (i) verificar si existe al menos una persona mayor de edad, (ii) verificar si existe una persona con tipo de sangre O+, (iii) verificar si un usuario quiere finalizar un programa, entre otros.

El siguiente código muestra cómo definir y utilizar una bandera para verificar si un estudiante obtuvo por lo menos una nota mayor a 4.5 (ver ejecución en Figura 15-5).

- En la línea 1 definimos la lista *notas* que contiene las notas del estudiante.
- En la línea 2 definimos la bandera llamada *banderaNotaAlta*, la cual nos indicará si encontramos una nota alta (mayor a 4.5). Inicializamos la bandera en *False* dado que todavía no hemos analizado ninguna nota.
- En la línea 4 definimos el ciclo.
- En la línea 5 verificamos si la nota sobre la cual estamos iterando es mayor a *4.5*. En caso afirmativo, modificamos la bandera a *True* (línea 6) y detenemos el ciclo con un *break* (ya que no es necesario seguir analizando el resto de los valores de la lista) en la línea 7.
- Finalmente, en la línea 9 verificamos si el valor de *banderaNotaAlta* es *True* o *False*. Si el valor es *True*, mostramos por pantalla el mensaje *"Sacaste por lo menos una nota alta"* (línea 10). En caso contrario, mostramos por pantalla el mensaje *"No sacaste notas altas"* (línea 12).

Codificar y ejecutar

```
1.  notas = [4.0, 2.0, 1.2, 4.7, 1.7]
2.  banderaNotaAlta = False
3.
4.  for nota in notas:
5.    if(nota > 4.5):
6.      banderaNotaAlta = True
7.      break
8.
9.  if(banderaNotaAlta):
10.   print("Sacaste por lo menos una nota alta")
11. else:
12.   print("No sacaste notas altas")
```

```
Sacaste por lo menos una nota alta
```

Figura 15-5. Ejecución del código anterior.

15.4. Ejercicios

Ejercicios teóricos

E15.1. ¿Cuál es la diferencia entre un contador y un acumulador?

E15.2. Analice el siguiente código y mencione cuáles líneas de código se ejecutan.

Analizar

```
1.  edades = [20, 10, 29]
2.
3.  for edad in edades:
4.    if(edad < 15):
5.      break
6.    print(edad)
```

Ejercicios prácticos

E15.3. La Nasa lo contrata para analizar una lista de asteroides y contar cuántos de ellos tienen alta probabilidad de colisión con la tierra (aquellos que se encuentran a una distancia menor a 7500000 kilómetros). La Nasa le entrega una lista de asteroides, que por dentro contiene el nombre y distancia de cada asteroide. Reutilice el siguiente código para contar cuántos asteroides tienen alta probabilidad de colisión.

Analizar

```
1. a1 = {"nombre":"2022 HX1", "distancia":2490000}
2. a2 = {"nombre":"2022 HA2", "distancia":4620000}
3. a3 = {"nombre":"101955 Bennu", "distancia":32490000}
4. asteroides = [a1, a2, a3]
```

E15.4. Basado en el código anterior, muestre por pantalla el promedio de la distancia de todos los asteroides de la lista que le entregó la Nasa.

Resumen

En este capítulo aprendimos cómo llevar a cabo operaciones que involucren contadores, acumuladores y banderas. Aprendimos que los contadores son variables que se utilizan para contar cosas. Aprendimos que los acumuladores son variables que se utilizan para acumular valores. Y aprendimos que las banderas son variables booleanas que se utilizan para verificar si cierta situación ha ocurrido.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una “X” los siguientes trucos: T74, T75, T76, T77, T78 y T79.

En el siguiente capítulo aprenderemos a manipular archivos y a analizar la información contenida en ellos.

Capítulo 16 – Archivos

Hasta el momento hemos utilizado variables para almacenar los datos de nuestros programas. Sin embargo, las variables solo permiten almacenar esos datos en la memoria de la computadora, mientras el programa está en ejecución. Si el programa finaliza su ejecución, todos esos datos desaparecen (no persisten).

En este capítulo explicaremos cómo utilizar archivos para que la información persista (incluso después de que nuestros programas finalicen su ejecución).

A continuación, desarrollaremos las siguientes secciones:

1. Introducción.
2. Crear y abrir archivos.
3. Leer información de archivos.
4. Escribir información en archivos.
5. Datasets.
6. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo16. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

16.1. Introducción

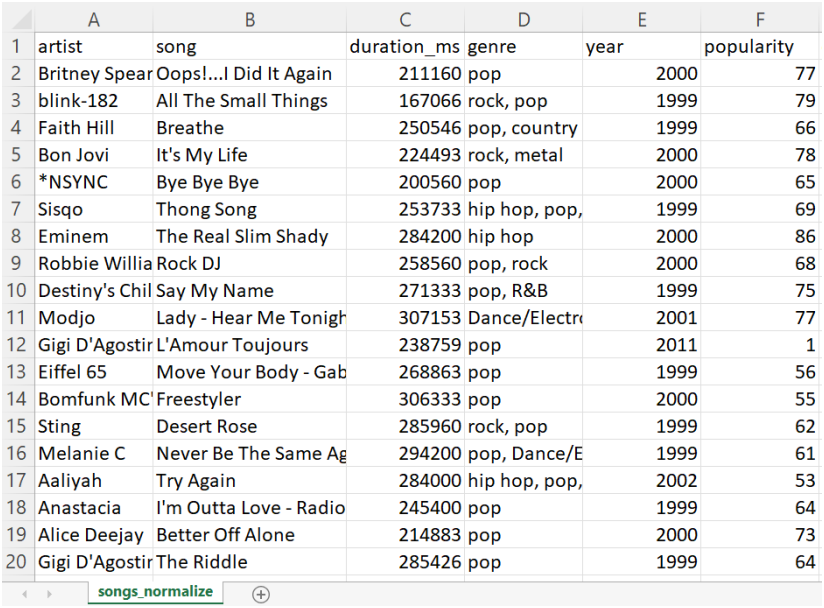
Como vimos anteriormente, las variables no permiten persistir información. Si queremos que la información de nuestros programas persista, incluso después de que nuestros programas finalizan su ejecución, podemos hacerlo mediante **archivos**.

Muchos programas persisten su información en archivos o gestores similares (comúnmente llamados bases de datos). Por ejemplo, los datos de las redes sociales, las compras de una tienda virtual, la información de la nómina de una empresa, los personajes que adquirimos en un videojuego, entre otros.

Aprender a manipular archivos será útil para persistir la información de nuestros programas. Además, podremos utilizar este aprendizaje para analizar información que encontremos en

internet, o que recopilemos por otros medios. Por ejemplo: tendencias de Twitter, datos meteorológicos, datos de tráfico, datos socioeconómicos, obras literarias, entre muchos otros.

La Figura 16-1 muestra un extracto de una hoja de cálculo que contiene información de algunas de las 2000 canciones más populares en Spotify de 1998 a 2020. Con lo que aprendamos en este capítulo, podremos manipular esos datos para realizar operaciones tales como: encontrar los artistas con más canciones, encontrar los géneros más populares, encontrar el promedio de duración de esas canciones, mostrar las canciones más populares en el 2015, entre otros. Pero recuerde que eso no aplicará únicamente para ese archivo y para ese tipo de información, usted como programador lo podrá aplicar en cualquier campo y tipo de información que desee.



	A	B	C	D	E	F
1	artist	song	duration_ms	genre	year	popularity
2	Britney Spear	Oops!...I Did It Again	211160	pop	2000	77
3	blink-182	All The Small Things	167066	rock, pop	1999	79
4	Faith Hill	Breathe	250546	pop, country	1999	66
5	Bon Jovi	It's My Life	224493	rock, metal	2000	78
6	*NSYNC	Bye Bye Bye	200560	pop	2000	65
7	Sisqo	Thong Song	253733	hip hop, pop,	1999	69
8	Eminem	The Real Slim Shady	284200	hip hop	2000	86
9	Robbie Willia	Rock DJ	258560	pop, rock	2000	68
10	Destiny's Chil	Say My Name	271333	pop, R&B	1999	75
11	Modjo	Lady - Hear Me Tonigh	307153	Dance/Electr	2001	77
12	Gigi D'Agostir	L'Amour Toujours	238759	pop	2011	1
13	Eiffel 65	Move Your Body - Gab	268863	pop	1999	56
14	Bomfunk MC'	Freestyler	306333	pop	2000	55
15	Sting	Desert Rose	285960	rock, pop	1999	62
16	Melanie C	Never Be The Same Ag	294200	pop, Dance/E	1999	61
17	Aaliyah	Try Again	284000	hip hop, pop,	2002	53
18	Anastacia	I'm Outta Love - Radio	245400	pop	1999	64
19	Alice Deejay	Better Off Alone	214883	pop	2000	73
20	Gigi D'Agostir	The Riddle	285426	pop	1999	64

songs_normalize

Figura 16-1. Extracto de una hoja de cálculo con canciones populares en Spotify.

16.2. Crear y abrir archivos

Supongamos que queremos crear y manipular un archivo donde almacenamos la información del dinero que vamos ahorrando semana a semana. Antes de manipular esta información, debemos crear un archivo.

Las Figuras 16-2 y 16-3 muestran los pasos que debemos seguir para crear un archivo en Colab.

1. Accedemos a un documento Colab y damos clic en el icono de “carpeta” que aparece en el menú lateral izquierdo. Esta acción nos abre una sección llamada “Archivos” donde

podemos ver algunas carpetas y archivos a los que tiene acceso el entorno de ejecución de Colab.

2. Luego, debajo de la carpeta *sample_data*, presionamos clic derecho y seleccionamos la opción *"Nuevo Archivo"*.
3. Renombramos el archivo y le colocamos por nombre *ahorros.txt*
4. Luego presionamos doble clic sobre el archivo *ahorros.txt* y copiamos en este los valores *150,30,1200* (esperamos un par de segundos a que se guarde automáticamente el archivo).

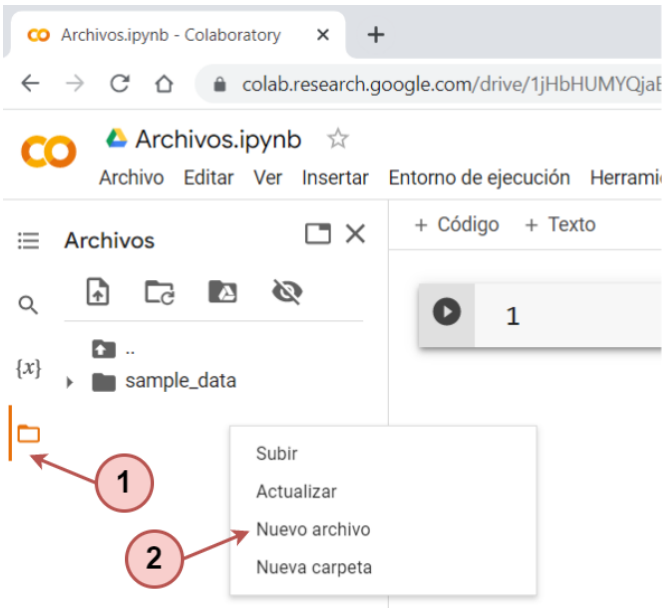


Figura 16-2. Pasos 1 y 2 para la creación de un archivo en Colab.

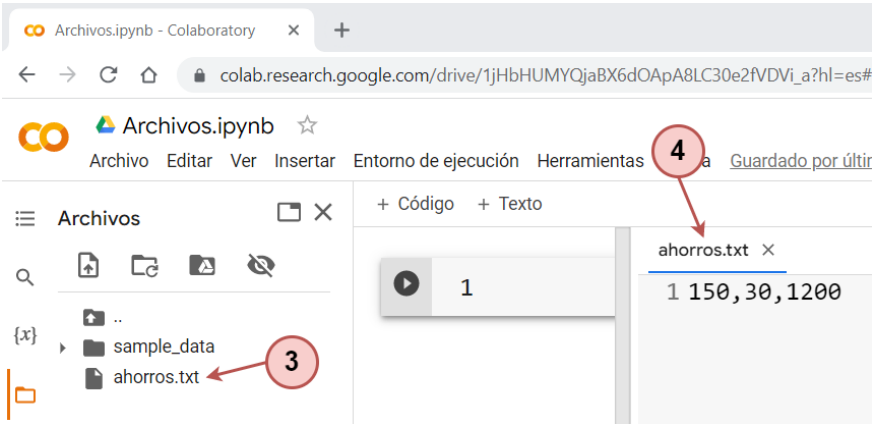


Figura 16-3. Pasos 3 y 4 para la creación de un archivo en Colab.

Suponemos que los tres valores (*150*, *30* y *1200*) que almacenamos en el archivo *ahorros.txt* representan el dinero ahorrado durante las semanas 1, 2 y 3, respectivamente.

Discusión rápida: Los archivos que se creen en Colab siguiendo los pasos anteriores, únicamente persisten mientras el entorno de ejecución de Colab está activo. Esto quiere decir, que, si cierra el navegador, o apaga el computador, esos archivos se borran. Sin embargo, puede leer este artículo donde enseñamos a conectar los archivos Colab, con documentos de Google Drive para que los archivos nunca se borren y la información persista: https://bit.ly/TutorialColabConDrive.

Abrir archivo en código

Dado que ya creamos nuestro archivo *ahorros.txt*, ahora veamos cómo abrirlo desde nuestro código Python. El siguiente código muestra cómo abrir el archivo *ahorros.txt* utilizando la función ***open*** de Python, a la cual le enviamos como argumento el nombre del archivo a abrir. El archivo abierto con toda su información queda asignado a la variable *archivoAhorros*. Esta variable será de tipo *"_io.TextIOWrapper"*, el cual es un tipo propio de Python para manejo de archivos. Si ejecutamos el código y no percibimos ningún error, implica que abrimos satisfactoriamente el archivo.

Codificar y ejecutar

```
1. archivoAhorros = open("ahorros.txt")
```

Dado que ya aprendimos a crear y abrir archivos, veamos en la siguiente sección cómo leer la información de esos archivos.

16.3. Leer información de archivos

Para leer la información contenida en un archivo, basta con utilizar la variable que representa el archivo, y luego invocar el método *read*. El método ***read*** retornará un *str* con todo el contenido del archivo.

El siguiente código muestra cómo leer la información de un archivo (ver ejecución en Figura 16-4). En la línea 1 abrimos el archivo *ahorros.txt*. En la línea 2 leemos el contenido del archivo *ahorros.txt* utilizando el método *read*, y almacenamos la información en la variable *contenidoArchivo*. Finalmente, en la línea 4 imprimimos el contenido del archivo.

Codificar y ejecutar

```
1. archivoAhorros = open("ahorros.txt")
2. contenidoArchivo = archivoAhorros.read()
3.
4. print(contenidoArchivo)
```

```
150,30,1200
```

Figura 16-4. Ejecución del código anterior.

Separar información por comas

Como pudimos observar, la información de los ahorros la separamos por comas. Esto lo hicimos intencionalmente. Utilizamos la coma como un separador de información, lo cual se conoce en programación como un **delimitador**. Es común que la información de muchos de los programas que utilizamos a diario esté almacenada con delimitadores. La coma "," y el punto y coma ";" son dos de los delimitadores más populares. En Excel, por ejemplo, un usuario puede almacenar un documento en formato CSV (que significa valores separados por comas). Pero en realidad, el programador puede definir el delimitador que desee.

Ahora veamos cómo separar la información del archivo. El siguiente código muestra cómo leer y separar la información de un archivo (ver ejecución en Figura 16-5).

- Las líneas 1 y 2 las discutimos en el ejemplo anterior.
- En la línea 3 tomamos la variable *contenidoArchivos* y le invocamos el método *split*. El método ***split*** recibe un argumento (un delimitador) y este método se encarga de dividir una cadena según el delimitador, y retorna una lista con los elementos separados (la cual se asigna a la variable *ahorros*). Dado que *contenidoArchivos* contiene tres valores separados por comas, entonces en esa línea se creará una lista con los tres valores separados.
- En la línea 5 imprimimos la lista *ahorros*.
- En la línea 6 imprimimos el tamaño de *ahorros*, que muestra el valor de *3*.
- Finalmente, en la línea 7 accedemos e imprimimos el valor en la posición *0* de *ahorros* (o sea *150*).

Codificar y ejecutar

```
1. archivoAhorros = open("ahorros.txt")
2. contenidoArchivo = archivoAhorros.read()
3. ahorros = contenidoArchivo.split(",")
4.
5. print(ahorros)
6. print(len(ahorros))
7. print(ahorros[0])
```

```
['150', '30', '1200']
3
150
```

Figura 16-5. Ejecución del código anterior.

Lectura por líneas

Cuando empezamos a manipular más información, es muy común dividir los datos almacenados en un archivo por líneas. Por ejemplo, si tenemos un sistema de clientes, la información de cada cliente la podríamos almacenar en una línea diferente de un archivo. Los datos del primer cliente en la primera línea del archivo, los datos del segundo cliente en la segunda línea del archivo, y así sucesivamente.

Primero creemos un archivo llamado *clientes.txt* con el siguiente texto (ver Figura 16-6). Este archivo almacena la información de diferentes clientes separada por líneas. En cada línea incluimos: su nombre, su apellido y su edad.

Archivo clientes.txt

```
1. Carlos,Cardona,41
2. Veronica,Calderon,24
3. Sara,Saldarriaga,30
```

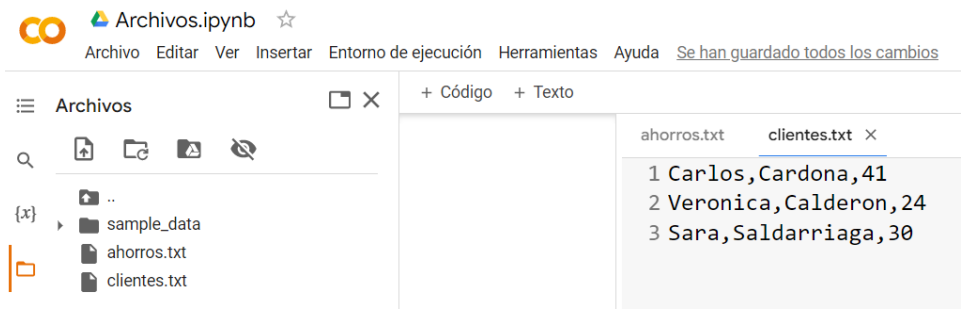


Figura 16-6. Archivo *clientes.txt* en Colab.

El siguiente código muestra cómo leer la información contenida en múltiples líneas de un archivo (ver ejecución en Figura 16-7).

- En la línea 1 abrimos el archivo *clientes.txt* y lo asignamos a la variable *archivoClientes*.
- En la línea 2 leemos el contenido del archivo de clientes, pero esta vez por líneas, utilizando el método readlines. El método ***readlines*** retorna una lista donde cada posición representa una línea del archivo que leímos (esa lista la asignamos a *contenidoPorLineas*). En este caso, en la posición *0* de *contenidoPorLineas* estará el contenido de la primera línea de *clientes.txt* (*Carlos,Cardona,41*).
- En la línea 4 recorremos *contenidoPorLineas* con un ciclo *for*. En la primera iteración, la variable *linea* quedará asociada a la primera línea del archivo.
- En la línea 5 limpiamos la línea. Esto es, utilizamos el método *strip* (disponible para valores y variables *str*), el cual nos permite borrar espacios indeseados o saltos de líneas.
- En la línea 6 tomamos la variable *lineaLimpia* y le invocamos el método *split*. Esta vez *split* nos retornará una lista donde la posición *0* será el nombre del cliente de la primera línea, la posición *1* el apellido del cliente de la primera línea y la posición *2* la edad del cliente de la primera línea.
- En las líneas 7 a 9 imprimimos los datos del cliente de la primera línea.
- Luego, el ciclo se vuelve a ejecutar tantas líneas existan en el archivo *clientes.txt*, en este caso se ejecuta tres veces.

Codificar y ejecutar

```
1. archivoClientes = open("clientes.txt")
2. contenidoPorLineas = archivoClientes.readlines()
3.
4. for linea in contenidoPorLineas:
5.   lineaLimpia = linea.strip()
6.   datosCliente = lineaLimpia.split(",")
7.   print("Nombre: "+datosCliente[0])
8.   print("Apellido: "+datosCliente[1])
9.   print("Edad: "+datosCliente[2])
```

```
Nombre: Carlos
Apellido: Cardona
Edad: 41
Nombre: Veronica
Apellido: Calderon
Edad: 24
Nombre: Sara
Apellido: Saldarriaga
Edad: 30
```

Figura 16-7. Ejecución del código anterior.

16.4. Escribir información en archivos

Para escribir información en archivos debemos realizar una variación al uso de la función *open* y utilizar dos métodos nuevos: *write* y *close*. El siguiente código muestra cómo escribir información en un archivo (ver ejecución en Figura 16-8).

- En la línea 1 utilizamos la función *open* para abrir el archivo *ahorros.txt*.
- En la línea 2 leemos el contenido completo del archivo de *ahorros.txt* y lo asignamos a la variable *contenidoArchivo*.
- En la línea 4 volvemos a abrir el archivo *ahorros.txt* con la función *open*. Esta vez, a la función *open* le pasamos un segundo argumento (el texto *"w"*). Este argumento indica que esta vez estamos abriendo el archivo en modo escritura, y lo asignamos a la variable *archivoAhorrosW*.
- En la línea 5 le pedimos al usuario por pantalla que ingrese un nuevo valor de ahorro y lo asignamos a la variable *nuevoAhorro*.

- En la línea 6 concatenamos en la variable *nuevoContenido* el contenido actual del archivo, con el texto *","* (que nos sirve para separar el ahorro anterior del nuevo ahorro), y finalmente añadimos el nuevo ahorro que ingresó el usuario por pantalla.
- En la línea 8 utilizamos el método *write* sobre la variable *archivoAhorrosW* y le enviamos un argumento (*nuevoContenido*). El método ***write*** sobrescribe todo el contenido anterior del archivo y lo reemplaza con el contenido de la variable *nuevoContenido*.
- Dado que abrimos el archivo en modo lectura y escritura, debemos cerrar el archivo en la línea 9 utilizando el método ***close*** sobre la variable *archivoAhorrosW*.

Codificar y ejecutar

```
1. archivoAhorros = open("ahorros.txt")
2. contenidoArchivo = archivoAhorros.read()
3.
4. archivoAhorrosW = open("ahorros.txt", "w")
5. nuevoAhorro = input("Ingrese el ahorro de la semana: ")
6. nuevoContenido = contenidoArchivo + "," + nuevoAhorro
7.
8. archivoAhorrosW.write(nuevoContenido)
9. archivoAhorrosW.close()
```

```
Ingrese el ahorro de la semana: 300
```

Figura 16-8. Ejecución del código anterior ingresando *300*.

Si ejecutamos el código anterior, e ingresamos un valor de ahorro, el programa modificará el archivo *ahorros.txt* y añadirá el valor ingresado al final del archivo. Cierre y abra en Colab el archivo *ahorros.txt* para identificar los cambios.

Discusión rápida: La función *open* provee otros valores que pueden ser enviados como segundo argumento (adicionales al que ya analizamos *"w"*). Por ejemplo, *"a"* que indica que el archivo se abre para escritura y que el contenido que se escriba se añadirá al final del archivo. La ventaja de *"w"* respecto a *"a"*, es que *"w"* nos permite modificar completamente el archivo, y no solo adicionar información al final de este. Puede encontrar más información sobre la función *open* en el siguiente enlace: https://docs.python.org/es/3/library/functions.html#open.

16.5. Datasets

Un **dataset** es una colección de datos habitualmente tabulada o separada por algún carácter específico (un delimitador). Cada columna del dataset representa una variable y cada fila corresponde a un elemento particular al que le estamos almacenando información. Por ejemplo, una fila podría corresponder a datos de una persona, de un paciente, de un producto, de una canción, etc.

A continuación, utilizaremos un dataset con información de algunas de las 2000 canciones más populares de Spotify de los últimos 20 años. Primero, descargue el siguiente archivo a su computador: https://bit.ly/CancionesCSV (tomado de: https://www.kaggle.com/datasets/paradisejoy/top-hits-spotify-from-20002019) y luego, siga los pasos de la Figura 16-9.

1. Arrastre el archivo *canciones.csv* (descargado desde el enlace anterior) a la sección de "Archivos" de Colab.
2. Presione doble clic sobre el archivo *canciones.csv* y podrá ver el contenido del archivo con la información de las canciones.

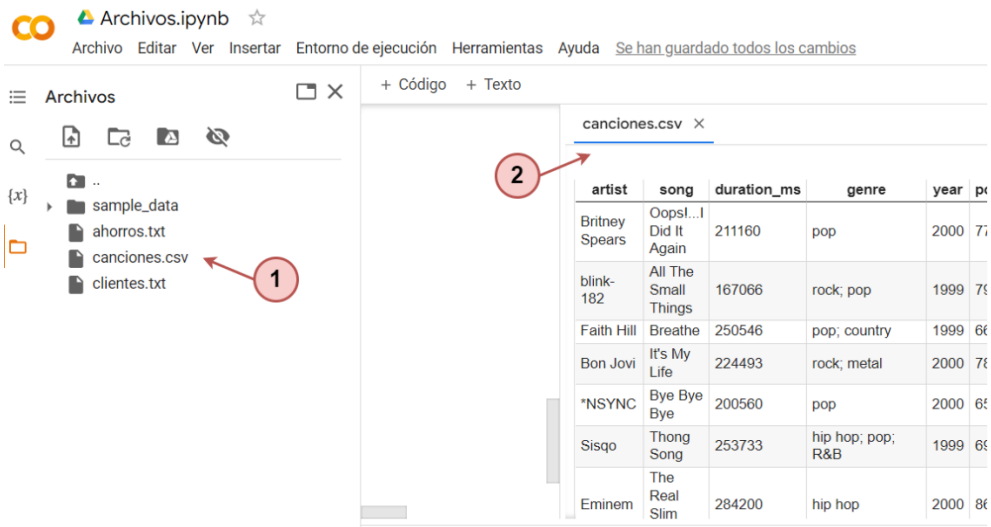


Figura 16-9. Carga del archivo *canciones.csv* a Colab.

Lectura de datos del dataset

Ahora leamos lo datos del dataset. El siguiente código muestra cómo leer y mostrar por pantalla los nombres de las canciones almacenadas en el dataset (ver extracto de la ejecución en Figura 16-10).

- En la línea 1 abrimos el archivo *canciones.csv* y lo asignamos a la variable *archivoCanciones*.
- En la línea 2 leemos el contenido del archivo de canciones por líneas y lo asignamos a la variable *contenidoPorLineas*.
- En la línea 4 definimos la variable de control *i* del ciclo. Esta vez inicializamos *i* en *1* dado que omitiremos la posición *0* porque allí se encuentra la línea que contiene los encabezados de las columnas.
- En las líneas 5 a 9 implementamos el ciclo, limpiamos la línea sobre la cual estamos iterando, separamos el texto de la línea por comas, y finalmente accedemos y mostramos por pantalla la posición *1* de *datosCancion* (que corresponde al nombre de la canción de cada línea).

Codificar y ejecutar

```
1. archivoCanciones = open("canciones.csv")
2. contenidoPorLineas = archivoCanciones.readlines()
3.
4. i = 1
5. while(i < len(contenidoPorLineas)):
6.   lineaLimpia = contenidoPorLineas[i].strip()
7.   datosCancion = lineaLimpia.split(",")
8.   print(datosCancion[1])
9.   i = i+1
```

```
Cruel Summer
The Git Up
Dancing With A Stranger (with Normani)
Circles
```

Figura 16-10. Extracto final de ejecución del código anterior.

16.6. Ejercicios

Ejercicios teóricos

E16.1. ¿Para qué sirve el método *read*?

E16.2. ¿Para qué sirve el método *readlines*?

Ejercicios prácticos

E16.3. Basado en el dataset de canciones (*canciones.csv*) utilizado en este capítulo, implemente los siguientes ejercicios como programas individuales.

- Implemente un programa que muestre por pantalla los nombres de las canciones lanzadas en el año 2000. **Pista:** podría utilizar la siguiente instrucción en el cuerpo del ciclo: *if(int(datosCancion[4]) == 2000):*
- Implemente un programa que muestre por pantalla los nombres de las canciones del artista “Eminem”.
- Implemente un programa que muestre por pantalla los nombres de las canciones que empiezan por la letra “D”.
- Implemente un programa que muestre por pantalla cuántas canciones tienen una duración de más de 400000 milisegundos.
- Implemente un programa que muestre por pantalla el promedio de popularidad de todas las canciones.

Resumen

En este capítulo aprendimos a manipular archivos. Aprendimos a crear archivos en Colab para almacenar información. Aprendimos a abrir archivos con el uso de la función *open*. Aprendimos a leer archivos completos con el uso del método *read*, y archivos por líneas con *readlines*. Aprendimos a abrir archivos en modo escritura y a escribirles información con el uso del método *write*. Y aprendimos a manipular datasets.

Hoja de trucos: Ahora tomaremos la hoja de trucos y marcaremos los trucos relacionados con los temas aprendidos en este capítulo. Marque con una “X” los siguientes trucos: T42, T80, T81, T82, T83, T84 y T85.

En el siguiente capítulo aprenderemos a utilizar librerías en Python, específicamente Matplotlib.

Capítulo 17 – Librerías – Matplotlib

En los capítulos anteriores aprendimos a utilizar diversos tipos de variables, elementos y funciones que proporciona Python. Pero, Python también proporciona una librería estándar que contiene múltiples módulos y componentes adicionales para desarrollar diferentes tipos de programas. Incluso existe un índice de paquetes de Python donde hay una colección de cientos de miles de componentes para realizar diferentes tipos de programas con Python.

En este capítulo presentaremos una breve introducción a las librerías en Python, y aprenderemos a utilizar algunas de ellas.

A continuación, desarrollaremos las siguientes secciones:

1. Introducción.
2. La librería estándar de Python.
3. El índice de paquetes de Python (PyPI).
4. Introducción a Matplotlib.
5. Ejercicios.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo17. Le recomendamos desarrollar los códigos por usted mismo (para adquirir habilidades en codificación), y si tiene problemas, puede contrastar con el código del repositorio.

Recuerde que le recomendamos que, para cada capítulo del libro, cree un nuevo documento Colab donde codifique los códigos presentados en el capítulo.

17.1. Introducción

Una **librería** o biblioteca de Python es una colección de módulos y/o funciones que están relacionados. Internamente, la librería contiene partes de código que se pueden reutilizar fácilmente en los programas que desarrollemos.

A continuación, analizaremos una librería que Python incluye por defecto, y otras librerías creadas por la comunidad de programadores de Python.

17.2. La librería estándar de Python

Python incorpora lo que se conoce como “La Librería Estándar de Python” (https://docs.python.org/es/3/library/index.html). Esta librería contiene múltiples módulos que

permiten ejecutar operaciones comunes para diferentes tipos de programas. Y que, de esta manera, el programador ahorre tiempo y se pueda concentrar en el desarrollo de sus propios programas (que no tenga que "reinventar la rueda").

En esta sección analizaremos dos de los módulos más utilizados de "La Librería Estándar de Python": *math* y *random*.

Módulo *math*

Este módulo proporciona acceso a múltiples elementos y funciones relacionados con matemática.

El siguiente código muestra cómo utilizar un módulo en Python (en este caso el módulo *math*, https://docs.python.org/es/3/library/math.html) y cómo acceder a algunas de las funciones de este módulo (ver ejecución en Figura 17-1).

- En la línea 1 utilizamos la palabra reservada de Python *import* para indicar que queremos importar un módulo. Y luego agregamos el nombre del módulo a importar (en este caso *math*).
- En la línea 3 empezamos a utilizar el módulo *math*. En este caso invocamos la función *sqrt* del módulo *math*. La función *sqrt* recibe un argumento, calcula la raíz cuadrada de ese argumento y retorna el resultado.
- En la línea 4 volvemos a utilizar el módulo *math*. Esta vez, invocamos la función *fabs* la cual calcula el valor absoluto del argumento recibido y retorna el resultado.

Codificar y ejecutar

```
1. import math
2.
3. print(math.sqrt(16))
4. print(math.fabs(-8))
```

```
4.0
8.0
```

Figura 17-1. Ejecución del código anterior.

Módulo *random*

Este módulo proporciona acceso a múltiples elementos y funciones relacionados con la generación de números aleatorios.

El siguiente código muestra cómo utilizar el módulo *random* (https://docs.python.org/es/3/library/random.html) y cómo acceder a algunas de sus funciones (ver ejecución en Figura 17-2).

- En la línea 1 importamos el módulo *random*.
- En la línea 3 invocamos la función *randint* del módulo *random*. Esta función recibe dos argumentos (dos números *int*), y retorna un número *int* aleatorio entre estos dos números.
- En la línea 4 imprimimos el número aleatorio generado.
- En la línea 6 creamos una lista de candidatos a un sorteo.
- En la línea 7 invocamos la función *choice* del módulo *random*. Esta función recibe una secuencia (en este caso una lista), y retorna un elemento aleatorio de la secuencia (de la lista).

Codificar y ejecutar

```
1. import random
2.
3. numAleatorio = random.randint(1, 10)
4. print(numAleatorio)
5.
6. candidatos = ["Jhair", "Harold", "Jeison", "Juan"]
7. print(random.choice(candidatos))
```

```
2
Harold
```

Figura 17-2. Ejecución del código anterior.

17.3. El índice de paquetes de Python (PyPI)

PyPI es un repositorio que contiene paquetes, librerías o módulos desarrollados y compartidos por la comunidad de Python (https://pypi.org/). Allí hay más de 400.000 proyectos y más de 600.000 usuarios.

Veamos algunos proyectos populares de PyPI que podría utilizar para desarrollar diferentes programas.

- **PyGame:** librería para desarrollar videojuegos.
- **Django:** librería para desarrollar aplicaciones web.
- **Matplotlib:** librería para generación de gráficos.
- **Pillow:** librería para manipular imágenes.

- **NumPy:** librería de cálculo matemático y análisis de datos.

A continuación, veremos cómo utilizar la librería Matplotlib.

17.4. Introducción a Matplotlib

Matplotlib es una librería que permite generar gráficos estáticos, animados e interactivos en Python. Muchos de estos gráficos se generan a partir de datos contenidos en listas o diccionarios (https://matplotlib.org/). Matplotlib hace parte de PyPI (https://pypi.org/project/matplotlib/) y fue creada 2003 por John D. Hunter. Aquí podemos observar cómo la comunidad de Python realiza desarrollos que en muchos casos tienen un impacto global.

Dado que Matplotlib no hace parte de la "La Librería Estándar de Python", significa que, para poder utilizarla, deberíamos instalarla. Sin embargo, debido a su popularidad, Colab incluye Matplotlib (sin necesidad de instalarla).

En esta sección, analizaremos uno de los módulos más populares de Matplotlib llamado *PyPlot* y algunas de sus funciones más populares.

PyPlot con función *plot*

PyPlot es un módulo de Matplotlib que propone varias funciones para añadir elementos tales como líneas, imágenes o textos a los ejes de un gráfico. En esta sección analizaremos la función *plot*.

La función ***plot*** recibe dos argumentos (dos listas *x* y *y*), y dibuja un polígono con los vértices dados por las coordenadas de la lista *x* en el eje X (horizontal), y las coordenadas de la lista *y* en el eje Y (vertical). El siguiente código muestra cómo utilizar la función *plot* (ver ejecución en Figura 17-3).

- En la línea 1 importamos el módulo *PyPlot* de *matplotlib.pyplot*, además le asignamos el alias *plt* (mediante el uso de la palabra reservada de Python *as*). El alias es un nombre corto que nos permitirá utilizar el módulo directamente con el nombre *plt*, y no con la versión original extensa *matplotlib.pyplot*.
- En la línea 4 definimos una lista *x*, donde simulamos los años en los que hemos ahorrado dinero. Esta lista se utilizará como el eje X de nuestra gráfica.
- En la línea 6 definimos una lista *y*, donde simulamos el dinero ahorrado en cada año. Esta lista se utilizará como el eje Y de nuestra gráfica.
- En la línea 8 utilizamos la función *plot* para crear la gráfica, le pasamos como argumentos las listas *x* y *y*.
- En las líneas 9 y 10 definimos los textos de las leyendas que acompañan los ejes X y Y, respectivamente.

- Finalmente, en la línea 11 utilizamos la función *show* para mostrar la gráfica por pantalla.

Codificar y ejecutar

```
1.  import matplotlib.pyplot as plt
2.
3.  # x representa los años
4.  x = [2012, 2013, 2014, 2015, 2016, 2017]
5.  # y representa el dinero ahorrado
6.  y = [0, 50, 70, 70, 100, 246]
7.
8.  plt.plot(x, y)
9.  plt.xlabel("Año")
10. plt.ylabel("Ahorro (dólares)")
11. plt.show()
```

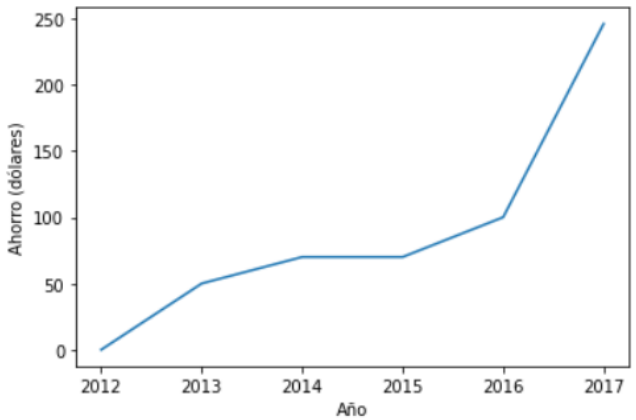


Figura 17-3. Ejecución del código anterior.

PyPlot con función *bar*

La función ***bar*** recibe dos argumentos (dos listas *x* y *y*), y dibuja un diagrama de barras donde *x* es una lista con la posición de las barras en el eje X, por otro lado, *y* es una lista con la altura de las barras en el eje Y. El siguiente código muestra cómo utilizar la función *bar* (ver ejecución en Figura 17-4).

- En la línea 1 importamos el módulo *PyPlot* con su respectivo alias.
- En la línea 4 definimos una lista *x* con los años.

- En la línea 6 definimos una lista *y* con la cantidad de terremotos por año.
- En la línea 8 utilizamos la función *bar* para crear la gráfica de barras, y le pasamos como argumentos las listas *x* y *y*.
- En las líneas 9 y 10 definimos los textos de las leyendas que acompañan los ejes X y Y respectivamente.
- Finalmente, en la línea 11 utilizamos la función *show* para mostrar la gráfica por pantalla.

Codificar y ejecutar

```
1.  import matplotlib.pyplot as plt
2.
3.  # x representa los años
4.  x = [2017, 2018, 2019, 2020, 2021]
5.  # y representa los terremotos
6.  y = [1566, 1808, 1637, 1433, 2206]
7.
8.  plt.bar(x, y)
9.  plt.xlabel("Año")
10. plt.ylabel("Terremotos")
11. plt.show()
```

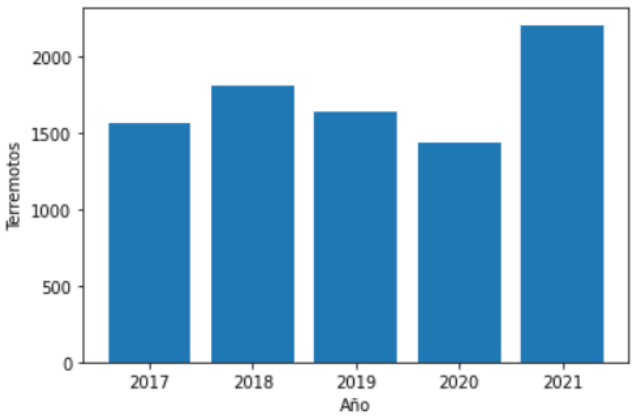


Figura 17-4. Ejecución del código anterior.

TIP: Existen muchos otros módulos y funciones disponibles para trabajar con Matplotlib. Por lo tanto, le recomendamos dar un vistazo rápido a este enlace: https://matplotlib.org/stable/gallery/index. Allí podrá encontrar múltiples ejemplos de gráficos que podría realizar con esta librería. Si lo prefiere, puede traducir la página a español.

17.5. Ejercicios

Ejercicios teóricos

E17.1. ¿Qué palabra reservada de Python se utiliza cuando deseamos importar una librería o módulo?

E17.2. ¿Qué palabra reservada de Python se utiliza cuando queremos darle un alias a un módulo o librería importado?

E17.3. ¿Para qué sirve Matplotlib?

Ejercicios prácticos

E17.4. Cree el código necesario para generar la siguiente gráfica por pantalla (ver Figura 17-5).

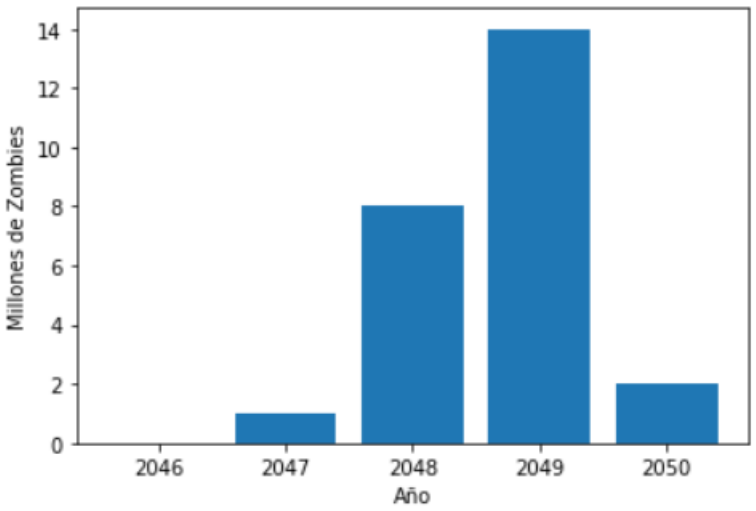


Figura 17-5. Gráfico año contra millones de Zombies.

Resumen

En este capítulo aprendimos cómo funcionan las librerías en Python. Aprendimos que Python incorpora lo que se denomina “La Librería Estándar de Python” (que incluye módulos con funcionalidades útiles que se pueden reutilizar). Aprendimos a utilizar los módulos *math* y *random* de la librería estándar. Aprendimos sobre el índice de paquetes de Python llamado PyPI, donde la comunidad de Python desarrolla y comparte proyectos o librerías para ser utilizados por cualquier persona. Y aprendimos a utilizar algunas funciones del módulo PyPlot de la librería Matplotlib.

Fue un largo recorrido donde aprendimos múltiples elementos sobre programación y sobre Python. Pero antes de terminar, veamos en el siguiente capítulo cómo podemos continuar nuestro viaje de aprendizaje en el mundo de Python y de la programación.

Capítulo 18 – Continúe su aprendizaje

Hemos aprendido demasiadas cosas desde que iniciamos este libro. Tomamos un enfoque práctico para diseñar e implementar cientos de pequeños programas con Python, y aprendimos múltiples conceptos y elementos. Sin embargo, existen muchos elementos que no desarrollamos en este libro.

En este capítulo presentaremos algunos recursos adicionales que podría utilizar para aprender nuevos elementos de manera independiente, y practicar sus habilidades de programación.

A continuación, desarrollaremos las siguientes secciones:

1. Temas adicionales.
2. Plataformas para realizar ejercicios.
3. Observaciones finales y libros futuros.

18.1. Temas adicionales

Existen muchos otros temas que quedaron por fuera del alcance de este libro. A continuación, presentamos algunos de ellos y algunos enlaces donde podría obtener más información.

- **Tupla (tuple):** es un tipo de dato secuencia (similar a las listas), pero una vez definida una tupla no se puede modificar, mientras que las listas si se pueden modificar (https://docs.python.org/es/3/tutorial/datastructures.html#tuples-and-sequences).
- **Conjunto (set):** es un tipo de dato secuencia (similar a las listas), pero no puede contener elementos repetidos (https://docs.python.org/es/3/tutorial/datastructures.html#sets).
- **Gestión de excepciones (errores):** permiten modificar la manera en la que se presentan los errores al usuario final (https://docs.python.org/es/3/tutorial/errors.html#handling-exceptions).
- **Clases (programación orientada a objetos):** permiten definir nuestros propios tipos de datos (https://docs.python.org/es/3/tutorial/classes.html).

18.2. Plataformas para realizar ejercicios

Si le gustaron los ejercicios planteados en este libro, podría utilizar los siguientes recursos gratuitos como complemento.

CodingBat

CodingBat es un sitio web para programar online y solucionar múltiples ejercicios en Python y Java (https://codingbat.com/python). Un usuario puede registrarse y llevar el progreso de los ejercicios completados o los puede programar sin necesidad de registrarse.

Todos los ejercicios de Python se deben programar como funciones. La Figura 18-1 muestra cómo solucionar uno de los ejercicios (https://codingbat.com/prob/p141905). Cada ejercicio es verificado con la ejecución de unos casos de prueba, al lado derecho de la Figura 18-1 se presenta un ejemplo de los casos de prueba del ejercicio, y si todos los casos pasan, el ejercicio se completa.

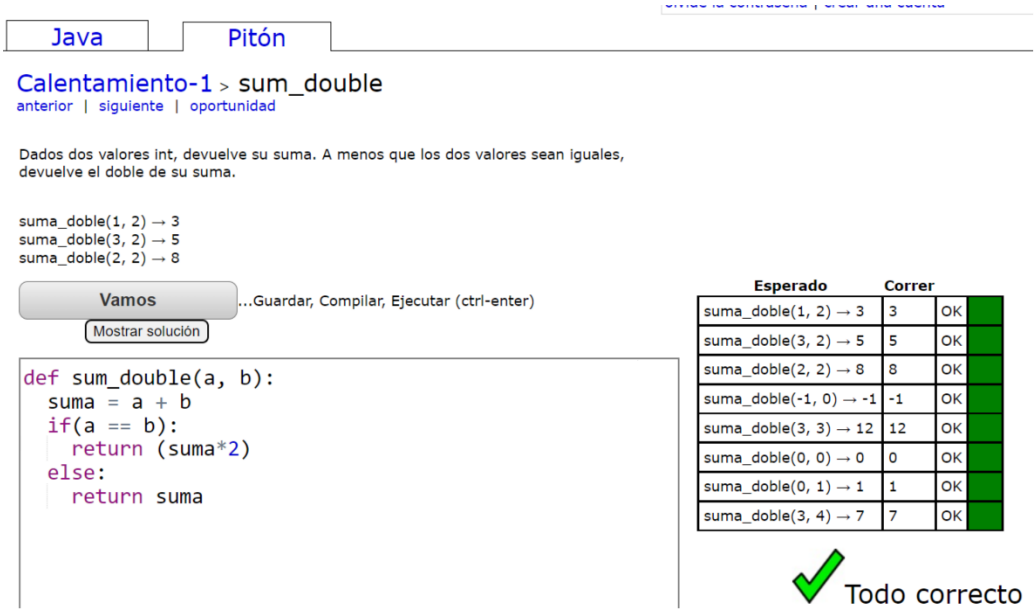


Figura 18-1. Ejercicio en CodingBat (sitio web parcialmente traducido al español).

Codewars

Codewars es una plataforma similar a CodingBat, permite programar online y solucionar múltiples ejercicios en múltiples lenguajes de programación (https://www.codewars.com/). Esta plataforma permite a sus usuarios crear sus propios ejercicios y compartirlos con la comunidad. Algunos de estos ejercicios se pueden acceder y ejecutar sin necesidad de registrarse, pero la gran mayoría requieren que el usuario esté registrado.

La Figura 18-2 muestra cómo solucionar uno de sus ejercicios utilizando funciones (https://www.codewars.com/kata/56bc28ad5bdaeb48760009b0/train/python?collection=pyth

on-ders). Cada ejercicio es verificado con la ejecución de unos casos de prueba, y si todos los casos pasan, el ejercicio se completa.

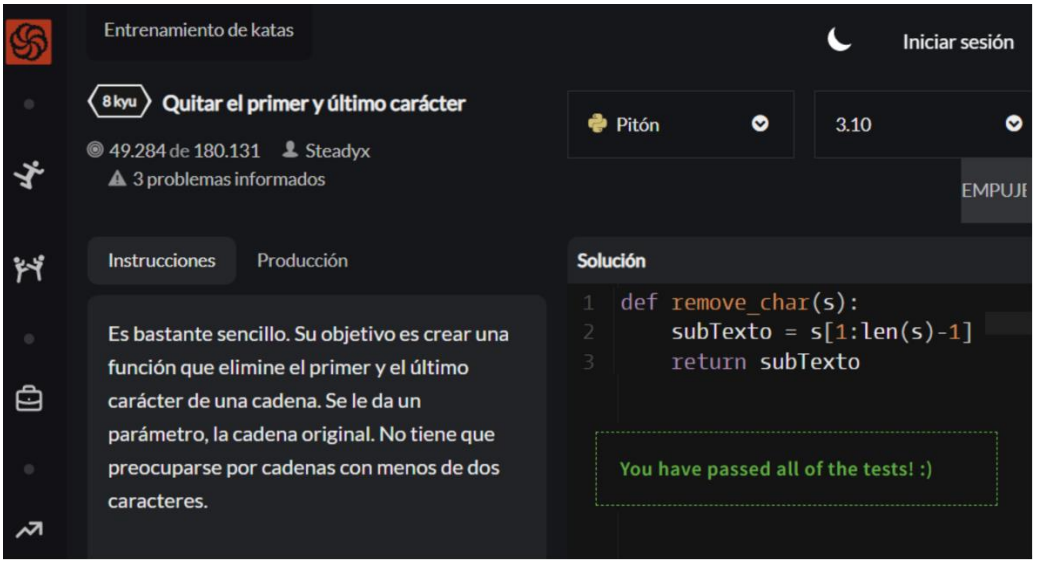


Figura 18-2. Ejercicio en Codewars (sitio web parcialmente traducido al español).

Otras plataformas

Adicional a los recursos anteriores, podría buscar en Google o utilizar plataformas tales como: freeCodeCamp, Codecademy, Code.org o HackerRank.

18.3. Observaciones finales y libros futuros

Los autores de este libro también son autores de otros libros de programación. Daniel, por ejemplo, es autor de otros seis libros sobre programación web (aunque todos en inglés). Uno de esos libros se llama “Django 4 for the Impatient” (publicado en 2022), el cual enseña cómo programar aplicaciones web con Python. Si desea, puede darle un vistazo en el siguiente enlace: https://www.amazon.com/gp/product/B0B25BLDV4/.

Adicionalmente, los autores de este libro trabajan constantemente en desarrollar nuevos libros. Si quiere mantenerse informado, siga la cuenta de Twitter de Daniel (**@danielgarax**).

Capítulos extra gratis

Para mantener un libro de corta longitud, desarrollamos un capítulo extra (sobre tuplas y conjuntos) y otro sobre un proyecto de una biblioteca. Si desea obtenerlos, escríbanos a practicalbooksco@gmail.com y se los enviaremos a su correo.

Comentarios

A los autores de libros nos encanta recibir comentarios de parte los lectores. Si lo desea, escribanos a practicalbooksco@gmail.com y déjenos saber lo que le gustó y lo que no le gustó de este libro. Tomamos muy en serio las opiniones de nuestros lectores, y las utilizaremos como base para el desarrollo de versiones futuras de este libro o de otros libros.

Reseña del libro

Por último, antes de terminar este libro, **le pedimos el favor de dejar una reseña honesta en Amazon**. Esto es vital para que lectores potenciales conozcan la opinión imparcial de personas que ya han leído el libro, y basados en esa información adquieran este libro. Este proceso solo le tomará unos minutos de su tiempo, pero es muy valioso para nosotros.

Y si Amazon no le permite enviar una reseña, valoramos bastante cualquier mención en redes sociales sobre el libro.

"Hecho en Medellín" – "Hecho con Amor"

Capítulo 19 – Solución de ejercicios

En este capítulo presentaremos la solución de los ejercicios teóricos y prácticos que se propusieron en algunos de los capítulos del libro. Estas soluciones no siempre coincidirán exactamente con las soluciones que usted desarrolle, ya que existen diferentes formas para solucionar un mismo ejercicio de programación.

Puede encontrar el código desarrollado en este capítulo en el siguiente enlace del repositorio del libro: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo19.

19.1. Solución ejercicios del capítulo 05

E5.1. (i) *totalDeuda* es *float*, (ii) *nombreProducto* es *str*, (iii) *lunas* es *int* y (iv) *motores* es *str*.

E5.2. *print* sirve para imprimir información en pantalla.

E5.3. En la línea 1 porque el nombre de la variable no puede contener un espacio en blanco.

E5.4.

Solución

```
1. nombre = "Maria"
2. apellido = "Perez"
3. edad = 25
4. estatura = 1.70
5.
6. print(nombre)
7. print(apellido)
8. print(edad)
9. print(estatura)
```

19.2. Solución ejercicios del capítulo 06

E6.1. (i) *edadPadre* es *int*, (ii) *edadMadre* es *str* y (iii) *edadModificada* es *int*.

E6.2. *input* sirve para recibir datos de entrada.

E6.3. Función *str*.

E6.4.

Solución

```
1.  nombre = input("Ingrese el nombre: ")
2.  apellido = input("Ingrese el apellido: ")
3.  edad = int(input("Ingrese la edad: "))
4.  estatura = float(input("Ingrese la estatura: "))
5.
6.  print(nombre)
7.  print(type(nombre))
8.  print(apellido)
9.  print(type(apellido))
10. print(edad)
11. print(type(edad))
12. print(estatura)
13. print(type(estatura))
```

E6.5.

Solución

```
1.  nombre = input("Ingrese el nombre: ")
2.  apellido = input("Ingrese el apellido: ")
3.  print(apellido + " " + nombre)
```

E6.6.

Solución

```
1.  anios = float(input("Ingrese el número de años: "))
2.  mitadAnios = anios/2
3.  print("El asteroide caerá en: " + str(mitadAnios)+ " años")
```

19.3. Solución ejercicios del capítulo 07

E7.1. Se ejecutan las líneas: 1, 2 y 4.

E7.2. El operador lógico *and* sirve para comparar variables o expresiones booleanas. Este operador únicamente evalúa a *True* si ambos valores o expresiones son *True*.

E7.3.

Expresión	Resultado
(carga == 200) *and* (tipo != "Camión")	True
(carga > 350) *or* (tipo == "Tractor")	True
(carga <= 600) *and* (tipo != "Tractor")	False
not(tipo == "Camión")	True

E7.4.

Solución

```
1. distancia = float(input("Ingrese la distancia: "))
2. if(distancia <= 4):
3.   print("Ya voy corriendo")
```

E7.5.

Solución

```
1. jugador = int(input("Ingrese el número de la camiseta: "))
2. if(jugador == 19):
3.   print("Andá pa allá, bobo")
```

E7.6.

Solución

```
1. magnitud = float(input("Ingrese la magnitud: "))
2. if(magnitud >= 5.2):
3.   print("Alerta de tsunami")
```

19.4. Solución ejercicios del capítulo 08

E8.1. Se ejecutan las líneas: 1, 2, 4, 5 y 6.

E8.2. Verdadero.

E8.3. *¿Qué es esooo? - Fin programa*

E8.4.

Solución

```
1.  generoMusical = input("ingrese el género musical: ")
2.  if(generoMusical == "Rock"):
3.    print("Tienes buen gusto")
4.  else:
5.    print("Qué asco")
```

E8.5.

Solución

```
1.  nombre1 = input("Ingrese el nombre del candidato 1: ")
2.  votos1 = input("Ingrese los votos del candidato 1: ")
3.  nombre2 = input("Ingrese el nombre del candidato 2: ")
4.  votos2 = input("Ingrese los votos del candidato 2: ")
5.
6.  if(votos1 > votos2):
7.    print(nombre1)
8.  elif(votos2 > votos1):
9.    print(nombre2)
10. else:
11.   print("Empate")
```

19.5. Solución ejercicios del capítulo 09

E9.1. Se ejecutan las líneas: 1, 2, 3, 4, 2, 3, 4, 2 y 5.

E9.2. Son: (i) inicialización de la variable de control, (ii) prueba sobre la variable de control, (iii) cuerpo del ciclo, (iii) cuerpo del ciclo y (iv) cambio sobre la variable de control.

E9.3. *0 - 3 - 6*

E9.4.

Solución

```
1.  numero = int(input("Ingrese un número: "))
2.  i = 0
3.  while(i < numero):
4.    print("****")
5.    i = i+1
```

E9.5.

Solución

```
1.  while(True):
2.    palabra = input("Ingrese una palabra: ")
3.    if(palabra != "Roberto"):
4.      print("Clave incorrecta")
5.    else:
6.      print("Clave correcta")
7.      break
```

19.6. Solución ejercicios del capítulo 10

E10.1. *G - Gat - 3*

E10.2. El segundo argumento del método *find* sirve para especificar la posición a partir de la cual queremos iniciar la búsqueda de un carácter o de un subtexto.

E10.3. *o*

E10.4.

Solución

```
1. texto = input("Ingrese un texto: ")
2. primero = texto[0]
3. ultimo = texto[len(texto)-1]
4. print(primero + ultimo)
```

E10.5.

Solución

```
1. texto = input("Ingrese un texto: ")
2. posZ = texto.find("z")
3. if(posZ != -1):
4.   print("Texto contiene letra z")
5. else:
6.   print("Texto no contiene letra z")
```

E10.6.

Solución

```
1. palabra = input("Ingrese una palabra: ")
2. i = 0
3. while(i < len(palabra)):
4.   if(palabra[i] != "a" and palabra[i] != "e" and palabra[i] != "i" and palabra[i] != "o" and
   palabra[i] != "u"):
5.     print(palabra[i])
6.   i = i+1
```

19.7. Solución ejercicios del capítulo 11

E11.1. *3 - Azul*

E11.2. *colores.pop(1)*

E11.3. Error porque *len(colores)* es *3* y la lista solo llega hasta la posición 2.

E11.4.

Solución

```
1.  pelicula1 = input("Ingrese el nombre la primera pelicula favorita: ")
2.  pelicula2 = input("Ingrese el nombre la segunda pelicula favorita: ")
3.  pelicula3 = input("Ingrese el nombre la tercera pelicula favorita: ")
4.  pelicula4 = input("Ingrese el nombre la cuarta pelicula favorita: ")
5.
6.  peliculas = []
7.  peliculas.append(pelicula1)
8.  peliculas.append(pelicula2)
9.  peliculas.append(pelicula3)
10. peliculas.append(pelicula4)
11.
12. print(peliculas)
```

E11.5.

Solución

```
1.  mensajeSecreto = ["Planeta", "Más", "Secreto", "Alienigena", "Zombie", "Serás", "Nuclear",
    "Vos"]
2.
3.  i = 1
4.  while(i < len(mensajeSecreto)):
5.    print(mensajeSecreto[i])
6.    i = i+2
```

19.8. Solución ejercicios del capítulo 12

E12.1. *Mago - float*

E12.2. Utilizamos un diccionario en lugar de una lista cuando: (i) queremos almacenar variables de diferentes tipos, (ii) queremos identificar a las posiciones con una palabra o (iii) el orden en el que se almacenan los elementos no es importante.

E12.3. (i) *dict*, (ii) *int*, (iii) *2* y (iv) Ningún valor, no existe la clave *"marca"*.

E12.4.

Solución

```
1.  nombre = input("Ingrese nombre de mascota: ")
2.  edad = int(input("Ingrese edad de mascota: "))
3.  raza = input("Ingrese raza de mascota: ")
4.
5.  mascota = {
6.    "nombre":nombre,
7.    "edad":edad,
8.    "raza":raza
9.  }
10.
11. print(mascota)
```

E12.5.

Solución

```
numero = int(input("Ingrese numero: "))
equipos = []

i = 1
while(i<=numero):
  nombre = input("Ingrese nombre del equipo: ")
  pais = input("Ingrese país de origen: ")
  equipo = {"nombre":nombre, "pais":pais}
  equipos.append(equipo)
  i=i+1

print(equipos)
```

19.9. Solución ejercicios del capítulo 13

E13.1. *Rocas - Rocas - Rocas - Rocas*

E13.2. Los tres argumentos que puede recibir la función *range* son: inicio, parada y paso.

E13.3. Se ejecutan las líneas: 1, 2, 3, 2, 3, 2 y 3

E13.4.i.

Solución

```
for i in range(1,81,1):
  print(i)
```

E13.4.ii.

Solución

```
1.  for i in range(1000,-1,-100):
2.    print(i)
```

E13.4.iii.

Solución

```
1.  for i in range(2,67,2):
2.    print(i)
```

E13.5.

Solución

```
1.  nombre1 = input("Ingrese nombre 1: ")
2.  nombre2 = input("Ingrese nombre 2: ")
3.  nombre3 = input("Ingrese nombre 3: ")
4.  nombre4 = input("Ingrese nombre 4: ")
5.
6.  nombres = [nombre1, nombre2, nombre3, nombre4]
7.
8.  for nombre in nombres:
9.    if(nombre[0] == "L"):
10.     print("Nombre descartado")
11.   else:
12.     print("Nombre posible")
```

19.10. Solución ejercicios del capítulo 14

E14.1. (i) *mensajeRaro*, (ii) 1, (iii) no, (iv) 1 y (v) *holaEdificiohola*

E14.2. *return*

E14.3. Se ejecutan las líneas: 5, 1, 2, 6, 1, 2 y 3

E14.4.

Solución

```
1.  def primerNombre(nombres):
2.    print(nombres[0])
3.
4.  nombres = ["Zeus", "Poseidon", "Ares"]
5.  primerNombre(nombres)
```

E14.5. Dada la longitud de este ejercicio, la solución la puede encontrar en: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo19.

19.11. Solución ejercicios del capítulo 15

E15.1. Un contador se utiliza para contar cosas, normalmente se cuenta de 1 en 1. Mientras que un acumulador se utiliza para acumular valores, y no siempre se suma la misma cantidad.

E15.2. Se ejecutan las líneas: 1, 3, 4, 6, 3, 4 y 5

E15.3.

Solución

```
1.  a1 = {"nombre":"2022 HX1", "distancia":2490000}
2.  a2 = {"nombre":"2022 HA2", "distancia":4620000}
3.  a3 = {"nombre":"101955 Bennu", "distancia":32490000}
4.  asteroides = [a1, a2, a3]
5.
6.  contador = 0
7.  for asteroide in asteroides:
8.    if(asteroide["distancia"] < 7500000):
9.      contador = contador+1
10.
11. print("Asteroides con alta probabilidad de colisión: "+str(contador))
```

E15.4.

Solución

```
1.  a1 = {"nombre":"2022 HX1", "distancia":2490000}
2.  a2 = {"nombre":"2022 HA2", "distancia":4620000}
3.  a3 = {"nombre":"101955 Bennu", "distancia":32490000}
4.  asteroides = [a1, a2, a3]
5.
6.  acumulador = 0
7.  for asteroide in asteroides:
8.    acumulador = acumulador+asteroide["distancia"]
9.
10. promedio = acumulador/len(asteroides)
11. print("Promedio de distancia de asteroides: "+str(promedio))
```

19.12. Solución ejercicios del capítulo 16

E16.1. El método *read* sirve para leer todo el contenido de un archivo.

E16.2. El método *readlines* sirve para retornar una lista en la que cada posición representa una línea del archivo que estamos leyendo.

E16.3. Dada la longitud de este ejercicio, la solución la puede encontrar en: https://github.com/PracticalBooks/Python-Para-Principiantes/tree/main/Capitulo19.

19.13. Solución ejercicios del capítulo 17

E17.1. Para importar una librería o módulo utilizamos la palabra *import*

E17.2. Para asignarle un alias a una librería o módulo importado utilizamos la palabra *as*

E17.3. Matplotlib sirve para generar gráficos estáticos, animados e interactivos en Python

E17.4.

Solución

```
1.  import matplotlib.pyplot as plt
2.
3.  x = [2046, 2047, 2048, 2049, 2050]
4.  y = [0, 1, 8, 14, 2]
5.
6.  plt.bar(x, y)
7.  plt.xlabel("Año")
8.  plt.ylabel("Millones de Zombies")
9.  plt.show()
```