Pensamiento Computacional

Aldana Rastrelli, Juan Pablo Bulacios, Llamell Martínez Gorbik, Pablo Notari

2023-12-27

Table of contents

Pensamiento Computacional

Bienvenidos y bienvenidas a la cátedra de Pensamiento Computacional del Ciclo Básico Común de la Facultad de Ingeniería - UBA.

Docentes de la Cátedra

- **Prof. Titular:** Méndez, Mariano
- Areco, Lucas
- Balbiano, Jose Luis
- Balbiano, Jose Luis
- Bulacios, Juan
- Cabibbo Arteaga, Nehuén Daniel
- Cáceres, Fernando
- Capón, Lucía
- Carletti, Joaquin
- Corti, Bautista
- Duchen, Leonardo
- Duzac, Emilia
- Juarez Goldemberg, Mariana
- Lopez, Fernando
- Lourengo Caridade, Lucía Gabriela
- Maciel, Laura
- Maxwell, Julian
- Notari, Pablo
- Ortielli, Bruno

- Pratto, Florencia
- Rastrelli, Aldana
- Retamozo, Melina
- Szischik, Mariana

La Materia

Fundamentación

El pensamiento computacional es una disciplina que ha sido definida como "el conjunto de procesos de pensamiento implicados en la formulación de problemas y sus soluciones, de manera que dichas soluciones sean representadas de una forma que puedan ser efectivamente ejecutadas por un agente de procesamiento de información", entendiendo por esto último a un humano, una máquina o una combinación de ambos.

Reconoce antecedentes en trabajos de la Carnegie Mellon University de la década de 1960 y del Massachusetts Institute of Technology de alrededor de 1980, aunque su auge en la educación superior llegó con la primera década del siglo XXI.

Las herramientas básicas en las que se funda el pensamiento computacional son la descomposición, la abstracción, el reconocimiento de patrones y la algoritmia. Está ampliamente aceptado que estas herramientas no sirven solamente a los profesionales de Ciencias de la Computación y de Informática, sino a cualquier persona que deba resolver problemas, con lo cual el pensamiento computacional deviene una técnica de resolución de problemas. Actualmente, los y las profesionales de la Ingeniería requieren de una capacidad analítica que les permita resolver problemas, y en ese sentido el pensamiento computacional se convierte en un soporte invaluable de esa competencia (cada vez más las ciencias de la computación y la informática constituyen una ciencia básica para todas las ingenierías).

Si bien el pensamiento computacional no necesariamente requiere del uso de computadoras, la programación de computadoras se convierte en su complemento ideal. En primer lugar, porque permite comprobar, mediante la codificación de un algoritmo en un programa, la validez de la solución encontrada al problema, de manera sencilla y prácticamente inmediata. En segundo lugar, porque la programación incentiva la creatividad, la capacidad para la auto organización y el trabajo en equipo. En tercer lugar, porque la programación constituye un recurso habitual del trabajo en el campo profesional de la ingeniería.

Objetivos Generales

El objetivo general de la asignatura es que los/as estudiantes adquieran habilidades de resolución de problemas de ingeniería mediante el soporte de un lenguaje de programación multi-

paradigma.

Regimen de Cursada, Calendario y Cursos

Formas de Evaluación

La cursada de la materia cuenta con dos parciales:

- Primer Parcial
  - Unidad 1
  - Unidad 2
  - Unidad 3
  - Unidad 4
- Segundo Parcial
  - Unidad 5
  - Unidad 6

Cada parcial cuenta con un único recuperatorio.

Aprobación de la Cursada/Materia

Se tiene dos formas de aprobación de la cursada:

1. Regularización
2. Promoción

Regularización

Para regularizar la cursada, se deben aprobar los dos parciales (o recuperatorios) con un mínimo de nota de 4 (cuatro) en cada uno.

La cursada regularizada habilita a rendir el examen final integrador, para el cual se tienen 3 (tres) oportunidades de rendir (más información abajo).

Promoción

Para promocionar la materia, se debe tener un promedio entre los dos parciales (o recuperatorios) de 7 (siete).

> **Rendir Recuperatorios para Promoción**
>
> Si se desea rendir el recuperatorio para intentar subir la nota para la promoción, se debe tener en cuenta que la cátedra considerará únicamente válida la nota del último examen que se haya rendido.
> Ejemplo:
>
> ```
> # caso 1
> parcial1 = 5
> recuperatorio1 = 7
> => nota final parcial1 = 7
>
> # caso 2
> parcial1 = 5
> recuperatorio1 = 4
> => nota final parcial1 = 4
> ```

Examen Final Integrador

El examen final integrador consta de una evaluación que incluye todos los temas de la materia. Los mismos se rinden al final del cuatrimestre. Se aprueba con una nota mayor o igual a 4 (cuatro).

> **Desaprobación de la Materia**
>
> Si se desaprueba alguno de los parciales, el mismo puede recuperarse una sola vez.
> Si se desaprueba un recuperatorio, se debe volver a cursar la materia el cuatrimestre siguiente.
> Si se desaprueba 3 (tres) veces el examen final integrador, se debe volver a cursar la materia el cuatrimestre siguiente.

Calendario y Cursos

Se puede acceder al calendario de la cursada y a las aulas y horarios de los cursos a través del siguiente link.

Videos Teóricos y Diapositivas

La parte teórica de la materia incluye ver los videos y leer los apuntes que se encuentran en esta página. Los apuntes contienen información que se tendrá en cuenta al momento de evaluar la materia en los parciales.

Tanto videos teóricos como diapositivas usadas en la práctica se encuentran en siguiente link.

Las clases teóricas son virtuales y asincrónicas. Los videos se encuentran en el link de arriba, en la carpeta titulada “Teóricas”.

Las clases prácticas son presenciales. Las diapositivas usadas se encuentran en el link de arriba, en la carpeta titulada “Prácticas”. Las mismas están organizadas por curso.

1 Introducción a la Algoritmia y a la Programación

1.1 Introducción

Como en todas las disciplinas, la Ingeniería de Software y la Programación de Sistemas en general tienen un **lenguaje técnico** específico. La utilización de ciertos términos y el compartir de ciertos conceptos agiliza el diálogo y mejora la comprensión con los pares.

En este capítulo vamos a hacer una breve introducción de ciertos conceptos, ideas y modelos que van a permitirnos establecer acuerdos y manejar un lenguaje común.

1.1.1 La Computadora

Una computadora es un dispositivo físico de procesamiento de datos, con un propósito general. Todos los programas que escribiremos serán ejecutados (o *corridos*) en una computadora. Una computadora es capaz de procesar datos y obtener nueva información o resultados.

1.1.2 Software y Hardware

Toda computadora funciona con software y hardware. El software es el conjunto de herramientas abstractas (programas), y se le llama **componente lógica** del modelo computacional. El hardware es el **componente físico** del dispositivo. Básicamente, el software dice qué hacer, y el hardware lo hace.

¿Es indispensable tener una computadora para crear un algoritmo?

La respuesta, sorprendentemente, es no: muchos de los algoritmos que se utilizan de forma computacional hoy en día fueron diseñados varias décadas atrás. Pero la implementación de un algoritmo depende del grado de avance del hardware y la tecnología disponible.

1.1.3 Sistema Operativo

El sistema operativo es el programa encargado de administrar los recursos del sistema. Los recursos (como la memoria, por ejemplo) son disputados entre diferentes programas o procesos ejecutándose al mismo tiempo. El sistema operativo es el que decide cómo administrar y asignar los recursos disponibles.

Los sistemas operativos más comunes el día de hoy son: Windows, Linux, iOS, Android; por ejemplo.

1.1.4 Algoritmo

Un algoritmo es una serie finita de pasos precisos para alcanzar un objetivo.

- "serie": porque son continuados uno detrás del otro, de forma ordenada.
- "finita": porque no pueden ser pasos infinitos, en algún momento deben terminar.
- "pasos precisos": porque en un algoritmo se debe ser lo más específico posible.

  Ejemplo Un algoritmo puede ser una receta de cocina: tiene una serie finita de pasos (son ordenados, uno detrás de otro, finitos porque en algún momento deben terminar), que son precisos (porque tienen indicaciones de cuánto agregar de cada ingrediente, cómo incorporarlo a la preparación, etc) y están orientados en alcanzar un objetivo (obtener una comida en particular).

1.1.4.1 Creación de un Algoritmo

La forma en la que trabajaremos la creación de un algoritmo es siguiendo los siguientes pasos:

1. Análisis del problema: entender el objetivo y los posibles casos puntuales del mismo.

2. Primer borrador de solución: confeccionar una idea generalizada de cómo podría resolverse el problema.

3. División del problema en partes: dividir el problema en partes ayuda a descomponer un problema complejo en varios más sencillos.

4. Ensamble de las partes para la versión final del algoritmo: acoplar todo el conjunto de partes del problema para lograr el objetivo general.

Estos cuatro pasos podrán iterarse (repetirse) la cantidad de veces que sean necesarios, para poder lograr acercarnos más a la solución en cada iteración.

1.1.5 Programa

Un programa es un algoritmo escrito en un lenguaje de programación.

1.1.6 Lenguaje de Programación

Un lenguaje de programación es un **protocolo de comunicación**.
Un protocolo es un **conjunto de normas consensuadas**.
$\Longrightarrow$ Entonces, un lenguaje de programación es un conjunto de normas consensuadas, entre la persona y la máquina, para poder comunicarse.

Cuando logramos que un *lenguaje* pueda ser comprendido por el humano y por la máquina, tenemos una comunicación efectiva en donde podremos hacer programas y pedirle a la máquina que los ejecute.

Un buen ejemplo de cómo una computadora interpreta nuestras instrucciones sin pensar al respecto, sin tener sentido común y sin ambigüedades, es este video. La computadora lo único que hace es *interpretar* de forma explícita lo que nosotros le pedimos que haga.

Un lenguaje de programación tiene reglas estrictas que se deben respetar y no se admiten ambiguedades o sobreentendidos.

1.1.7 Entorno de Desarrollo

Un entorno de desarrollo es un conjunto de herramientas que nos permiten escribir, editar, compilar y ejecutar programas.

En la materia utilizaremos un entorno de desarrollo llamado Google Colab, que nos permite escribir código en un editor de texto, compilarlo y ejecutarlo en un mismo lugar de forma online. Pero existen muchos otros entornos de desarrollo, como por ejemplo Visual Studio Code, Eclipse, NetBeans, etc.

1.2 Lenguaje Python

En este curso utilizaremos el lenguaje de programación **Python**. Python es un lenguaje de programación de propósito general, que se utiliza en muchos ámbitos de la industria y la academia.

Python es un lenguaje realmente fácil de aprender, con una curva de aprendizaje muy suave. Es un lenguaje de alto nivel, lo que significa que es un lenguaje que se asemeja mucho al lenguaje natural, y que no requiere de conocimientos de bajo nivel para poder utilizarlo.

1.2.1 Hola, Mundo!

El primer programa que se escribe en cualquier lenguaje de programación es el programa "Hola, Mundo!". Este programa es un programa que imprime en pantalla el texto "Hola, Mundo!".

En Python, el programa "Hola, Mundo!" se escribe de la siguiente forma:

```
print("Hola, Mundo!")
```

```
Hola, Mundo!
```

`print` es una función que imprime en pantalla el texto que se le pasa entre paréntesis. En este caso, el texto que se le pasa como parámetro es `"Hola, Mundo!"`. Al escribir las comillas dobles, estamos indicando que el texto que se encuentra entre ellas es un texto literal.

De la misma forma, podremos imprimir cualquier otro mensaje en pantalla, como por ejemplo:

```
print("Hola, me llamo Rosita y soy programadora")
```

```
Hola, me llamo Rosita y soy programadora
```

Al igual que Rosita, al hacer nuestro primer 'Hola, Mundo!' nos convertimos en programadores. ¡Felicitaciones!

A partir de la próxima clase, comenzaremos a ver cómo escribir programas más complejos, que nos permitan resolver problemas más interesantes.

1.3 Anexo: Google Colab

1.3.1 Cómo usar Google Colab

Para usar Google Colab, debemos ingresar a este link. Si es necesario, debemos crear una cuenta de Google.

Al abrir Google Colab por primera vez, vamos a ver lo siguiente:

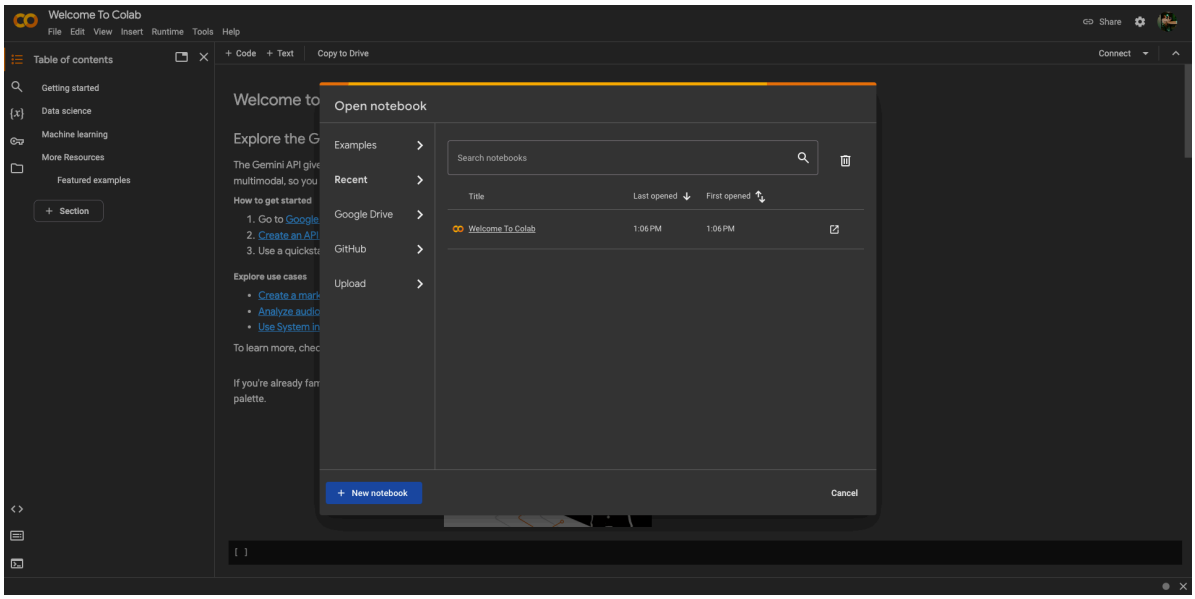


Figure 1.1: Inicio en Google Colab

Vamos entonces a hacer click en “New Notebook”, y se va a abrir un archivo nuevo, con extensión `.ipynb` (que es la extensión de un archivo del tipo IPython Notebook). Vamos a cambiarle el nombre de ‘Untitled0’ a ‘Unidad_1’ o el nombre que prefieran.

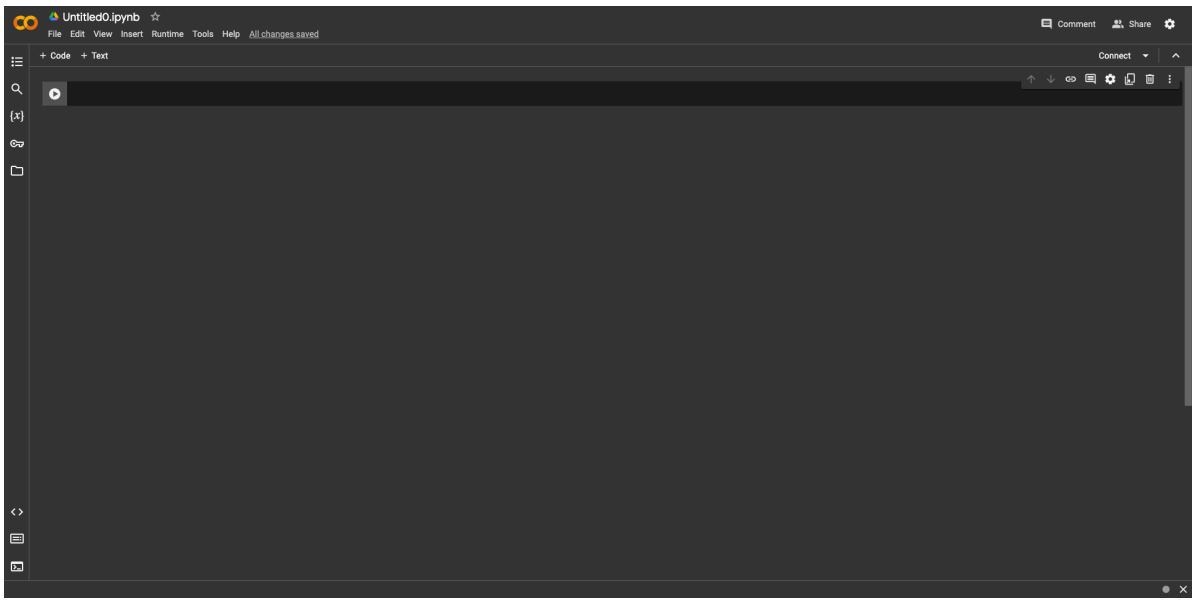


Figure 1.2: Archivo nuevo

1.3.1.1 Celdas de Código

Con Colab vamos a poder correr nuestro código. Colab se divide en celdas individuales: cada celda es un bloque de código que se puede correr por separado. Para agregar una celda nueva, se hace click en el botón de "+ Code" que aparece en la parte superior izquierda de la celda. Para correr la celda, se hace click en el botón de "play" que aparece a la izquierda de la celda. El output (la salida) de la celda va a aparecer debajo de la misma.

> **Cómo usar las celdas**
>
> Les recomendamos que cada ejercicio de la guía esté en una celda separada. A medida que avance la materia vamos a terminar de entender bien por qué.

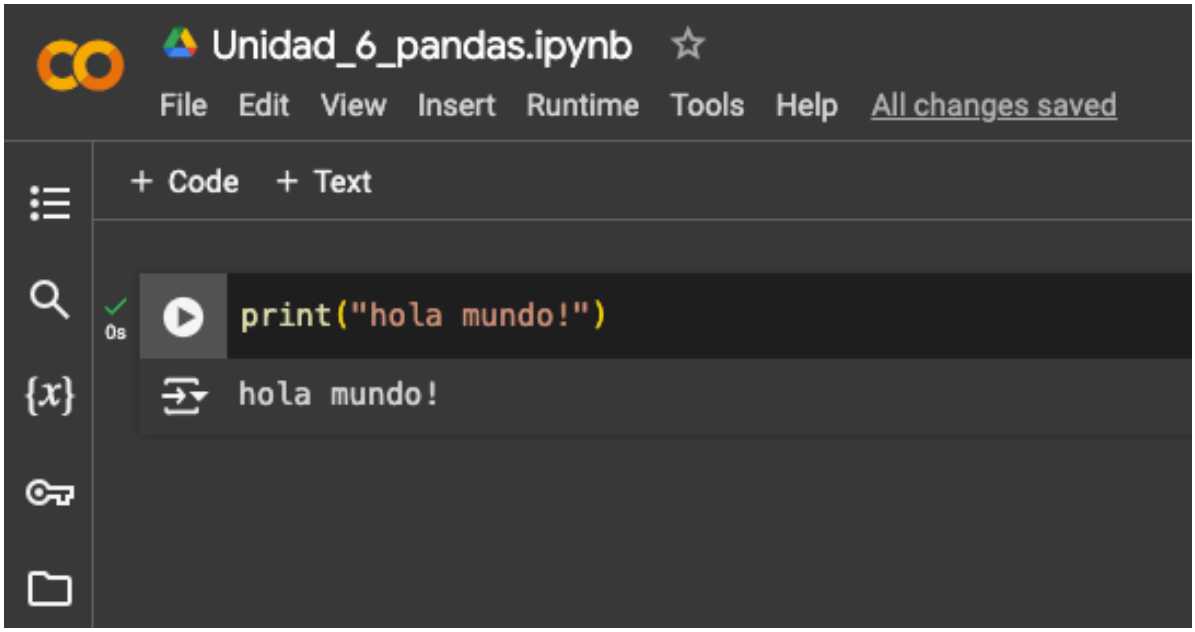


Figure 1.3: Ejecución de una celda de código

Si tenemos varias celdas con código, podemos correrlas todas juntas haciendo click en "Runtime" o "Entorno de Ejecución" en el menú superior y luego en "Run all" (o "Ejecutar Todo"). Cada celda de código va a tener su propio output debajo de ella.

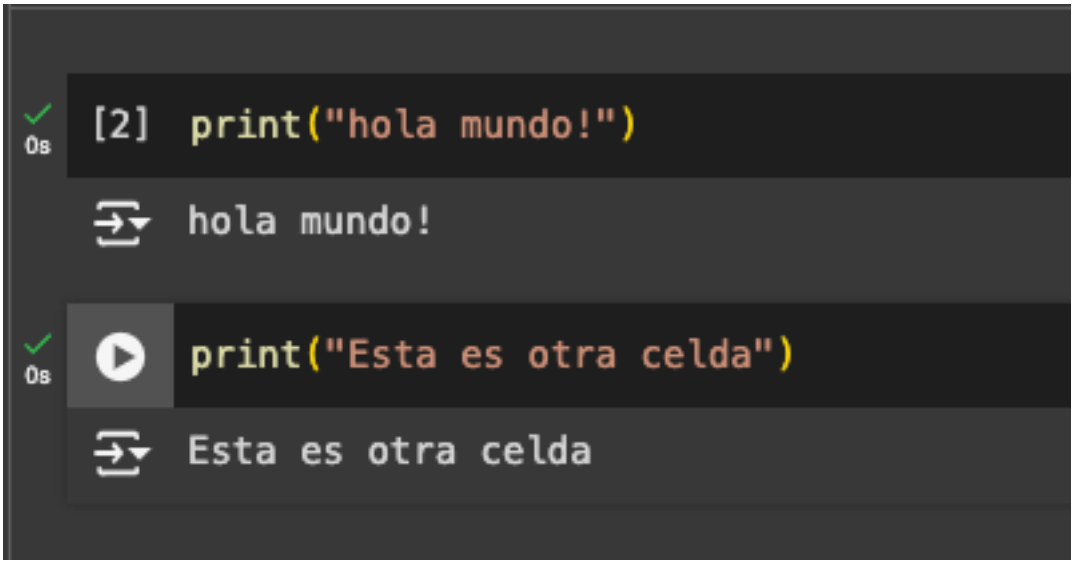
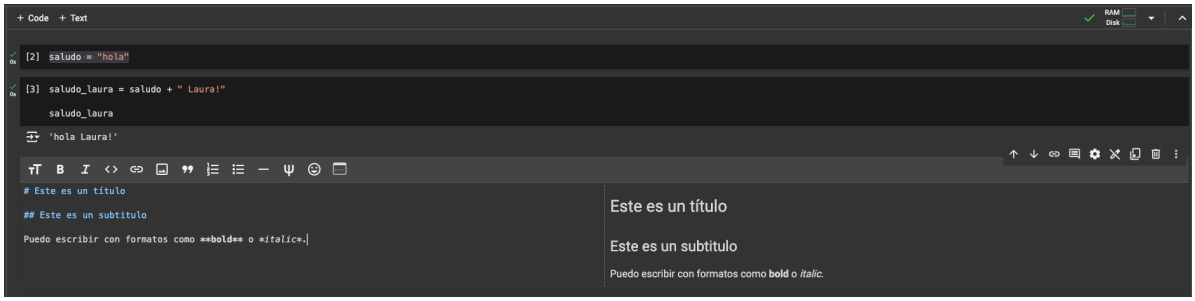


Figure 1.4: Ejecución simultánea de dos celdas de código

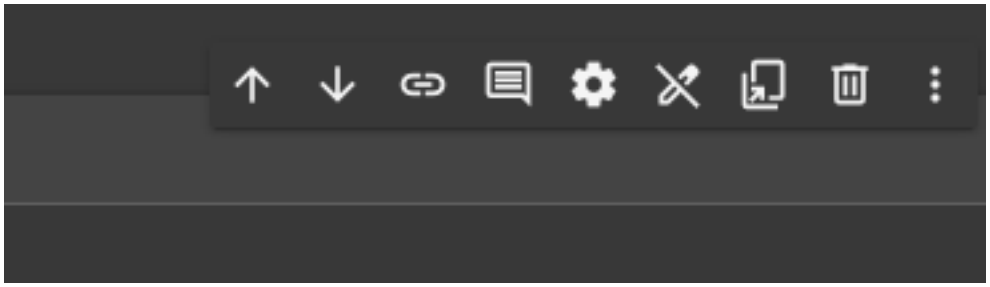
1.3.1.2 Celdas de Texto

Así también como podemos agregar celdas de código, podemos agregar celdas de texto. Para eso, hacemos click en el botón de "+ Text" que aparece en la parte superior izquierda de la celda. Dentro podemos escribir texto con formato e incluso agregar imágenes.



1.3.1.3 Opciones de Celdas

Para reordenar, eliminar o copiar celdas, al seleccionar una celda aparece un menú a la derecha con distintos íconos. Podemos usar estas opciones para realizar estas distintas acciones.



1.3.1.4 Aclaración IA

- Para desactivar las sugerencias con IA (recomendado hacerlo) pueden ir a: Herramientas > Configuración > Asistente IA > Tildar “Ocultar funciones de IA generativa”.

1.3.1.5 Beneficios de usar Google Colab

- Google Colab tiene una mejor interfaz para usar Pandas, mostrando el output en forma de tablas.
- Permite importar datos de Google Drive
- Permite compartir el código con otras personas, de tal forma que todas ellas puedan editar un mismo archivo
- Permite guardar los archivos en Google Drive, con guardado automático de cambios
- Permite exportar el archivo en distintos formatos, como PDF, HTML, etc.
- Permite intercalar código ejecutable con texto explicativo, lo que lo hace ideal para la creación de informes, presentaciones, o tutoriales.

2 Tipos de Datos, Expresiones y Funciones

2.1 Sentencias Básicas

En esta unidad vamos a centrarnos en la herramienta que vamos a emplear, que es Python. Vamos a hacer un programa sencillo, interactuar con el usuario y más.

2.1.1 Flujo de Control de un Programa

El flujo de control de un programa es la forma en la que se ejecutan las instrucciones de un programa. En Python, el flujo de control es secuencial, es decir, se ejecutan las instrucciones una detrás de otra.

Ejemplo:

```
Esta línea se ejecutaría primero        ↓
Esta línea se ejecutaría después        ↓
Esta línea se ejecutaría a lo último
```

En este curso, la comunicación de los programas con el mundo exterior se realizará casi exclusivamente con el usuario por medio de la consola (o terminal, la presentamos en la unidad anterior en el anexo de Colab).

> ⚠ ¡Cuidado!
>
> Esto no significa que todos los programas siempre se comuniquen con el usuario para todo. Pensemos en las aplicaciones que usamos generalmente, como instagram: imaginémonos si para cada acción que hiciéramos dentro de la app la misma nos preguntara si queremos hacerlo o no:
>
> **- "¿Estás seguro/a de que querés iniciar sesión?"**
> **- "¿Estás seguro/a de que querés traer tu nombre de usuario para mostrarse en el perfil?"**
> **- "¿Estás seguro/a de que querés traer tu foto de usuario para mostrarse en el perfil?"**

Sería extremadamente molesto. Uno simplemente inicia sesión, y hay un montón de cosas y procesos que se ejecutan uno detrás de otro, automáticamente.

Hay cosas que no necesitan de la interacción del usuario. Nosotros nos vamos a centrar en la interacción con el usuario en gran parte del curso, pero no es lo único que se puede hacer. Los programas pueden comunicarse con otros programas y las partes de un mismo programa pueden comunicarse con otras partes del mismo programa. Más adelante vamos a ver un poco más de esta diferencia.

2.1.2 Valores y Tipos

Si tenemos la operación `7 * 5`, sabemos que el resultado es `35`. Decimos que tanto `7`, `5` como `35` son *valores*. En los lenguajes de programación, cada valor tiene un *tipo*.

En este caso, `7`, `5` y `35` son *enteros* (o *integers* en inglés). En Python, los enteros se representan con el tipo `int`.

Python tiene dos tipos de datos numéricos:

- número enteros
- números de punto flotante

Los **números enteros** representan un valor entero exacto, como `42`, `0`, `-5` o `10000`.
Los **números de punto flotante** tienen una parte fraccionaria, como `3.14159`, `1.0` o `0.0`.

Según los operandos (los valores que se operan) y el operador (el símbolo que indica la operación), el resultado puede ser de un tipo u otro. Por ejemplo, si tenemos `7 / 5`, el resultado es `1.4`, que es un número de punto flotante. Si tenemos `7 + 5`, el resultado es `12`, que es un número entero.

```
print(1 + 2)
```

```
3
```

> **Note**
>
> `print` es una función de Python que nos deja imprimir cosas por pantalla. Al hacer `print(1+2)`, Python está calculando el resultado de 1+2 e imprimiéndolo para que podamos verlo.

Vamos a elegir usar enteros cada vez que necesitemos recordar, almacenar o representar un valor exacto, como pueden ser por ejemplo: la cantidad de alumnos, cuántas veces repetimos

una operación, un número de documento, etc.
Vamos a elegir usar números de punto flotante cada vez que necesitemos recordar, almacenar o representar un valor aproximado, como pueden ser por ejemplo: la altura o el peso de una persona, la temperatura de un día, una distancia recorrida, etc.

```
print(0.1 + 0.2)
```

```
0.30000000000000004
```

Como vemos, cuando hay números de punto flotante, el resultado es aproximado. `0.1 + 0.2` nos debería dar `0.3`, pero nos da `0.30000000000000004`. Esto es porque los números de punto flotante son aproximados, y no pueden representar todos los valores de forma exacta. Esto es algo que vamos a tener que tener en cuenta cuando trabajemos con números de punto flotante.

> **Uso de punto**
>
> Notemos que para representar números de punto flotante, usamos el punto (`.`) y no la coma (`,`). Esto es porque en Python, la coma se usa para separar valores, como vamos a ver más adelante.

Además de efectuar operaciones matemáticas, Python nos permite trabajar con porciones de texto, que se llaman **cadenas** (o *strings* en inglés). Las cadenas se representan con el tipo `str`.
Las cadenas se escriben entre comillas simples (`'`) o dobles (`"`).

```
print( "¡Hola!" )
```

```
¡Hola!
```

```
print( '¡Hola!' )
```

```
¡Hola!
```

Las cadenas también tienen operaciones disponibles, como por ejemplo la concatenación, que es la unión de dos cadenas en una sola. Esto se hace con el operador `+`.

```
print( "¡Hola!" + " ¿Cómo estás?" )
```

```
¡Hola! ¿Cómo estás?
```

Vamos a ver más de estas operaciones más adelante.

2.1.3 Variables

Python nos permite asignarle un nombre a un valor, de forma tal que podamos "recordarlo" y usarlo más adelante. A esto se le llama **asignación**.
Estos nombres se llaman **variables**, y son espacios de memoria donde podemos almacenar valores.

La asignación se hace con el operador `=` de la siguiente forma: `<nombre> = <valor o expresion>`.

Ejemplos: Vamos a guardar el valor 5 en la variable `x`. Luego, vamos a sumarle 2 y guardarlo en la variable `y`.

```
x = 5
```

```
y = x + 2
```

```
print(y)
```

```
7
```

```
print(y * 2)
```

```
14
```

```
lenguaje = "Python"

texto = "Estoy programando en " + lenguaje
print(texto)
```

```
Estoy programando en Python
```

En este ejemplo, creamos las siguientes variables:

- x
- y
- lenguaje
- texto

y las asociamos a los valores 5, 7, "Python" y "Estoy programando en Python" respectivamente. Luego podemos usar esas variables como parte de cualquier expresión, y en el momento de evaluarla, Python reemplazará las variables por su valor asociado.

Variables y Constantes

Si el dato es inmutable (no puede cambiar) durante la ejecución del programa, se dice que ese dato es una **constante**. Si tiene la habilidad de cambiar, se dice que es una variable. En Python, todas las variables son mutables, es decir, pueden cambiar su valor durante la ejecución del programa.
Y no sólo pueden cambiar su valor, sino también su tipo: `x = 5` y `x = "Hola"` son dos asignaciones válidas, y se pueden hacer una debajo de la otra:

```
x = 5
x = "Hola"
print(x)
```

```
Hola
```

Nombres de Variables

No se puede usar el mismo nombre para dos datos diferentes a la vez; una variable puede referenciar un sólo dato por vez. Si se usa un mismo nombre para un dato diferente, se pierde la referencia al dato anterior.

2.1.4 Funciones

Para poder realizar algunas operaciones particulares, necesitamos introducir el concepto de *función*. Una función es un bloque de código que se ejecuta cuando se la llama.
Es un fragmento de programa que permite efectuar una operación determinada. `abs`, `print`, `max` son ejemplos de funciones de Python: `abs`permite calcular el valor absoluto de un número, `print` permite mostrar un valor por pantalla y `max` permite calcular el máximo entre dos valores.

Una función puede recibir cero o más *parámetros* o *argumentos*, que son valores que se le pasan a la función entre paréntesis y separados por comas, para que los use.

```
abs(-5)
```

```
5
```

```
print("¡Hola!")
```

```
¡Hola!
```

```
max(5, 7)
```

```
7
```

La función recibe los parámetros, efectúa una operación y devuelve un *resultado*.

Python viene equipado de muchas funciones predefinidas, pero nosotros como programadores debemos ser capaces de escribir nuevas instrucciones para la computadora. Las grandes aplicaciones como el correo electrónico, navegación web, chat, juegos, etc. no son más que grandes programas implementados introduciendo nuevas funciones a la máquina, escritas por uno o más programadores.

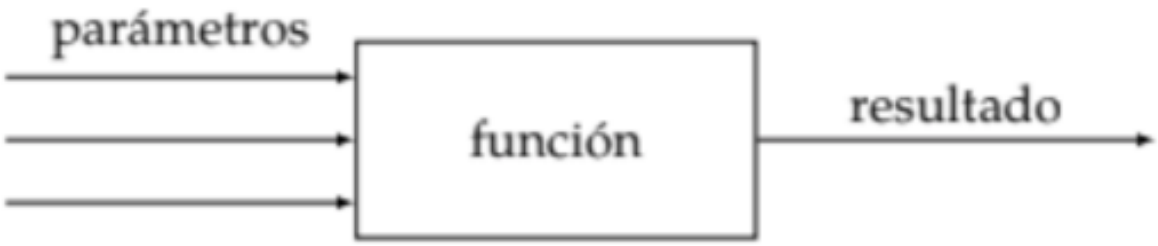


Figure 2.1: Una función recibe parámetros y devuelve un resultado

> **Python es Case Sensitive**
>
> Python es Case Sensitive, es decir, distingue entre mayúsculas y minúsculas.
> Es muy importante respetar mayúsculas y minúsculas: PRINT() o prINT() no serán reconocidas. Esto aplica para todo lo que escribamos en nuestros programas.

Si queremos crear una función que nos devuelva un saludo a Lucia cada vez que se la llama, debemos ingresar el siguiente conjunto de líneas en Python:

```
def saludar_lucia():
  return "Hola, Lucia!"
```

Podés copiar el código y pegarlo en Colab. Luego, apretá “Run”. Vas a notar que no pasa nada, ahora vamos a ver por qué.

Varias cosas a notar del código:

1. `saludar_lucia` es el nombre de la función. Podría ser cualquier otro nombre, pero es una buena práctica que el nombre de la función describa lo que hace.

2. `def` es una palabra clave que indica que estamos definiendo una función.

3. `return` indica el valor que devuelve la función. Es decir, el *resultado*. Puede devolverse una sola cosa, como en este caso, o varias cosas separadas por comas.

4. La sangría (el espacio inicial) en el renglón 2 le indica a Python que estamos dentro del *cuerpo* de la función. El *cuerpo* de la función es el bloque de código que se ejecuta cuando se llama a la misma.

> **Sangría**
>
> La sangría puede ingresarse utilizando dos o más espacios, o presionando la tecla Tab. Es importante prestar atención en no mezclar espacios con tabs, para evitar "confundir" al intérprete (en nuestro caso, Colab).

> **Firma de la función**
>
> La firma de una función es la primera línea de la misma, donde se indica el nombre de la función y los parámetros que recibe. Así como la firma de una persona permite identificarla de otra, la firma de una función permite identificarla y diferenciarla de otra.

Como vimos más arriba, el bloque de código anterior no hace nada. Para que la función haga algo, tenemos que llamarla. Para llamar a una función, escribimos su nombre, seguido de paréntesis y los parámetros que recibe (si es que recibe alguno), separados por comas.

```
saludar_lucia()
```

Se dice que estamos *invocando* o *llamando* a la función. Y al invocar una función, se ejecutan las instrucciones que habíamos escrito en su cuerpo.

Pero de nuevo, vemos que no pasa nada. ¿Por qué? Porque la función usa `return` para devolver un valor. Pero nosotros no estamos haciendo nada con ese valor. Para poder verlo, tenemos que imprimirlo por pantalla.

```
saludo = saludar_lucia()
print(saludo)
```

```
Hola, Lucia!
```

Lo que hicimos fue asignar el resultado devuelto por `saludar_lucia` a la variable `saludo`, y luego imprimir el valor de la variable por pantalla (aunque el paso de guardado es opcional, podríamos imprimirlo directamente).

Bueno, ahora podemos saludar a Lucia. Pero vamos a querer saludar a otras personas también. ¿Tiene sentido hacer una función por cada persona? No, porque tendríamos muchas funciones (llamadas por ejemplo `saludar_lucia`, `saludar_mariana`, `saludar_emilia`, etc) que básicamente hacen lo mismo: saludar a alguien.

Una de las características de una función es que sea una solución reutilizable a un problema. En nuestro caso, queremos saludar personas.

¿Cómo hacemos entonces? Podemos hacer una función que reciba el nombre de la persona a saludar como parámetro. Un parámetro es un valor necesario para la ejecución de la función. En nuestro caso, indica a quién vamos a saludar:

```
def saludar(nombre):
  return "Hola, " + nombre + "!"
```

De esta forma, podemos saludar a cualquier persona, pasando su nombre como parámetro.

```
# Esta es otra forma de imprimir, sin necesidad de guardarnos
# el resultado de la función en una variable,
# simplemente la imprimimos
print(saludar("Lucia"))
```

```
Hola, Lucia!
```

```
print(saludar("Serena"))
```

```
Hola, Serena!
```

> **Return vs Print**
>
> ¿Qué significa que una función devuelva o retorne algo?
> Que una función devuelva o retorne algo, significa que no se está encargando de mostrar el resultado en pantalla. Pero, ¿está haciendo algo si no lo puedo ver en pantalla? Por supuesto, hay muchas cosas que ocurren "por detrás", sin que nos demos cuenta, en una computadora. Si nosotros llamamos a la función `saludar` pasándole un nombre de esta forma `saludar("Serena")`, la función está armando el saludo y devolviéndolo, aunque nosotros no estemos viendo nada en la pantalla. Incluso si lo guardáramos en una variable (`saludo = saludar("Serena")`), también se está ejecutando y la variable `saludo` está

almacenando el saludo, por más de que no veamos eso en ningún lado. La gran mayoría de las funciones van a retornar valores calculados, pero no van a ser responsables de mostrarlos en pantalla. Eso es algo que vamos a tener que ocuparnos por fuera, si quisiéramos ver el resultado.

2.1.4.1 Ejemplos

Ejemplo
Escribir una función que calcule el doble de un número.

```
def obtener_doble(numero):
  return numero * 2
```

Para invocarla, debemos llamarla pasándole un número:

```
doble = obtener_doble(5)
print(doble)
```

```
10
```

Ejemplo
Pensá un número, duplícalo, súmale 6, divídelo por 2 y resta el número que elegiste al comienzo. El número que queda es siempre 3.

```
def f(numero):
  return ((numero * 2) + 6) / 2 - numero
```

```
print(f(5))
```

```
3.0
```

2.1.5 Ingreso de Datos por Consola

Hasta ahora, los programas que hicimos no interactuaban con el usuario. Pero para que nuestros programas puedan interactuar, vamos a querer que el usuario pueda ingresar datos, y que el programa pueda mostrarle datos por pantalla. Para esto, vamos a usar la función `input`.

```
input()
```

Input es una función que bloquea el flujo del programa, esperando a que el usuario ingrese una entrada por consola y presione *enter*. Cuando el usuario presiona *enter*, la función devuelve el valor ingresado por el usuario.

```
input()
print("terminé!")
```

Si corremos el bloque de código anterior (te recomendamos que lo hagas), vamos a tener un comportamiento como este:

1. La consola va a quedar vacía, esperando el ingreso del usuario
2. Ingresamos un valor, el que tengamos ganas, y presionamos enter.
3. La consola muestra el mensaje "terminé!".

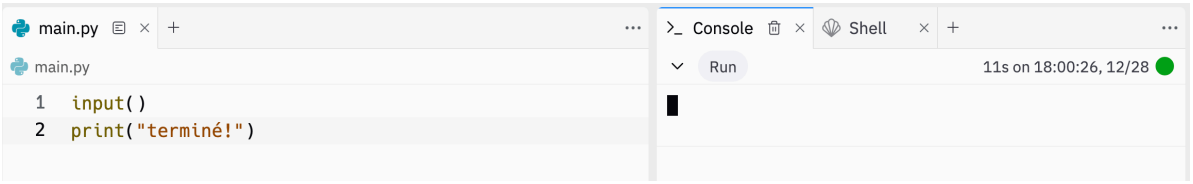


Figure 2.2: Input bloquea el flujo del programa

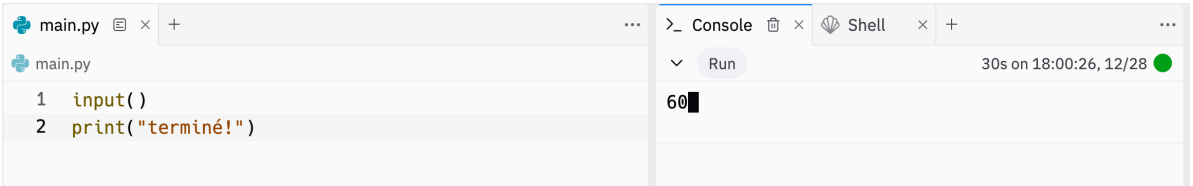


Figure 2.3: Ingresamos un valor (puede ser un número, texto, o ambos)

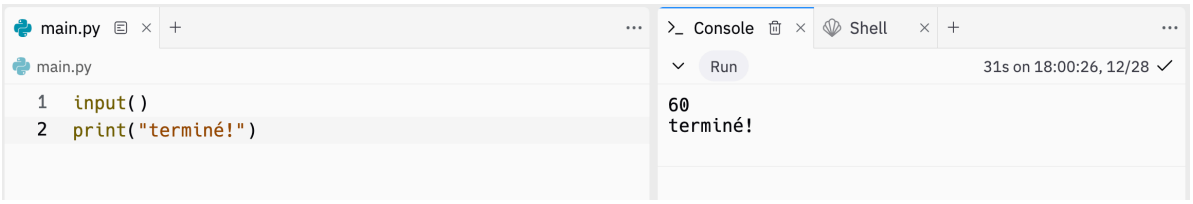


Figure 2.4: Al presionar Enter, la consola muestra el mensaje "terminé!"

2.1.5.1 Obteniendo el Valor Ingresado

Como dijimos más arriba, la función `input` devuelve el valor ingresado por el usuario. Para poder usarlo, tenemos que guardarlo en una variable.

```
nombre = input()
print("Hola, " + nombre + "!")
```

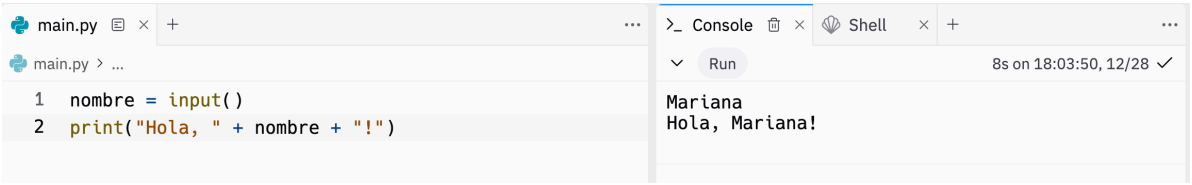


Figure 2.5: Ingresamos “Mariana” y presionamos Enter.

Para hacer nuestro programa más amigable, podemos mostrarle al usuario un mensaje antes de pedirle que ingrese un valor. Para esto, podemos pasarle un parámetro a la función `input`, que es el mensaje que queremos mostrarle al usuario.

```
nombre = input("Ingresá tu nombre: ")
print("Hola, " + nombre + "!")
```

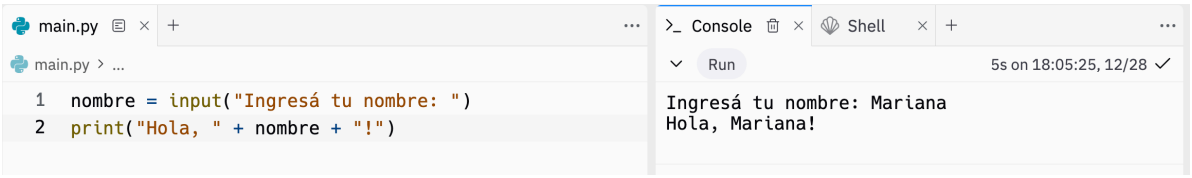


Figure 2.6: Ingresamos “Mariana” y presionamos Enter.

> ⚠ ¡Cuidado!
>
> A partir de la guía 2, a menos que el ejercicio diga específicamente “pedirle al usuario”, no se debe usar `input`, sino que todo tiene que recibirse por parámetro en la función. Lo mismo con `print`: A menos que el ejercicio diga específicamente “imprimir”, todo siempre se tiene que devolver con un return.

2.2 Buenas Prácticas de programación

2.2.1 Sobre Comentarios

Los comentarios son líneas que se escriben en el código, pero que no se ejecutan. Sirven para que el programador pueda dejar notas en el código, para que se entienda mejor qué hace el programa.
Los comentarios se escriben con el símbolo `#`. Todo lo que esté a la derecha del `#` no se ejecuta. También se pueden encerrar entre tres comillas dobles (`"""`) para escribir comentarios de varias líneas.

```
# Esto es un comentario

""" Esto es un comentario
de varias líneas """
```

No es correcto escribir comentarios que no aporten nada al código, o tener el código absolutamente plagado de comentarios. Los comentarios deben ser útiles, y deben aportar información que no se pueda inferir del código. Nuestro primer intento de hacer el código más entendible no tienen que ser los comentarios, sino mejorar el código en sí.

2.2.2 Sobre Convención de Nombres

Para nombres de variables y funciones, en Python se usa *snake_case*, que es básicamente dejar todas las palabras en minúscula y unirlas con un guión bajo. Ejemplos: `numero_positivo`, `sumar_cinco`, `pedir_numero`, etc. Siempre emplear un nombre que nos remita al significado que tendrá ese dato, siempre en snake_case: `numero`, `letra`, `letra2`, `edad_hermano`, etc.

2.2.2.1 Variables

Las variables son cosas. Entonces sus nombres son *sustantivos*: `nombre`, `numero`, `suma`, `resta`, `resultado`, `respuesta_usuario`. La única excepción son las variables booleanas (ya las vamos a ver, son aquellas que pueden guardar dos posibles valores: verdadero o falso), que suelen tener nombres como `es_par`, `es_cero`, `es_entero`, porque su valor es true o false.

A veces es útil alguna frase para identificar mejor el contenido:
`edad_mayor_hijo`, `apellido_conyuge`

2.2.2.2 Funciones

Las funciones hacen algo. Entonces sus nombres son *verbos*. Se usan siempre verbos en infinitivo (terminan en `-ar`, `-er`, `-ir`): `calcular_suma`, `imprimir_mensaje`, `correr_prueba`, `obtener_triplicado`, etc.
De nuevo, las excepciones son las funciones que devuelven un valor booleano (V o F). Esas pueden llamarse como: `es_par`, `da_cero`, `tiene_letra_a`, porque devuelven verdadero o falso, y eso nos confirma o niega la afirmación que hace el nombre.

2.2.3 Sobre Ordenamiento de Código

Cuando uno corre Python, lo que hace el lenguaje es leer línea a línea nuestro código. Lo que se puede ejecutar, lo ejecuta. Las funciones las guarda en memoria para poder usarlas luego. Entonces es más ordenado y prolijo primero poner todas las funciones, y después el código "ejecutable" (si van a dejar código suelto en el archivo, cosa que en general no se suele recomendar).

Además, no olvidemos que Python tiene un flujo de control de arriba para abajo. Si intentamos invocar funciones antes de que estén definidas (`def`), Python no va a saber qué hacer, y nos va a tirar un error.

Esto es correcto:

Esto es incorrecto:

2.2.4 Sobre uso de Parámetros en Funciones

Una función se puede pensar como una caja cerrada o una fábrica. La función tiene dos puertas: una de entrada y una de salida.
La puerta de entrada son los `parámetros` y la de salida es el `output` (el resultado).

Cuando se llama o invoca a la función, la puerta de entrada se abre, permitiéndonos enviarle (pasarle) cero, uno o más parámetros a la función (según cómo esté definida). Los parámetros son datos que la función necesita para funcionar, y como ya dijimos, se le pasan a la misma entre los paréntesis de la llamada.

> **Ejemplo:** `saludar(nombre)`, `imprimir_elementos(lista)`, `sumar(numero1, numero2)`, etc.

Una vez que la función se empieza a ejecutar, ambas puertas se cierran. Esto quiere decir que, mientras la función se está ejecutando, nada entra y nada sale de la misma.
La función debería trabajar únicamente con los datos que se le hayan pasado por parámetro o que se le pidan al usuario dentro de ella, pero no debería utilizar nada que esté por fuera de la misma.

> ⚠ ¡Cuidado!
>
> Python nos deja usar cosas por fuera de la función y sin recibir los datos por parámetro, porque es un lenguaje muy benevolente. Pero está mal usar cosas que no se hayan recibido por parámetro: es una mala práctica.

Una vez que la función terminó de ejecutarse, el o los valores de salida (resultados) se devuelven por el output. Una función puede retornar uno o más elementos, o podría simplemente no retornar nada.
`return suma`, `return numero1, numero2`, `return`, etc.

Podemos ver la diferencia entre enviar algo por parámetro y usarlo por fuera de la función a continuación:

Esto está mal

Esto está bien

```
def saludar():
  print("Hola, " + nombre + "!")

nombre = "Manuela"
saludar()
```

```
def saludar(persona):
  print("Hola, " + persona + "!")

nombre = "Manuela"
saludar(nombre)
```

> 💡 Tip
>
> Como podemos observar los nombres de los argumentos cuando se invoca y en la definición de la firma pueden ser los mismos o distintos. En este caso, la función sabe que está recibiendo algo como parámetro, y sabe que dentro de su cuerpo a este dato lo va a identificar como `persona`, pero no hace falta que la variable que nosotros le pasamos como parámetro también se llame `persona`: en este caso se llama `nombre`.

2.3 Tipos de Datos

2.3.1 Datos Simples

Los programas trabajan con una gran variedad de datos. Los datos más simples son los que ya vimos: números enteros, números de punto flotante y cadenas.
Pero dependiendo de la naturaleza o el **tipo** de información, cabrá la posibilidad de realizar distintas transformaciones aplicando **operadores**. Por eso, a la hora de representar información no sólo es importante que identifiquemos al dato y podamos conocer su valor, sino saber qué tipo de tratamiento podemos darle.

Todos los lenguajes tienen tipos predefinidos de datos. Se llaman predefinidos porque el lenguaje ya los conoce: sabe cómo guardarlos en memoria y qué transformaciones puede aplicarles.

En Python, tenemos los siguientes tipos de datos:

Tipo	Descripción	Ejemplo
`int`	Números enteros	`5`, `0`, `-5`, `10000`
`float`	Números de punto flotante o reales	`3.14159`, `1.0`, `0.0`
`complex`	Números complejos	`(1, 2j)`, `(1.0,-2.0j)`, `(0,1j)`. La componente con `j` es la parte imaginaria.
`bool`	Valores booleanos o valores lógicos	`True`, `False`
`str`	Cadenas de caracteres	`"Hola"`, `"Python"`, `"¡Hola, mundo!"`, `""` (cadena vacía, no contiene ningún caracter)

> **i** ¿Por qué se llaman “cadenas de caracteres”?
>
> Porque son una cadena de caracteres, es decir, una secuencia de caracteres. Por ejemplo, la cadena “Hola” está formada por los caracteres “H”, “o”, “l” y “a”. Esto nos permite acceder a cada uno de los caracteres de la cadena por separado si quisiéramos, o a porciones de una cadena, como vamos a ver más adelante.
> Más aún, podemos ver que el texto “hola” no será igual a “aloh” ni a “Holá”, porque son cadenas distintas.
> Un string permite almacenar cualquier tipo de caracter unicode dentro (letras, números, símbolos, emojis, etc.).

2.3.2 Operadores Numéricos

Los operadores son símbolos que representan una operación. Por ejemplo, el operador `+` representa la suma.

Para transformar datos numéricos, emplearemos los siguientes operadores:

Símbolo	Definición	Ejemplo
`+`	Suma	`5 + 3`
`-`	Resta	`5 - 3`
`*`	Producto	`5 * 3`
`**`	Potencia	`5 ** 2`
`/`	División	`5 / 3`
`//`	División entera	`5 // 3`
`%`	Módulo o Resto	`5 % 3`
`+=`	Suma abreviada	`x = 0x += 3`
`-=`	Resta abreviada	`x = 0x -= 3`
`*=`	Producto abreviado	`x = 0x *= 3`
`/=`	División abreviada	`x = 0x /= 3`
`//=`	División entera abreviada	`x = 0x //= 3`
`%=`	Módulo o Resto abreviado	`x = 0x %= 3`

Como pasa en matemática, para alterar cualquier precedencia (prioridad de operadores) se pueden usar paréntesis.

```
(5 + 3) * 2
```

```
16
```

```
5 + (3 * 2)
```

```
11
```

El orden de prioridad de ejecución para los operadores va a ser el mismo que en matemática.

2.3.3 Operadores de Texto

Para transformar datos de texto, emplearemos los siguientes operadores:

Símbolo	Definición	Ejemplo
`+`	Concatenación	`"Hola" + " " + "Mundo"`
`*`	Repetición	`"Hola" * 3`
`+=`	Concatenación abreviada	`x = "Hola"x += " Mundo"`
`*=`	Repetición abreviada	`x = "Hola"x *= 3`
[k] o [-k]	Acceso a un caracter	`"Hola"[0]"Hola"[-1]`
[k1:k2]	Acceso a una porción	`"Hola"[0:2]"Hola"[1:]"Hola"[:2]"Hola"[:`

De nuevo, para alterar precedencias, se deben usar `()`.

2.3.3.1 Manipulando Strings

Si bien esto se va a ahondar en la siguiente sesión de la materia, es importante saber que los strings, como se dijo más arriba, son un conjunto de caracteres. Pero no sólo un conjunto, sino un **conjunto ordenado**. Esto quiere decir que cada caracter tiene una posición dentro de la cadena, y que esa posición es importante.

Por ejemplo, la cadena `"Hola"` tiene 4 caracteres: `"H"`, `"o"`, `"l"` y `"a"`.
La posición de cada caracter es la siguiente:

Posición	0	1	2	3
Caracter	“H”	“o”	“l”	“a”

Entonces, si queremos acceder al caracter `"H"`, tenemos que usar la posición 0. Si queremos acceder al caracter `"a"`, tenemos que usar la posición 3.

> **Tip**
>
> Para acceder a un caracter de una cadena, usamos los corchetes (`[]`) y dentro de ellos la posición del caracter que queremos acceder.

```
letra = "Hola"[0]
print(letra)
```

```
H
```

Pero no sólo puedo obtener los caracteres en las posicione de la palabra, sino que puedo obtener *slices* o *porciones* de la misma, usando algo que vemos por primera vez: los **rangos**.

Un rango tiene tres partes:

```
[start : end : step]
```

- `start` es el índice de inicio del rango. Si no se especifica, se toma el índice 0. El caracter en la posición de inicio siempre se incluye.
- `end` es el índice de fin del rango. Si no se especifica, se toma el índice final de la cadena. El caracter en la posición de fin nunca se incluye.
- `step` es el tamaño del paso. Si no se especifica, se toma el valor 1.

  Ejemplos:

2.3.4 Input y Casteo

Cuando usamos la función `input`, el valor que devuelve es siempre una cadena. Esto es porque el usuario puede ingresar cualquier cosa, y no sabemos qué tipo de dato es.

Por ejemplo, si le pedimos al usuario que ingrese un número, el usuario puede ingresar un número entero, un número de punto flotante, un número complejo, o incluso un texto. Entonces, el valor que devuelve `input` es siempre una cadena, y nosotros tenemos que transformarla al tipo de dato que necesitemos.

Por ejemplo:

```
edad = input("Indique su edad:")
print("Su edad es:", edad_nueva)
```

> Imprimiendo Strings y Variables (Iterpolación de Cadenas)
>
> Existen muchas formas de concatenar variables con texto.
>
> 1. Usando el operador `+`: `"Su edad es: " + edad`

2. Usando el método `fstring`: `f"Su edad es: {edad}"`
3. Usando el caracter `,`: `print("Su edad es:", edad)`

La forma más recomendada es la segunda, usando `fstring`. Pero dependerá de cada caso.

El problema es que, si bien nuestro código anterior funciona, no podemos operar `edad` como si fuese un número, porque es un string.
El siguiente código va a fallar:

```
edad = input("Indique su edad:")
edad_nueva = edad + 1
print("Edad siguiente:", edad_nueva)
```

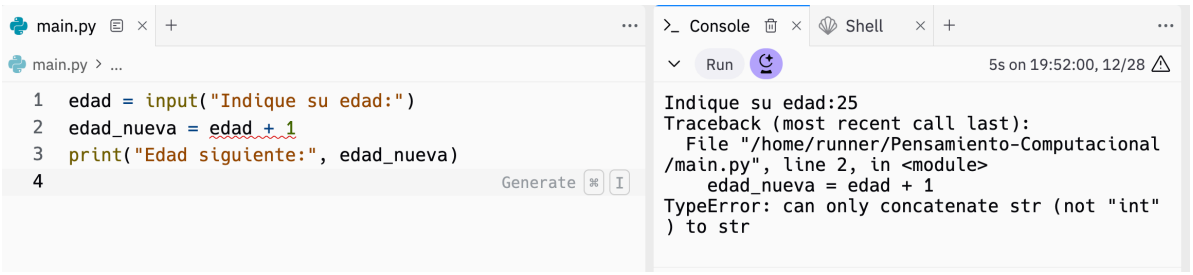


Figure 2.7: Ejecución del bloque de código

Como vemos, la consola nos arroja un error, o en términos simples decimos que "explotó".

¿Qué es un error?

Los errores son información que nos da la consola para que podamos corregir nuestro código.

En este caso, nos dice que no se puede concatenar un string con un int.
¿Por qué nos dice eso? Porque `edad` es un string: `"25"`, y estamos tratando de sumarle 1, que es un int: `1`.

Para poder operar con `edad` como si fuese un número, tenemos que transformarla a un número. Esto se llama **castear**.

Para castear un valor a un tipo de dato, usamos el nombre del tipo de dato, seguido de paréntesis y el valor que queremos castear.

```
int("25")
```

De esta forma, podemos modificar nuestro código anterior:

```
edad = int(input("Indique su edad:")) # Le agregamos int
edad_nueva = edad + 1
print("Edad siguiente:", edad_nueva)
```

Y obtenemos un código que funciona correctamente.

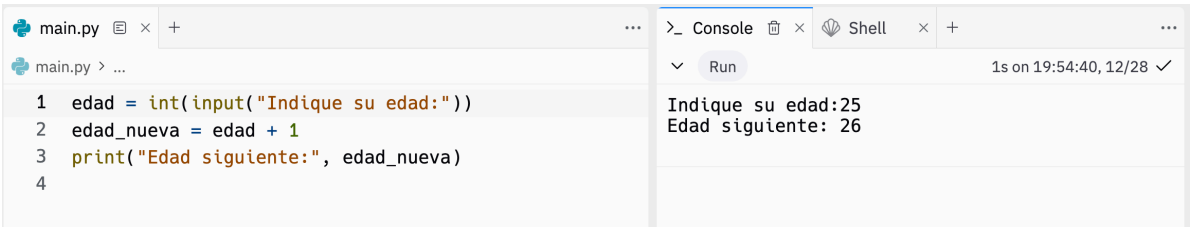


Figure 2.8: Ejecución del bloque de código

De esta forma, podemos castear a varios tipos de datos:

```
numero_entero = int(input("Ingrese un número"))
punto_flotante = float(input("Ingrese un número"))

punto_flotante2 = float(numero_entero)

numero_en_str = str(numero_entero)
```

Ejemplo:

```
nombre_menor = input('Ingresá el nombre de un conocido/a:')
edad_menor = int(input(f'Ingresá la edad de { nombre_menor } '))
nombre_mayor = input(f'Cómo se llama el hermano/a mayor de {nombre_menor}? ')
diferencia = int(input(f'Cuántos años más grande es {nombre_mayor}?  '))

edad_mayor = edad_menor + diferencia

print(nombre_menor,'tiene',edad_menor,'años')
print(nombre_mayor,'es mayor y tiene', edad_mayor, 'años')
```

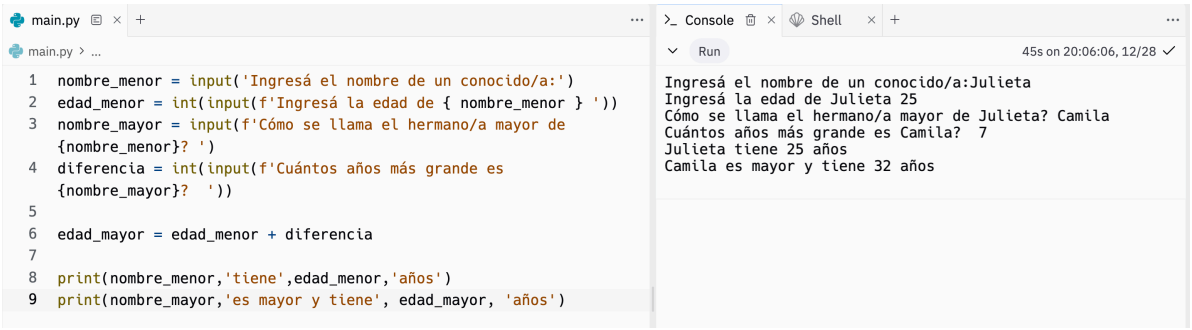


Figure 2.9: Ejecución del bloque de código

2.4 Bonus Track: Algunas Funciones Predefinidas de Python

> **Recomendación**
>
> Te recomendamos que te animes a probar estas funciones, para ver qué hacen y terminar de entenderlas.

Función	Definición	Ejemplo de uso
`print()`	Imprime un mensaje o valor en la consola	`print("Hello, world!")`
`input()`	Lee una entrada de texto desde el usuario	`name = input("Enter your name: ")`
`abs()`	Devuelve el valor absoluto de un número	`abs(-5)`
`round()`	Redondea un número al entero más cercano	`round(3.7)`
`int()`	Convierte un valor en un entero	`x = int("5")`
`float()`	Convierte un valor en un número de punto flotante	`y = float("3.14")`
`str()`	Convierte un valor en una cadena de texto	`message = str(42)`
`bool()`	Convierte un valor en un booleano	`is_valid = bool(1)`
`len()`	Devuelve la longitud (número de elementos) de un objeto	`length = len("Hello")`

Función	Definición	Ejemplo de uso
`max()`	Devuelve el valor máximo entre varios elementos o una secuencia	`max(4, 9, 2)`
`min()`	Devuelve el valor mínimo entre varios elementos o una secuencia	`min(4, 9, 2)`
`pow()`	Calcula la potencia de un número	`result = pow(2, 3)`
`range()`	Genera una secuencia de números	`numbers = range(1, 5)`
`type()`	Devuelve el tipo de un objeto	`data_type = type("Hello")`
`round()`	Redondea un número a un número de decimales específico	`rounded_num = round(3.14159, 2)`
`isinstance()`	Verifica si un objeto es una instancia de una clase específica	`is_instance = isinstance(5, int)`
`replace()`	Reemplaza todas las apariciones de un substring por otro	`text = "Hello, World!"new_text = text.replace("Hello", "Hi")`
`eval(<expr>)`	Evalúa una expresión	`eval("2 + 2")`

3 Estructuras de Control

3.1 Decisiones

> **Ejemplo** Leer un número y, si el número es positivo, imprimir en pantalla "Número positivo".

Necesitamos *decidir* de alguna forma si nuestro número x es positivo (>0) o no. Para resolver este problema, introducimos una nueva instrucción, llamada *condicional*: `if`.

```
if <expresion>:
    <cuerpo>
```

Donde `if` es una palabra reservada, `<expresion>` es una *condición* y `<cuerpo>` es un bloque de código que se ejecuta sólo si la condición es verdadera.

Por lo tanto, antes de seguir explicando sobre la instrucción `if`, debemos entender qué es una *condición*. Estas expresiones tendrán valores del tipo *sí* o *no*.

3.1.1 Expresiones Booleanas

Las expresiones booleanas forman parte de la lógica binomial, es decir, sólo pueden tener dos valores: `True` (verdadero) o `False` (falso). Estos valores no tienen elementos en común, por lo que no se pueden comparar entre sí. Por ejemplo, `True > False` no tiene sentido. Y además, son complementarios: algo que **no** es True, es False; y algo que **no** es False, es True. Son las únicas dos opciones posibles.

Python, además de los tipos numéricos como `int`y `float`, y de las cadenas de caracteres `str`, tiene un tipo de datos llamado `bool`. Este tipo de datos sólo puede tener dos valores: `True` o `False`. Por ejemplo:

```
n = 3 # n es de tipo 'int' y tiene valor 3
b = True # b es de tipo 'bool' y tiene valor True
```

3.1.2 Expresiones de Comparación

Las expresiones booleanas se pueden construir usando los operadores de comparación: sirven para comparar valores entre sí, y permiten construir una pregunta en forma de código.

Por ejemplo, si quisiéramos saber si `5` es mayor a `3`, podemos construir la expresión:

```
print(5 > 3)
```

```
True
```

Como 5 es en efecto mayor a 3, esta expresión, al ser evaluada, nos devuelve el valor `True`.

Si quisiéramos saber si `5` es menor a `3`, podemos construir la expresión:

```
print(5 < 3)
```

```
False
```

Como 5 no es menor a 3, esta expresión, al ser evaluada, nos devuelve el valor `False`.

Las expresiones booleanas de comparación que ofrece Python son:

Expresión	Significado
`a == b`	`a` es igual a `b`
`a != b`	`a` es distinto de `b`
`a < b`	`a` es menor que `b`
`a > b`	`a` es mayor que `b`
`a <= b`	`a` es menor o igual que `b`
`a >= b`	`a` es mayor o igual que `b`

Veamos algunos ejemplos:

```
5 == 5
```

```
5 != 5
```

```
5 < 5
```

```
5 >= 5
```

```
5 > 4

5 <= 4
```

> **Tip**
>
> Te recomendamos fuertemente probar estas expresiones para ver qué valores devuelven. Podés hacerlo de dos formas:
>
> 1. Guardando el resultado de la expresión en una variable, para luego imprimirla:
>
> ```
> resultado = 5 == 5
> print(resultado)
> ```
>
> 2. Imprimiendo directamente el resultado de la expresión:
>
> ```
> print(5 == 5)
> ```

3.1.3 Operadores Lógicos

Además de los operadores de comparación, Python también tiene operadores lógicos, que permiten combinar expresiones booleanas para construir expresiones más complejas. Por ejemplo, quizás no sólo queremos saber si `5` es mayor a `3`, sino que también queremos saber si `5` es menor que `10`. Para esto, podemos usar el operador `and`:

```
5 > 3 and 5 < 10
```

Python tiene tres operadores lógicos: `and`, `or` y `not`. Veamos qué hacen:

Operador	Significado
`a and b`	El resultado es `True`solamente si `a` es `True` **y** `b` es `True`. Ambos deben ser `True`, de lo contrario devuelve `False`.
`a or b`	El resultado es `True` si `a` es `True` **o** `b` es `True` (o ambos). Si ambos son `False`, devuelve `False`.
`not a`	El resultado es `True` si `a` es `False`, y viceversa.

Algunos ejemplos:

```
5 > 2 and 5 > 3
```

```
True
```

```
5 > 2 or 5 > 3
```

```
True
```

```
5 > 2 and 5 > 6
```

```
False
```

```
5 > 2 or 5 > 6
```

```
True
```

```
5 > 6
```

```
False
```

```
not 5 > 6
```

```
True
```

```
5 > 2
```

```
True
```

```
not 5 > 2
```

```
False
```

Prioridad de Operadores

Las expresiones lógicas complejas (con más de un operador), se resuelven al igual que en matemática: respetando precedencias y de izquierda a derecha. También admiten el uso

de `()` para alterar las precedencias.
Sin embargo, si no tenemos precedencias explícitas con `()`, Python prioriza resolver primero los `and`, luego los `or` y por último los `not`.
Ejemplos:

```
True or False and False
```

```
True
```

Por la prioridad del `and`, primero se resuelve `False and False`, que da `False`. Luego, se resuelve `True or False`, que da `True`.

```
True or False or False
```

```
True
```

Como no hay `and`, se resuelve de izquierda a derecha. Primero se resuelve `True or False`, que da `True`. Luego, se resuelve `True or False`, que da `True`.

```
(True or False) and False
```

```
False
```

Como hay paréntesis, se resuelve primero lo que está dentro de los paréntesis. `True or False` da `True`. Luego, `True and False` da `False`.

3.1.4 Comparaciones Simples

Volvamos al problema inicial: Queremos saber, dado un número x, si es positivo o no, e imprimir un mensaje en consecuencia.
Recordemos la instrucción `if` que acabamos de introducir y que sirve para tomar decisiones simples. Esta instrucción tiene la siguiente estructura:

```
if <expresion>:
    <cuerpo>
```

donde:

1. `<expresion>`debe ser una expresión lógica.
2. `<cuerpo>`es un bloque de código que se ejecuta sólo si la expresión es verdadera.

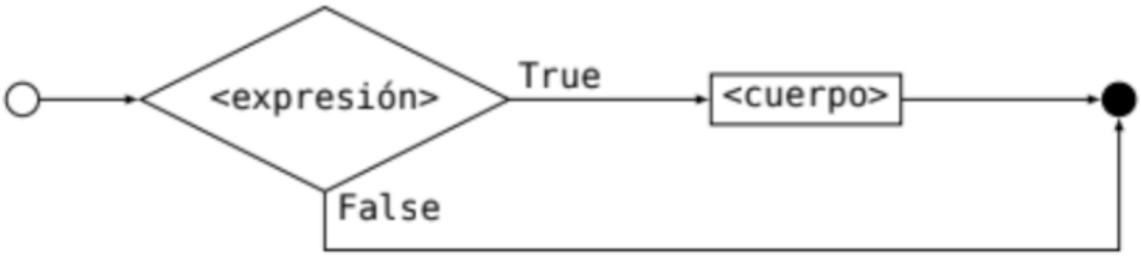


Figure 3.1: Diagrama de Flujo para la instrucción `if`

Como ahora ya sabemos cómo construir condiciones de comparación, vamos a comparar si nuestro número `x` es mayor a `0`:

```
def imprimir_si_positivo(x):
  if x > 0:
      print("Número positivo")
```

Podemos probarlo:

```
imprimir_si_positivo(5)
imprimir_si_positivo(-5)
imprimir_si_positivo(0)
```

```
Número positivo
```

Como vemos, si el número es positivo, se imprime el mensaje. Pero si el número no es positivo, no se imprime nada. Necesitamos además agregar un mensaje "Número no positivo", si es que la condición no se cumple.

Modifiquemos el diseño: 1. Si $x > 0$, se imprime "Número positivo". 2. En caso contrario, se imprime "Número no positivo".

Podríamos probar con el siguiente código:

```
def imprimir_si_positivo(x):
  if x > 0:
      print("Número positivo")
  if not x > 0:
      print("Número no positivo")
```

Otra solución posible es:

```
def imprimir_si_positivo(x):
  if x > 0:
      print("Número positivo")
  if x <= 0:
      print("Número no positivo")
```

Ambas están bien. Si lo probamos, vemos que funciona:

```
imprimir_si_positivo(5)
imprimir_si_positivo(-5)
imprimir_si_positivo(0)
```

```
Número positivo
Número no positivo
Número no positivo
```

Sin embargo, hay una mejor forma de hacer esta función. Existe una condición alternativa para la estructura de decisión `if`, que tiene la forma:

```
if <expresion>:
    <cuerpo>
else:
    <cuerpo>
```

donde `if` y `else` son palabras reservadas. Su efecto es el siguiente:

1. Se evalúa la `<expresion>`.
2. Si la `<expresion>` es verdadera, se ejecuta el `<cuerpo>` del `if`.
3. Si la `<expresion>` es falsa, se ejecuta el `<cuerpo>` del `else`.

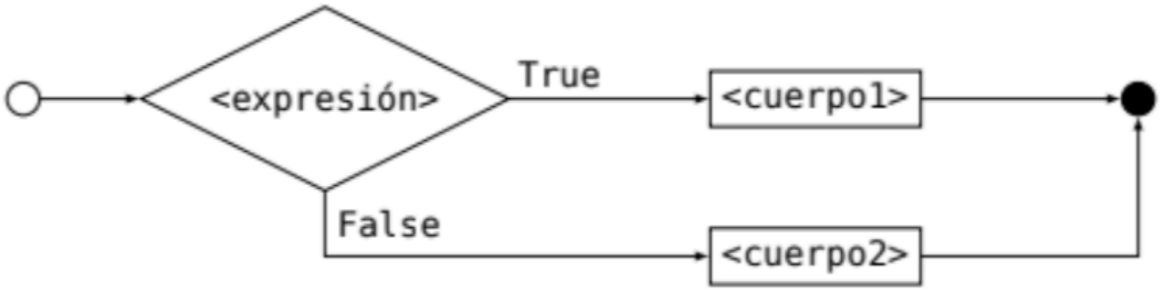


Figure 3.2: Diagrama de Flujo para la instrucción `if-else`

Por lo tanto, podemos reescribir nuestra función de la siguiente forma:

```
def imprimir_si_positivo_o_no(x): # le cambiamos el nombre
  if x > 0:
      print("Número positivo")
  else:
      print("Número no positivo")
```

Probemos:

```
imprimir_si_positivo_o_no(5)
imprimir_si_positivo_o_no(-5)
imprimir_si_positivo_o_no(0)
```

```
Número positivo
Número no positivo
Número no positivo
```

¡Sigue funcionando!

Lo importante a destacar es que, si la condición del `if` es verdadera, se ejecuta el `<cuerpo>` del `if` y **no** se ejecuta el `<cuerpo>` del `else`. Y viceversa: si la condición del `if` es falsa, se ejecuta el `<cuerpo>` del `else` y **no** se ejecuta el `<cuerpo>` del `if`. Nunca se ejecutan ambos casos, porque son caminos paralelos que no se cruzan, como vimos en el diagrama de flujo más arriba.

3.1.5 Múltiples decisiones consecutivas.

Supongamos que ahora queremos imprimir un mensaje distinto si el número es positivo, negativo o cero. Podríamos hacerlo con dos decisiones consecutivas:

```
def imprimir_si_positivo_negativo_o_cero(x):
  if x > 0:
      print("Número positivo") # cuerpo del primer if
  else:
      if x == 0:                      #
          print("Número cero")        #
      else:                           #
          print("Número negativo")    # todo esto es el cuerpo del primer else
```

A esto se le llama *anidar*, y es donde dentro de unas ramas de la decisión (en este caso, la del `else`), se anida una nueva decisión. Pero no es la única forma de implementarlo. Podríamos hacerlo de la siguiente forma:

```
def imprimir_si_positivo_negativo_o_cero(x):
  if x > 0:
      print("Número positivo")
  elif x == 0:
      print("Número cero")
  else:
      print("Número negativo")
```

La estructura `elif` es una abreviatura de `else if`. Es decir, es un `else` que tiene una condición. Su efecto es el siguiente:

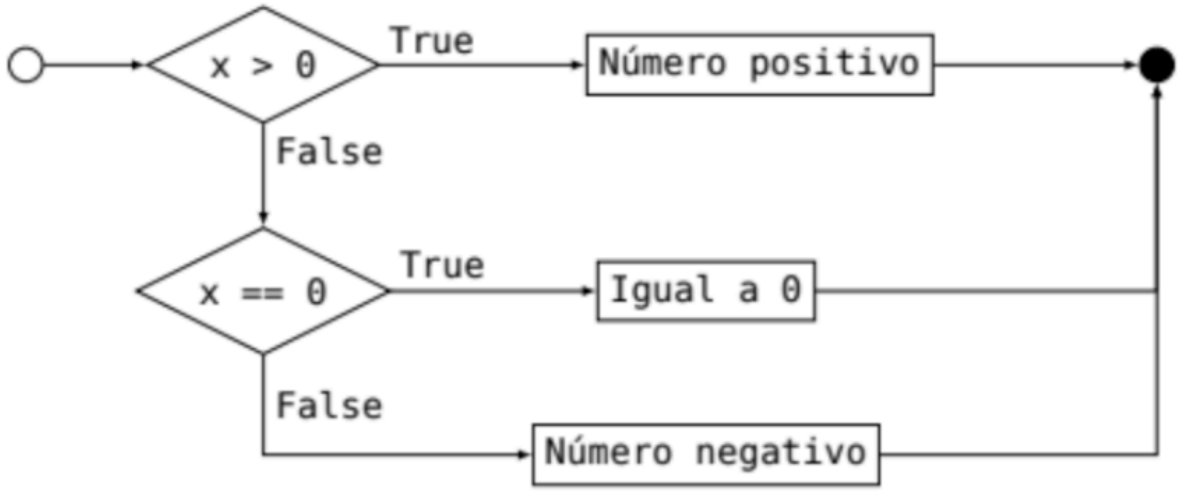


Figure 3.3: Diagrama de Flujo para la instrucción `if-elif-else` del ejemplo

1. Se evalúa la `<expresion>` del `if`.
2. Si la `<expresion>` es verdadera, se ejecuta el `<cuerpo>` del `if`.
3. Si la `<expresion>` es falsa, se evalúa la `<expresion>` del `elif`.
4. Si la `<expresion>` del `elif` es verdadera, se ejecuta su `<cuerpo>`.
5. Si la `<expresion>` del `elif` es falsa, se ejecuta el `<cuerpo>` del `else`.

Sabías que... ?

En Python se consideran *verdaderos* (True) también todos los valores numéricos distintos de 0, las cadenas de caracteres que no sean vacías, y cualquier valor que no sea vacío en

general. Los valores nulos o vacíos son *falsos*.

```
if x == 0:
```

es equivalente a:

```
if not x:
```

Y además, existe el valor especial **None**, que representa la ausencia de valor, y es considerado *falso*. Podemos preguntar si una variable tiene el valor `None` usando el operador `is`:

```
if x is None:
```

o también:

```
if not x:
```

! Ejercicio Desafío (opcional)

Debemos calcular el pago de una persona empleada en nuestra empresa. El cálculo debe hacerse por la cantidad de horas trabajadas, y se le debe pedir al usuario la cantidad de horas y cuánto vale cada hora.
Adicionalmente, se abona un plus fijo de guardería a todo empleado/a con infantes a su cargo. Y se paga un 10% de incentivo a todo empleado/a que haya trabajado 30 horas o más y **no** reciba el plus por guardería.

Pista: pensar los distintos tipos de liquidación:
a) Empleado/a con menos de 30 horas y sin infantes a cargo.
b) Empleado/a con 30 horas o más y sin infantes a cargo.
c) Empleado/a con menos de 30 horas y con infantes a cargo.
d) Empleado/a con 30 horas o más y con infantes a cargo.

Ayuda: Flujo de la resolución

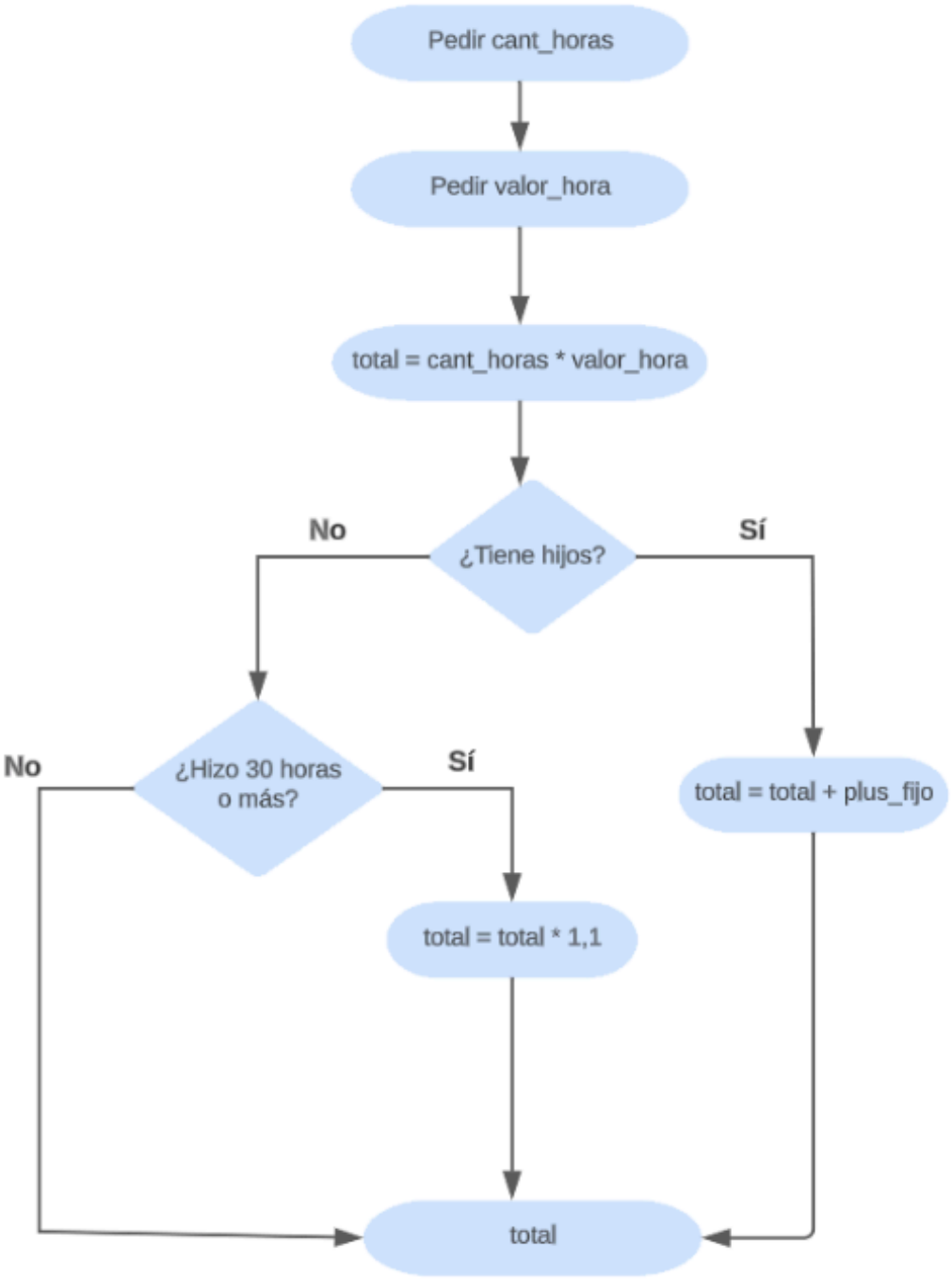


Figure 3.4: Diagrama de Flujo para el desafío

3.2 Ciclos y Rangos

Supongamos que en una fábrica se nos pide hacer un procedimiento para entrenar al personal nuevo. Para comenzar se nos encarga la descripción de uno muy simple: descarga de cajas de material del camión del proveedor y almacenamiento en el depósito. Así que aplicamos lo que venimos aprendiendo hasta ahora sobre algoritmos y describimos la operación para la descarga de 3 cajas:

```
1 Abrir la puerta del depósito y encender luces

2 Ir al garage o playón donde estacionó el camión
3 Abrir las puertas traseras de la caja del transporte
4 Tomar una caja con ambas manos, asegurándola para no tirarla
5 Caminar sosteniendo la caja hasta el depósito
6 Colocar la caja sobre el piso en el sector correspondiente

7 Ir al garage o playón donde estacionó el camión
8 Tomar OTRA caja con ambas manos, asegurándola para no tirarla
9 Caminar sosteniendo la caja hasta el depósito
10 Colocar la caja sobre la caja anterior

11 Ir al garage o playón donde estacionó el camión
12 Tomar OTRA caja con ambas manos, asegurándola para no tirarla
13 Caminar sosteniendo la caja hasta el depósito
14 Colocar la caja sobre la caja anterior

15 Apagar luces y cerrar puerta del depósito
16 Ir al garage o playón donde estacionó el camión
17 Cerrar y trabar puertas del camión
18 Avisar fin de descarga al transportista
```

Ya lo tenemos. Ahora la persona a cargo dice que en el camión suelen venir entre 5 y 15 cajas de material y pide que definas el mismo procedimiento para todos los casos posibles. Notemos que se repiten las instrucciones 2, 3, 4, 5 y 6 para cada caja ¿Qué hacemos? ¿Vamos a seguir copiando y pegando las instrucciones para cada caja? ¿Y si algún día vienen más de 15 o menos de 5? ¿Vamos a tener una lista de instrucciones distinta para cada cantidad de cajas que puedan venir? Parece ser necesario hacer algo más genérico que le facilite la vida a todos. Una nueva versión:

```
1 Abrir la puerta del depósito y encender luces
2 Ir al garage o playón donde estacionó el camión
```

```
3 Abrir las puertas traseras de la caja del transporte
4 Tomar una caja con ambas manos, asegurándola para no tirarla
5 Caminar sosteniendo la caja hasta el depósito
6 Si es la primera caja, colocarla sobre el piso en el sector correspondiente;
si no, apilarla sobre la anterior;
salvo que ya haya 3 apiladas,
en ese caso colocarla a la derecha sobre el piso
7 Ir al garage o playón donde estacionó el camión

8 Repetir 4,5,6,7 mientras queden cajas para descargar

9 Cerrar y trabar puertas del camión
10 Avisar fin de descarga al transportista
11 Volver a depósito
12 Apagar luces y cerrar puerta del depósito
```

Esta descripción es bastante más compacta y cubre todas las posibles cantidades de cajas en un envío (habituales y excepcionales), de modo que con una única página en el manual de procedimientos será suficiente.

Sin embargo, los algoritmos que venimos escribiendo se parecen más al primer procedimiento que al segundo. ¿Cómo podemos mejorarlos?

> **Ciclos**
>
> El **ciclo**, **bucle** o **sentencia iterativa** es una instrucción que permite ejecutar un bloque de código varias veces. En Python, existen dos tipos de ciclos: `while` y `for`.

3.2.1 Ciclo `for`

La instrucción `for` nos indica que queremos repetir un bloque de código una cierta cantidad de veces. Por ejemplo, si queremos imprimir los números del 1 al 10, podemos hacerlo de la siguiente forma:

```python
for i in range(1, 11):
    print(i)
```

```
1
2
3
4
```

```
5
6
7
8
9
10
```

El ciclo `for` incluye una línea de *inicialización* y una línea de `<cuerpo>`, que puede tener una o más instrucciones. El ciclo definido es de la forma:

```
for <nombre> in <expresion>:
    <cuerpo>
```

El ciclo se dice *definido* porque una vez evaluada la `<expresion>`, se sabe cuántas veces se va a ejecutar el `<cuerpo>`: tantas veces como elementos tenga la `<expresion>`.

La expresión puede indicarse con `range`:

- `range(n)` devuelve una secuencia de números desde 0 hasta `n-1`.
- `range(a, b)` devuelve una secuencia de números desde `a` hasta `b-1`.
- `range(a, b, c)` devuelve una secuencia de números desde `a` hasta `b-1`, de a `c` en `c`.

Se podría decir que el `range` puede recibir 3 valores: `range(start, end, step)` o `range(inicio, fin, paso)`, donde:

- `start` o `inicio` es el valor inicial de la secuencia. Por defecto es 0.
- `end` o `fin` es el valor final de la secuencia. **No** se incluye en la secuencia.
- `step` o `paso` es el incremento entre cada elemento de la secuencia. Por defecto es 1.

Si le pasamos un sólo parámetro, lo toma como `end`.
Si le pasamos dos, los toma como `start` y `end`.
Y si le pasamos tres, los toma como `start`, `end` y `step`.

> **Note**
>
> ¿Te suena quizás a algo que ya vimos? Quizás... ¿los *slices* de las cadenas de caracteres?

Además, la variable `<nombre>` va a ir tomando el valor de cada elemento de la `<expresion>` en cada iteración. En nuestro ejemplo de imprimir los números del 1 al 10, vemos que `i` toma los valores 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10, en ese orden.

Ejemplo
Se pide una función que imprima todos los números pares entre dos números dados `a` y `b`. Se considera que `a` y `b` son siempre números enteros positivos, y que `a` es menor que `b`.

```
def imprimir_pares(a, b):
  for i in range(a, b):
    if i % 2 == 0: # si el resto de dividir por 2 es cero, es par
      print(i)

imprimir_pares(1,15)
```

```
2
4
6
8
10
12
14
```

Ejemplo
Se pide una función que imprima todos los números del 1 al 10, en orden inverso.

```
def imprimir_inverso():
  for i in range(10, 0, -1):
      print(i)

imprimir_inverso()
```

```
10
9
8
7
6
5
4
3
2
1
```

3.2.1.1 Iterables

Como dijimos más arriba, la expresión del `for` puede ser cualquier expresión que devuelva una secuencia de valores. A estas expresiones se las llama *iterables*.

Un ciclo `for` también podría iterar sobre elementos de una lista (tema que vamos a ver más adelante), o sobre caracteres de una palabra. Por ejemplo:

```
for num in [1, 3, 7, 5, 2]:
    print(num)
```

```
1
3
7
5
2
```

```
for c in "Hola":
    print(c)
```

```
H
o
l
a
```

3.2.2 Ciclo `while`

La instrucción `while` nos indica que queremos repetir un bloque de código *mientras* se cumpla una condición. Por ejemplo, si queremos imprimir los números del 1 al 10, podemos hacerlo de la siguiente forma:

```
i = 1
while i < 11:
    print(i)
    i += 1
```

```
1
2
3
4
5
6
7
8
9
10
```

El ciclo `while` incluye una línea de *inicialización* y una línea de `<cuerpo>`, que puede tener una o más instrucciones. El ciclo definido es de la forma:

```
while <expresion>:
    <cuerpo>
```

El ciclo se dice *indefinido* porque una vez evaluada la `<expresion>`, **no** se sabe cuántas veces se va a ejecutar el `<cuerpo>`: se ejecuta mientras la `<expresion>` sea verdadera.

Para usar la instrucción `while`, tenemos cuatro aspectos para armar y afinar correctamente:

- Cuerpo
- Condición
- Estado Previo
- Paso

Antes, para la instrucción `for`, sólo considerábamos el cuerpo y la condición. Ahora, además, tenemos que considerar el estado previo y el paso.

El **cuerpo** es la porción de código que se repetirá mientras la condición sea verdadera.
La **condición** es la expresión booleana que se evalúa para decidir si se ejecuta el cuerpo o no.
El **estado previo** es el estado de las variables antes de ejecutar el cuerpo. En general, se refiere al estado de las variables que participan de la condición.
El **paso** es la porción de código que modifica el estado previo. En general, se refiere a la modificación de las variables que participan de la condición.

> **Warning**
>
> Con los ciclos `while` hay que tener mucho cuidado de no caer en un loop infinito. Esto sucede cuando la condición siempre es verdadera, y el cuerpo no modifica el estado previo. Por ejemplo:

```
while True: # más adelante sobre el uso de `while True`
    print("Hola")
```

o bien:

```
i = 0
while i < 10:
    print(i) # el valor de i nunca cambia
```

Ejercicio
Repetir el ejercicio 7.b de la guía 2 usando un ciclo `while`. Repetir usando un ciclo `for`. ¿Qué diferencias hay entre ambos?

3.2.3 Break, Continue y Return

`break` y `continue` son dos palabras clave en Python que se utilizan en bucles (tanto `for` como `while`) para alterar el flujo de ejecución del bucle.

> ⚠ Warning
>
> Si bien son parte del apunte, desrecomendamos fuertemente su uso de forma ligera: el comportamiento de un ciclo while no debería depender ni de break ni de continue. Si es que decidimos usarlos, es porque le estamos dando **funcionalidad adicional** al código. Por ejemplo, si estamos intentando realizar operaciones y podemos encontrarnos con un error. En ese caso, tenemos la posibilidad de ignorar el error (continuar con el bucle) o cortar la ejecución (dejar de iterar). Pero esto lo vamos a ver más adelante.

3.2.3.1 Break

La declaración `break` se usa para salir inmediatamente de un bucle antes de que se complete su iteración normal. Cuando se encuentra una declaración `break` dentro de un bucle, el bucle `for` o `while` se detiene inmediatamente y continúa con la ejecución de las instrucciones que están después del mismo.

Por ejemplo, supongamos que queremos encontrar al primer número múltiplo de 3 entre 10 y 30:

```
numero = 10
while numero <= 30:
  if numero % 3 == 0:
```

```
      print("El primer número múltiplo de 3 es:", numero)
      break
  numero += 1
```

```
El primer número múltiplo de 3 es: 12
```

```
for numero in range(10, 31):
  if numero % 3 == 0:
      print("El primer número múltiplo de 3 es:", numero)
      break
```

```
El primer número múltiplo de 3 es: 12
```

3.2.3.2 Continue

La declaración `continue` se usa para omitir el resto del código dentro de una iteración actual del bucle y continuar con la siguiente iteración. Cuando se encuentra una declaración `continue` dentro de un bucle, el bucle `for` o `while` salta a la siguiente iteración del bucle sin ejecutar las instrucciones que están después del `continue`.

Por ejemplo, supongamos que queremos imprimir todos los números entre 1 y 20, excepto los múltiplos de 4:

```
numero = 1
while numero <= 20:
  if numero % 4 == 0:
      numero += 1
      continue
  print(numero)
  numero += 1
```

```
1
2
3
5
6
7
9
10
11
```

```
13
14
15
17
18
19
```

```
for numero in range(1, 21):
  if numero % 4 == 0:
      continue
  print(numero)
```

```
1
2
3
5
6
7
9
10
11
13
14
15
17
18
19
```

> **Note**
>
> Notemos que tanto para el uso de `break` como de `continue`, si el código se encuentra con uno de ellos en la ejecución, no ejecuta nada posterior a ellos: en el caso de `break`, corta o interrumpe la ejecución del bucle; en el case de `continue`, saltea el resto del código de esa iteración y pasa a la siguiente, volviendo a evaluar la condición si el bucle es `while`. Es por esto que en el último ejemplo no necesitamos un `else`, sino que con sólo tener un `if` alcanza: si se ejecuta el cuerpo del if, nos encontramos con un `continue` y el resto del código no se ejecuta (el print).

3.2.3.3 Return

Cuando estamos dentro de una función, la instrucción `return` nos permite devolver un valor y salir de la función. Ahora, si además estamos dentro de un ciclo, también nos permite salir del mismo sin ejecutar el resto del código.

Por ejemplo:

```
def obtener_primer_par_desde(n):
  for num in range(n, n+10):
    print(f"Analizando si el número {num} es par")
    if num % 2 == 0:
      return num
  return None
```

```
obtener_primer_par_desde(9)
```

```
Analizando si el número 9 es par
Analizando si el número 10 es par
```

```
10
```

Como vemos, la función `obtener_primer_par_desde` recibe un número `n`, y devuelve el primer número par que encuentra a partir de `n`. Si no encuentra ningún número par, devuelve `None`. Si encuentra un número par, no sigue analizando el resto de los números. Usa `return` para salir del ciclo y devuelve el número encontrado.

3.2.4 Consideraciones del While

3.2.4.1 No repitas

Es importante **no** ser redundantes con el código y no "hacer preguntas" que ya sabemos.

```
while <condicion>:
  <cuerpo>

<codigo cuando ya no se cumple la condición>
```

Veamos un ejemplo:

```
numero = 0
while numero < 3:
  print(numero)
  numero += 1

if numero == 3:
  print("El número es 3")
else:
  print("El número no es 3")
```

El output va a ser siempre el mismo:

```
1
2
3
El número es 3
```

¿Por qué? Porque nuestra condición del while es lo que dice *"mientras esto se cumpla, yo repito el bloque del código de adentro"*. Nuestra condición es que `numero < 3`. En el momento en que `numero` llega a 3, el bucle `while`**deja** de cumplir con la condición, y la ejecución se corta, se termina con el bucle.

Es decir, el bloque

```
if numero == 3:
  print("El número es 3")
```

siempre se ejecuta.

Y el bloque

```
else:
  print("El número no es 3")
```

nunca se ejecuta.

Por lo tanto, podemos reescribir el código de la siguiente forma:

```
numero = 0
while numero < 3:
  print(numero)
  numero += 1

print("El número es 3")
```

De la misma forma, no tendría sentido hacer algo así:

```
numero = 0
while numero < 3:
  print(numero)
  numero += 1
  continue

  if numero == 3:
    break
```

1. `if numero == 3` está absolutamente de más. Si `numero` es 3, el bucle `while` **no** se ejecuta, por lo que nunca se va a llegar a esa línea de código. No es necesario "re-chequear" la condición del while dentro del mismo, porque asumimos que si llegamos a esa línea de código, es porque la condición se cumplió. Por lo tanto, podemos reescribir el código de la siguiente forma:

```
numero = 0
while numero < 3:
  print(numero)
  numero += 1
  continue
```

2. Ahora, el continue está de más también, porque se usa cuando nosotros queremos *forzar* a que el ciclo pase a la siguiente iteración. Pero en este caso, el ciclo ya va a pasar a la siguiente iteración, porque estamos en la última línea del cuerpo.

Este es nuestro código final, escrito de forma correcta:

```
numero = 0
while numero < 3:
  print(numero)
  numero += 1
```

3.2.4.2 While True

La instrucción `while` está hecha para que se ejecute mientras la condición sea verdadera. Pero, ¿qué pasa si usamos `while True`? Lo que pasa al usar `while True` es que nuestro código se vuelve más propenso al error: si no tenemos cuidado, podemos caer en un loop infinito.

Como no tenemos una condición a evaluar ni modificar en cada iteración, el bucle se ejecuta infinitamente. Dependería de nosotros, como programadores, que el bucle se corte en algún momento. Es decir, dependería de que nos acordemos de poner dentro del `while` alguna decisión que haga que el bucle se corte. Y si por alguna razón no nos acordamos, el bucle se ejecutaría infinitamente, dejando al programa "congelado" o "colgado", sin responder, y usando todos los recursos de la computadora.

En pocas palabras, podemos afirmar que el uso de `while True` en Python es una **mala práctica de programación**, y en el transcurso de la materia, pedimos **evitar usarla**.

3.2.4.3 Modificando la Condición

```
while <condicion>:
  <cuerpo>
```

La mejor decisión que se puede tomar para el de un bloque `while` es asumir que, durante toda su ejecución exceptuando la última línea, la condición se cumple. Es decir, que el cuerpo del bucle se ejecuta mientras la condición sea verdadera. Por lo tanto, si queremos modificar la condición, debemos hacerlo en la última línea del cuerpo.

Por ejemplo, esto no es correcto:

```
# Se deben imprimir los números 0, 1, 2
numero = 0
while numero < 3:
  numero += 1     # actualización de la condición
  print(numero)
```

```
1
2
3
```

Como vemos, se imprimen los números 1, 2, 3; pero no el 0. Esto es porque estamos modificando la condición ni bien empieza el bucle, y no en la última línea del cuerpo.

La forma correcta de hacerlo sería:

```
# Se deben imprimir los números 0, 1, 2
numero = 0
while numero < 3:
  print(numero)
  numero += 1     # actualización de la condición
```

```
0
1
2
```

De esta forma, todo lo que se encuentre antes de la última línea del cuerpo se ejecuta mientras la condición sea verdadera. Y la última línea del cuerpo es la que modifica la condición.

Una forma genérica, bastante común, de plantear un problema es:

```
<actualizar condición>

while <condición>:
  <hacer algo>
  <actualizar condición/es>
```

Donde `atualizar condición` implica actualizar todas las variables (una o más) relacionadas a la condición del ciclo, y `hacer algo` incluye al comportamiento repetitivo que queremos tener si la condición se cumple.

Por ejemplo: Queremos imprimir un número, empezando de 0, siempre que sea menor a 10.

```
i = 0 # actualizar condición

while i < 10: # condicion
  print(i) # hacer algo
  i += 1 # actualizar condición
```

```
0
1
2
3
4
5
6
7
8
9
```

Ejercicio Desafío

Escribir un programa que pida al usuario un número entero positivo y muestre por pantalla todos los números pares desde 1 hasta ese número.
Resolver primero usando un ciclo `while` y luego usando un ciclo `for`.

Ejercicio Desafío

Escribir un programa que pida al usuario un número par. Mientras el usuario ingrese números que no cumplan con lo pedido, se lo debe volver a solicitar.
Pista: resolver usando `while`.

4 Tipos de Estructuras de Datos

4.1 Introducción: Secuencias

Una secuencia es una serie de elementos ordenados que se suceden unos a otros.

Una secuencia en Python es un grupo de elementos con una organización interna, que se alojan de manera contigua en memoria.

Las secuencias son tipos de datos que pueden ser iterados, y que tienen un orden definido. Las secuencias más comunes son los rangos, las cadenas de caracteres, las listas y las tuplas. En este capítulo vamos a ver las características de cada una de ellas y cómo podemos manipularlas.

4.2 Rangos

Los rangos ya los hemos visto antes, pero lo que no habíamos comentado es que son secuencias. Los rangos representan específicamente una secuencia de números inmutable.

Los rangos se definen con la función `range()`, que recibe como parámetros el inicio, el fin y el paso. El inicio es opcional y por defecto es 0, el paso también es opcional y por defecto es 1.

> **Note**
>
> Para más información de los rangos, ver la unidad 3.

4.3 Cadenas de Caracteres

Un `string` es un tipo de secuencia que sólo admite caracteres como elementos. Los strings son inmutables, es decir, no se pueden modificar una vez creados.

Internamente, cada uno de los caracteres se almacenará de forma contigua en memoria. Es por esto que podemos acceder a cada uno de los caracteres de un string a través de su índice haciendo uso de `[]`.

Índice	0	1	2	3	4	5	6	7	8	9
Letra	H	o	l	a		M	u	n	d	o

Hasta ahora, vimos que:

1. Las cadenas de caracteres pueden ser concatenadas con el operador `+`:

```
saludo = "Hola"
despedida = "Chau"
print(saludo + despedida)
```

```
HolaChau
```

2. Las cadenas de caracteres pueden ser *sliceadas* o incluso acceder a un único elemento usando `[]`:

```
saludo = "Hola Mundo"
print(saludo[0:4])
print(saludo[5])
```

```
Hola
M
```

Podemos agregar también que:

3. Las cadenas de caracteres pueden ser multiplicadas por un número entero (y el resultado es la concatenación de la cadena consigo misma esa cantidad de veces):

```
saludo = "Hola"
print(saludo * 3)
```

```
HolaHolaHola
```

Adicional a esas 3 operaciones, las cadenas de caracteres tienen una gran cantidad de métodos que nos permiten manipularlas. Veamos algunos de ellos.

4.3.1 Métodos de Cadenas de Caracteres

Todos los métodos de las cadenas de caracteres devuelven una nueva cadena de caracteres o un valor, y no modifican la cadena original (ya que las cadenas de caracteres son inmutables).

4.3.1.1 Longitud de una Cadena

Se puede averiguar la cantidad de caracteres que conforman una cadena utilizando la función predefinida `len()`:

```
print(len("Pensamiento Computacional"))
```

```
25
```

Existe también una cadena especial, la cadena vacía (ya la hemos visto antes), que es la cadena que no contiene ningún caracter entre las comillas. La longitud de la cadena vacía es 0.

> Tip: Len e Índices de la Cadena
>
> Es interesante notar lo siguiente: si tenemos una cadena de caracteres de longitud `n`, los índices de la cadena van desde `0` hasta `n-1`. Esto es porque el índice `n` no existe, ya que el primer índice es `0` y el último es `n-1`.
> Veámoslo con un ejemplo: tenemos el caracter `Hola`.

Índice	0	1	2	3
Letra	H	o	l	a

La longitud de la cadena es 4, pero el último índice es 3. Si intentamos acceder al índice 4, nos dará un error:

```
saludo = "Hola"
print(saludo[4])
```

```
IndexError: string index out of range
```

Lo que nos indica el error es que el índice está fuera del rango de la cadena. Esto es porque el índice `4` no existe, ya que el último índice es `3`. El largo de la cadena es `4`, y el último índice disponible es `4-1=3`.

- Los índices positivos (entre `0` y `len(s) - 1`) son los caracteres de la cadena del primero al último.
- Los índices negativos (entre `-len(s)` y `-1`) proveen una notación que hace más fácil indicar cuál es el último caracter de la cadena: `s[-1]` es el último caracter, `s[-2]` es el penúltimo, y así sucesivamente.

```
saludo = "Hola"
print(saludo[-1])
print(saludo[-2])
print(saludo[-3])
print(saludo[-4])
```

```
a
l
o
H
```

Además, el uso de índices negativos también es válido para *slices*:

```
saludo = "Hola"
print(saludo[-3:-1])
```

```
ol
```

Al usar índices negativos, es importante no salirse del rango de los índices permitidos.

4.3.1.2 Recorriendo Cadenas de Caracteres

Dijimos que los strings son secuencias, y por lo tanto podemos iterar sobre ellos. Esto significa que podemos recorrerlos con un ciclo `for`:

```
saludo = "Hola Mundo"
for caracter in saludo:
    print(caracter)
```

```
H
o
l
a

M
u
n
d
o
```

Si bien esto ya lo habíamos nombrado en la sección anterior como una posibilidad, ahora sabemos por qué: todas las secuencias son iterables, y por lo tanto, podemos recorrerlas.

4.3.1.3 Buscando Subcadenas

El operador `in` nos permite saber si una subcadena se encuentra dentro de otra cadena. En la guía de la unidad 3 te pedimos que investigues acerca del operador `in` y `not in` para el ejercicio de vocales y consonantes.

`a in b` es una expresión (¿qué era una expresión?, repasar de ser necesario la unidad 3) que devuelve `True` si `a` es una subcadena de `b`, y `False` en caso contrario.

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

```
True
```

Al ser una expresión booleana, se puede usar como condición tanto de un `if` como de un `while`:

```
if "Hola" in "Hola Mundo":
    print("Se encontró una subcadena!")
```

```
Se encontró una subcadena!
```

Ejercicio

1. Investigar, para un string dado s, cuál es el resultado del slice `s[:]`
2. Investigar, para un string dado s, cuál es el resultado del slice `s[j:]` con j un número entero negativo.

4.3.1.4 Inmutabilidad

Las cadenas son inmutables. Esto significa que no se pueden modificar una vez creadas. Por ejemplo, si queremos cambiar un caracter de una cadena, no podemos hacerlo:

```
saludo = "Hola Mundo"
saludo[0] = "h"
```

```
TypeError: 'str' object does not support item assignment
```

Si queremos realizar una modificación sobre una cadena, lo que tenemos que hacer es crear una nueva cadena con la modificación que queremos:

```
saludo = "Hola Mundo"
saludo = "h" + saludo[1:]
print(saludo)
```

```
hola Mundo
```

4.3.1.5 Otros Métodos de Cadenas de Caracteres

Las cadenas de caracteres tienen una gran cantidad de métodos que nos permiten manipularlas. Algunos ya los vimos, como `len()`, `in` y `not in`. Veamos otros.

> **Recomendación**
>
> Te recomendamos que te animes a probar estas funciones, para ver qué hacen y terminar de entenderlas.

Método	Descripción	Ejemplo
`count(subcadena)`	Devuelve la cantidad de veces que aparece la subcadena en la cadena	`"Hola mundo".count("o")` devuelve `2`
`find(subcadena)`	Devuelve el índice de la primera aparición de la subcadena en la cadena, o `-1` si no se encuentra. Cada vez que se llama, devuelve la primer aparición. Puede recibir un parámetro adicional para buscar a partir de una posición particular.	`"Hola mundo".find("mundo")` devuelve `5`.`"Hola mundo".find("Hola",6)` devuelve `-1`.
`upper()`	Devuelve una copia de la cadena con todos los caracteres en mayúscula	`"Hola mundo".upper()` devuelve `"HOLA MUNDO"`
`lower()`	Devuelve una copia de la cadena con todos los caracteres en minúscula	`"Hola mundo".lower()` devuelve `"hola mundo"`
`replace(subcadena1, subcadena2)`	Devuelve una copia de la cadena reemplazando todas las apariciones de la subcadena1 por la subcadena2	`"Hola mundo".replace("mundo", "amigos")` devuelve `"Hola amigos"`
`split()`	Devuelve una lista de subcadenas separando la cadena por los espacios en blanco	`"Hola mundo  ".split()` devuelve `["Hola", "mundo"]`
`split(separador)`	Devuelve una lista de subcadenas separando la cadena por el separador	`"Hola, mundo".split(", ")` devuelve `["Hola", "mundo"]`
`isdigit()`	Devuelve `True` si todos los caracteres de la cadena son dígitos, `False` en caso contrario	`"123".isdigit()` devuelve `True`
`isalpha()`	Devuelve `True` si todos los caracteres de la cadena son letras, `False` en caso contrario	`"Hola".isalpha()` devuelve `True`

Método	Descripción	Ejemplo
`index(subcadena)`	Devuelve el índice de la primera aparición de la subcadena en la cadena, o produce un error si no se encuentra	`"Hola mundo".index("mundo")` devuelve `5`

4.4 Tuplas

Las tuplas son una secuencia de elementos inmutable. Esto significa que no se pueden modificar una vez creadas. En Python, el tipo de dato asociado a las tuplas se llama `tuple` y se definen con paréntesis `()`:

```
tupla = (1, 2, 3)
```

Las tuplas pueden tener elementos de cualquier tipo, es decir, pueden ser heterogéneas. Por ejemplo, podemos tener una tupla con un número, un string y un booleano:

```
tupla = (1, "Hola", True)
```

Una tupla de un sólo elemento (unitaria) debe definirse de la siguiente manera:

```
tupla = (1,)
```

La coma al final es necesaria para diferenciar una tupla de un número entre paréntesis `(1)`.

Ejemplos de tuplas podrían ser:

- Una fecha, representada como una tupla de 3 elementos: día, mes y año: `(1, 1, 2020)`
- Datos de una persona: `(nombre, edad, dni)`: `("Carla", 30, 12345678)`

Incluso es posible anidar tuplas, como por ejemplo guardar, para una persona, la fecha de nacimiento: `("Carla", 30, 12345678, (1, 1, 1990))`

4.4.1 Tuplas como Secuencias

Como las tuplas son secuencias, al igual que las cadenas, podemos utilizar la misma notación de índices para obtener cada uno de sus elementos y, de la misma forma que las cadenas, los elementos comienzan a enumerarse en su posición desde el `0`:

```
fecha = (1, 12, 2020)
print(fecha[0])
```

```
1
```

También podemos usar la notación de rangos, o *slices*, para obtener subconjuntos de la tupla. Esto es algo típico de las secuencias:

```
fecha = (1, 12, 2020)
print(fecha[0:2])
```

```
(1, 12)
```

4.4.2 Tuplas como Inmutables

Al igual que con las cadenas, las componentes de las tuplas no pueden ser modificadas. Es decir, no puedo cambiar los valores de una tupla una vez creada:

```
fecha = (1, 12, 2020)
fecha[0] = 2
```

```
TypeError: 'tuple' object does not support item assignment
```

4.4.3 Longitud de una Tupla

La longitud de una tupla se puede obtener con la función predefinida `len()`, que devuelve la cantidad de elementos o componentes que tiene esa tupla:

```
fecha = (1, 12, 2020)
print(len(fecha))
```

```
3
```

Una tupla vacía es una tupla que no tiene elementos: `()`. La longitud de una tupla vacía es 0.

> **Ejercicio** Calcular la longitud de la tupla anidada `("Carla", 30, 12345678, (1, 1, 1990))`. ¿Cuántos elementos tiene?

4.4.4 Empaquetado y desempaquetado de tuplas

Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por esos valores. Ejemplo:

```
a = 1
b = 2
c = 3
d = a, b, c

print(d)
```

```
(1, 2, 3)
```

A esto se le llama *empaquetado*.

De forma similar, si se tiene una tupla de largo k, se puede asignar cada uno de los elementos de la tupla a k variables distintas. Esto se llama *desempaquetado*.

```
d = (1, 2, 3)
a, b, c = d

print(a)
print(b)
print(c)
```

```
1
2
3
```

¡Cuidado!

Si estamos desempaquetando una tupla de largo k, pero lo hacemos en una cantidad de variables menor a k, se producirá un error.

```
d = (1, 2, 3)
a, b = d
```

Obtendremos:

```
ValueError: too many values to unpack
o
ValueError: not enough values to unpack
```

4.5 Listas

Las listas, al igual que las tuplas, también pueden usarse para modelar datos compuestos, pero cuya cantidad y valor varían a lo largo del tiempo. Son secuencias *mutables*, y vienen dotadas de una variedad de operaciones muy útiles.

La notación para lista es una secuencia de valores entre corchetes y separados por comas.

```
lista = [1, 2, 3]

lista_vacia = []
```

4.5.1 Longitud de una Lista

La longitud de una lista se puede obtener con la función predefinida `len()`, que devuelve la cantidad de elementos que tiene esa lista:

```
lista = [1, 2, 3]
print(len(lista))
```

```
3
```

4.5.2 Listas como Secuencias

De la misma forma que venimos haciendo con las cadenas y las tuplas, podremos acceder a los elementos de una lista a través de su índice, *slicear* y recorrerla con un ciclo `for`.

```
lista = [1, 2, 3]
print(lista[0])
```

```
1
```

```
lista = ["Civil", "Informática", "Química", "Industrial"]
print(lista[1:3])
```

```
['Informática', 'Química']
```

```
lista = ["Civil", "Informática", "Química", "Industrial"]
for elemento in lista:
    print(elemento)
```

```
Civil
Informática
Química
Industrial
```

4.5.3 Listas como Mutables

A diferencia de las tuplas, las listas son mutables. Esto significa que podemos modificar sus elementos una vez creadas.

- Para cambiar un elemento de una lista, se usa la notación de índices:

```
lista = [1, 2, 3]
lista[0] = 4
print(lista)
```

```
[4, 2, 3]
```

- Para agregar un elemento al final de una lista, se usa el método `append()`:

```
lista = [1, 2, 3]
lista.append(4)
print(lista)
```

```
[1, 2, 3, 4]
```

- Para agregar un elemento en una posición específica de una lista, se usa el método `insert()`:

```
lista = [1, 2, 3]
lista.insert(0, 4)
print(lista)
```

```
[4, 1, 2, 3]
```

El método ingresa el número `4` en la posición `0` de la lista, y desplaza el resto de los elementos hacia la derecha.

```
lista = [1, 2, 3]
lista.insert(1, 3)
print(lista)
```

```
[1, 3, 2, 3]
```

El método ingresa el número `3` en la posición `1` de la lista, y desplaza el resto de los elementos hacia la derecha.

Las listas no controlan si se insertan elementos repetidos, por lo que si queremos exigir unicidad, debemos hacerlo mediante otras herramientas en nuestro código.

- Para eliminar un elemento de una lista, se usa el método `remove()`:

```
lista = [1, 2, 3]
lista.remove(2)
print(lista)
```

```
[1, 3]
```

Remove busca el elemento `2` en la lista y lo elimina. Si el elemento no existe, se produce un error.

Si el valor está repetido, se eliminará la primera aparición del elemento, empezando por la izquierda.

```
lista = [1, 2, 3, 2]
lista.remove(2)
print(lista)
```

```
[1, 3, 2]
```

- Para quitar el último elemento de una lista, se usa el método `pop()`:

```
lista = [1, 2, 3]
lista.pop()
print(lista)
```

```
[1, 2]
```

El método `pop()` devuelve el elemento que se eliminó de la lista. Si la lista está vacía, se produce un error.

```
lista = [1, 2, 3]
elemento = lista.pop()
print(elemento)
```

```
3
```

- Para quitar un elemento de una lista en una posición específica, se usa el método `pop()` con un índice:

```
lista = [1, 2, 3]
lista.pop(1)
print(lista)
```

```
[1, 3]
```

Al igual que antes, el método `pop()` devuelve el elemento que se eliminó de la lista. Si la lista está vacía, se produce un error.

- `extend()` agrega los elementos de una lista al final de otra. Es lo mismo que concatenar dos listas con el operador `+`:

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
lista1.extend(lista2)
print(lista1)
```

```
[1, 2, 3, 4, 5, 6]
```

4.5.4 Referencias de Listas

```
a = [1,2,3,4]
b = a
a.pop()

print(b)
```

```
[1, 2, 3]
```

Se dice que `b` es una referencia a `a`. Esto significa que `b` no es una copia de `a`, sino que es `a` misma. Por lo tanto, si modificamos `a`, también modificamos `b`.

Una forma de crear una copia de una lista es usando el método `copy()`:

```
a = [1,2,3,4]
b = a.copy()
a.pop()

print(b)
```

```
[1, 2, 3, 4]
```

4.5.5 Búsqueda de Elementos en una Lista

- Para saber si un elemento se encuentra en una lista, se puede utilizar el operador `in`:

```
lista = [1, 2, 3]
print(2 in lista)
```

```
True
```

Como vemos, el operador `in` es válido para todas las secuencias, incluyendo tuplas y cadenas.

- Para averiguar la posición de un valor dentro de una lista, usaremos el método `index()`:

```
lista = ["a", "b", "t", "z"]
print(lista.index("t"))
```

```
2
```

Si el valor no se encuentra en la lista, se produce un error.
Si el valor se encuentra repetido, se devuelve la posición de la primera aparición del elemento, empezando por la izquierda.

4.5.6 Iterando sobre Listas

Las listas son secuencias, y por lo tanto podemos iterar sobre ellas. Esto significa que podemos recorrerlas con un ciclo `for`:

```
lista = [1, 2, 3]
for elemento in lista:
    print(elemento)
```

```
1
2
3
```

Esta forma de recorrer elementos usando `for` es utilizable con todos los tipos de secuencias.

4.5.7 Ordenando Listas

Nos puede interesar que los elementos de una lista estén ordenados según algún criterio. Python provee dos operaciones para obtener una lista ordenada a partir de la desordenada.

- `sorted(s)` devuelve una lista ordenada con los elementos de la secuencia `s`. La secuencia `s` no se modifica.

```
lista = [3, 1, 2]
lista_nueva = sorted(lista)

print(lista)
print(lista_nueva)
```

```
[3, 1, 2]
[1, 2, 3]
```

- `s.sort()` ordena la lista `s` en el lugar. Es decir, modifica la lista `s` y no devuelve nada.

```
lista = [3, 1, 2]
lista.sort()

print(lista)
```

```
[1, 2, 3]
```

Tanto el método `sort()` como el método `sorted()` ordenan la lista en orden ascendente. Si queremos ordenarla en orden descendente, podemos usar el parámetro `reverse`:

```
lista = [3, 1, 2]
lista.sort(reverse=True)
print(lista)
```

```
[3, 2, 1]
```

Existe un método `reverse` (no disponible en todos los ambientes de codeo) que invierte la lista sin ordenarla. Una forma de reemplazarlo es usando *slices*, como ya vimos: `lista[::-1]`.

¡Cuidado con los Ordenamientos!

1. Todos los elementos de la secuencia deben ser comparables entre sí. Si no lo son, se producirá un error. Por ejemplo, no se puede ordenar una lista que contenga números y strings.

2. Al ordenar, las letras en minúscula no valen lo mismo que las letras en mayúscula. Si queremos ordenar "hola" y "HOLA" (por ejemplo), tenemos que compararlas convirtiendo todo a minúscula o todo a mayúscula.
   De lo contrario, se ordena poniendo las mayúsculas primero y luego las minúsculas. Es decir, para una lista con los valores `["hola", "HOLA"]`, el ordenamiento será `["HOLA", "hola"]`.

¿Existe una forma mejor de hacerlo? Sí. Usando *keys* de ordenamiento:

```
lista = ["hola", "HOLA"]
lista.sort(key=str.lower)
print(lista)
```

```
['hola', 'HOLA']
```

Lo importante de momento es que sepas que existe esta forma de ordenar. A *key* se le puede pasar una función que se va a aplicar a cada elemento de la lista antes de ordenar. En este caso, la función `str.lower` convierte todo a minúscula antes de intentar ordenar.

4.5.8 Listas anidadas

Las listas también puede estar anidadas, es decir, una lista puede contener a otras listas. Por ejemplo, podemos tener una lista de listas de números:

```
valores = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Aquí *valores* es una lista que contiene 3 elementos, que a su vez son también listas. Entonces, `valores[0]` sería la lista `[1,2,3]`. Si quisiéramos, por ejemplo, acceder al número `2` de dicha lista, tendríamos que volver a acceder al índice `1` de la lista `valores[0]`:

```
valores = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
numero = valores[0][1]
print(numero)
```

```
2
```

Generalización

Este concepto de listas anidadas se puede generalizar a cualquier secuencia anidada. Por ejemplo, una tupla de tuplas, o una lista de tuplas, o una tupla de listas, etc.

```
tupla = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
numero = tupla[1][2]
print(numero)
```

```
6
```

```
lista = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
numero = lista[2][0]
print(numero)
```

```
7
```

```
tupla = ([1, 2, 3], [4, 5, 6], [7, 8, 9])
numero = tupla[0][1]
print(numero)
```

```
2
```

Incluso se puede reemplazar un elemento anidado por otro:

```
tupla = ([1, 2, 3], [4, 5, 6], [7, 8, 9])
tupla[0][1] = 10
print(tupla)
```

```
([1, 10, 3], [4, 5, 6], [7, 8, 9])
```

Esto es válido siempre y cuando el *elemento* a reemplazar esté dentro de una secuencia *mutable*. En el caso de arriba, estamos cambiando el valor de una lista, que se encuentra dentro de la tupla. La tupla no cambia: sigue teniendo 3 listas guardadas.
Si quisiéramos editar una tupla guardada dentro de una lista, no funcionaría:

```
lista = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
lista[0][1] = 10
print(lista)
```

```
TypeError: 'tuple' object does not support item assignment
```

Las listas anidadas suelen usarse para representar matrices. Para ello, se puede pensar que cada lista representa una fila de la matriz, y cada elemento de la lista representa un elemento de la fila. Por ejemplo, la matriz:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

se puede representar como la lista de listas:

```
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

> **!** Ejercicio Desafío
>
> Escribir una función que reciba una cantidad de filas y una cantidad de columnas y devuelva una matriz de ceros de ese tamaño.
> Ejemplo: `matrix(2,3)` devuelve `[[0, 0, 0], [0, 0, 0]]`

Ejemplo Dada una lista de tuplas de dos elementos (precio, producto), desempaquetar la lista en dos listas separadas: una con los precios y otra con los productos.

```
lista = [(100, "Coca Cola"), (200, "Pepsi"), (300, "Sprite")]

precios = []
productos = []

for precio, producto in lista: # Acá estamos desempaquetando: precio, producto
    precios.append(precio)
    productos.append(producto)

print(precios)
print(productos)
```

```
[100, 200, 300]
['Coca Cola', 'Pepsi', 'Sprite']
```

4.6 Listas y Cadenas

Vimos que las cadenas tienen el método `split`, que nos permite separar una cadena en una lista de subcadenas. Por ejemplo:

```
cadena = "Esta es una      cadena con    espacios   varios"
lista = cadena.split()

print(lista)
```

```
['Esta', 'es', 'una', 'cadena', 'con', 'espacios', 'varios']
```

También podemos hacer lo contrario: podemos unir una lista de subcadenas en una cadena usando el método `join`:

```
lista = ["Esta", "es", "una", "cadena", "con", "espacios", "varios"]
cadena = " ".join(lista)

print(cadena)
```

```
Esta es una cadena con espacios varios
```

La sintaxis del método join es:

```
<separador>.join(<lista>)
```

El separador es el caracter que se va a usar para unir los elementos de la lista. En el ejemplo, el separador es un espacio `" "`, pero puede ser cualquier caracter. La lista contiene a las subcadenas que se van a unir.

4.7 Operaciones de las Secuencias

Tanto las cadenas, como las tuplas y las listas son secuencias, y por lo tanto comparten una serie de operaciones que podemos realizar sobre ellas.

Operación	Descripción
`x in s`	Devuelve `True` si el elemento `x` se encuentra en la secuencia `s`, `False` en caso contrario
`s + t`	Concatena las secuencias `s` y `t`
`s * n`	Repite la secuencia `s` `n` veces
`s[i]`	Devuelve el elemento de la secuencia `s` en la posición `i`
`s[i:j:k]`	Devuelve un *slice* de la secuencia `s` desde la posición `i` hasta la posición `j` (no incluída), con pasos de a `k`
`len(s)`	Devuelve la cantidad de elementos de la secuencia `s`
`min(s)`	Devuelve el elemento mínimo de la secuencia `s`
`max(s)`	Devuelve el elemento máximo de la secuencia `s`
`sum(s)`	Devuelve la suma de los elementos de la secuencia `s`
`count(x)`	Devuelve la cantidad de veces que aparece el elemento `x` en la secuencia `s`
`index(x)`	Devuelve el índice de la primera aparición del elemento `x` en la secuencia `s`

> **Tip**
>
> Te recomendamos que pruebes cada una de estas operaciones con las distintas secuencias que vimos en este capítulo.

Además, es posible crear una lista o tupla a partir de cualquier otra secuencia, usando las funciones `list` y `tuple` respectivamente:

```
lista = list("Hola")
print(lista)
```

```
['H', 'o', 'l', 'a']
```

```
tupla = tuple("Hola")
print(tupla)
```

```
('H', 'o', 'l', 'a')
```

```
lista = list( (1, 2, 3) ) # Convertimos una tupla en una lista
print(lista)
```

```
[1, 2, 3]
```

Esta última es particularmente útil cuando necesitamos trabajar con una tupla, pero como son inmutables, la convertimos a lista para manipularla sin problemas.

> **Ejercicio** Escribir una función que le pida al usuario que ingrese números enteros positivos, los vaya agregando a una lista, y que cuando el usuario ingrese un 0, devuelva la lista de números ingresados.

```
def ingresar_numeros():
    numeros = []
    numero = int(input("Ingrese un número: "))

    while numero != 0:
        numeros.append(numero)
        numero = int(input("Ingrese un número: "))
    return numeros
```

> **Ejercicio** Escribir una función que cuente la cantidad de letras que tiene una cadena de caracteres, y devuelva su valor.
> Luego, usar esa función como *key* para ordenar la siguiente lista: `["Año", "Onomatopeya", "Murcielago", "Arañas", "Messi", "Camiseta"]`.

```
def contar_letras(cadena):
    return len(cadena)

lista = ["Año", "Onomatopeya", "Murcielago", "Arañas", "Messi", "Camiseta"]
lista.sort(key=contar_letras)

print(lista)
```

```
['Año', 'Messi', 'Arañas', 'Camiseta', 'Murcielago', 'Onomatopeya']
```

> **!** Ejercicio Desafío
>
> Escribir una función que cuente la cantidad de vocales que tiene una cadena de caracteres, y devuelva su valor. Debe considerar mayúsculas y minúsculas. Pista: podés usar la función para saber si una letra es vocal que hiciste en la unidad 3.
> Luego, usar esa función como *key* para ordenar la siguiente lista: `["Año", "Onomatopeya", "Murcielago", "Arañas", "Messi", "Camiseta"]`.

4.7.1 Map

La función `map` aplica una función a cada uno de los elementos de una secuencia, y devuelve una nueva secuencia con los resultados.

```
def obtener_cuadrado(x):
  return x**2

lista = [1, 2, 3, 4]
lista_cuadrados = list(map(obtener_cuadrado, lista))
print(lista_cuadrados)
```

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

La sintaxis es:

```
map(<funcion>, <secuencia>)
```

La función `map` devuelve un objeto de tipo `map`, por lo que en general lo vamos a convertir a una lista usando `list()`. Sin embargo, el tipo `map` es iterable, por lo que podríamos recorrerlo con un ciclo `for`:

```
for n in lista_cuadrados:
  print(n)
```

```
1
4
9
16
```

> **Tip**
>
> Las funciones a pasar como parámetro a `map` devuelven *valores transformados* del elemento original. Lo que hace `map` es aplicar la función a cada uno de los elementos de la secuencia original.

4.7.2 Filter

La función `filter` aplica una función a cada uno de los elementos de una secuencia, y devuelve una nueva secuencia con los elementos para los cuales la función devuelve `True`.

```
def es_par(x):
  return x % 2 == 0

lista = [1, 2, 3, 4]
lista_pares = list(filter(es_par, lista))
print(lista_pares)
```

```
[2, 4]
```

La sintaxis es:

```
filter(<funcion>, <secuencia>)
```

La función `filter` devuelve un objeto de tipo `filter`, por lo que en general lo vamos a convertir a una lista usando `list()`. Sin embargo, el tipo `filter` es iterable, por lo que podríamos recorrerlo con un ciclo `for`:

```
for n in lista_pares:
  print(n)
```

```
2
4
```

> **Tip**
>
> Las funciones a pasar como parámetro a `filter` devuelven *valores booleanos* del elemento original. Lo que hace `filter` es filtrar la secuencia original y quedarse sólo con los valores para los cuales la función devuelve `True`.

Ejemplo Escribir una función que reciba una lista de números y devuelva una lista con los números positivos de la lista original.

```
def es_positivo(x):
  return x > 0

def quitar_negativos_o_cero(lista):
  return list(filter(es_positivo, lista))

lista = [1, -2, 3, -4, 5, 0]
lista_positivos = quitar_negativos_o_cero(lista)
print(lista_positivos)
```

```
[1, 3, 5]
```

Ejemplo Escribir una función que reciba una lista de nombres y devuelva una lista con los mismos nombres pero con la primer letra en mayúscula.

```
def capitalizar_nombre(nombre):
  return nombre.capitalize()

def capitalizar_lista(lista):
  return list(map(capitalizar_nombre, lista))

lista = ["pilar", "barbie", "violeta"]
lista_capitalizada = capitalizar_lista(lista)
print(lista_capitalizada)
```

```
['Pilar', 'Barbie', 'Violeta']
```

> **Note**
>
> Tanto `map` como `filter` son aplicables a cualquiera de las secuencias vistas (rangos, cadena de caracteres, listas, tuplas).

> **Ejercicio Desafío**
>
> Se está procesando una base de datos para entrenar un modelo de Machine Learning. La base de datos contiene información de personas, y cada persona está representada por una tupla de 2 elementos: nombre, edad.
> Escribir una función que reciba una lista de estas tuplas. La función debe devolver la lista ordenada por edad; y filtrada de forma que sólo queden los nombres de las personas mayores de edad (>18). Además, los nombres deben estar en mayúscula.
> Ejemplo:
> Si se tiene `[("sol", 40), ("priscila", 15), ("agostina", 30)]`
> una vez ejecutada, la función debe devolver: `[("AGOSTINA",30), ("SOL",40)]`

4.8 Diccionarios

Un diccionario es una colección de pares clave-valor. Es una estructura de datos que nos permite guardar información de forma organizada, y acceder a ella de forma eficiente. Cada clave está asociada a un valor determinado.

Figure 4.1: Diccionario cuyas claves son dominios de internet (.ar, .es, .tv) y cuyos valores asociados son los países correspondientes.

Las claves deben ser únicas, es decir, no puede haber dos claves iguales en un mismo diccionario. Los valores pueden repetirse. Si se asigna un valor a una clave ya existente, se reemplaza el valor anterior.

Podemos acceder a un valor a través de su clave porque las claves son únicas, pero no a la inversa. Es decir, no podemos acceder a una clave a través de su valor, porque los valores pueden repetirse y podría haber varias claves asociadas al mismo valor.

Además, los diccionarios no tienen un orden interno particular. Se consideran entonces iguales dos diccionarios si tienen las mismas claves asociadas a los mismos valores, independientemente del orden en que se hayan agregado.

Al igual que las listas, los diccionarios son mutables. Esto significa que podemos modificar sus elementos una vez creados.

- Cualquier valor de tipo inmutable puede ser `clave` de un diccionario: cadenas, enteros tuplas.
- No hay restricciones para los `valores`, pueden ser de cualquier tipo: cadenas, enteros, tuplas, listas, otros diccionarios, etc.

4.8.1 Diccionarios en Python

Para definir un diccionario, se utilizan llaves `{}` y se separan las claves de los valores con dos puntos `:`. Cada par clave-valor se separa con comas `,`.

```
dominios = {"ar": "Argentina", "es": "España", "tv": "Tuvalu"}
```

El tipo asociado a los diccionarios es `dict`:

```
print(type(dominios))
```

```
<class 'dict'>
```

Para declararlo vacío y luego ingresar valores, se lo declara como un par de llaves vacías. Luego, haciendo uso de la notación de corchetes `[]`, se le asigna un valor a una clave:

```
materias = {}
materias["lunes"] = [6103, 7540]
materias["martes"] = [6201]
materias["miércoles"] = [6103, 7540]
materias["jueves"] = []
materias["viernes"] = [6201]
```

En el código de arriba, se está creando una variable `materias` de tipo `dict`, y se le están asignando valores a las claves `"lunes"`, `"martes"`, `"miércoles"`, `"jueves"` y `"viernes"`. Los valores asociados a cada clave son listas con los códigos de las materias que se dan esos días. El diccionario se ve algo así:

```
{
    "lunes": [6103, 7540],
    "martes": [6201],
    "miércoles": [6103, 7540],
    "jueves": [],
    "viernes": [6201]
}
```

4.8.2 Accediendo a los Valores de un Diccionario

Para acceder a los valores de un diccionario, se utiliza la notación de corchetes `[]` con la clave correspondiente:

```
cods_lunes = materias["lunes"]
print(cods_lunes)
```

```
[6103, 7540]
```

Veamos que la clave “lunes” no va a ser igual a la clave “Lunes” o “LUNES”, porque como ya dijimos antes, Python es *case sensitive*.

> ⚠ ¡Cuidado! Acceso a Claves que no Existen
>
> Si intentamos acceder a una clave que no existe en el diccionario, se produce un error:
>
> ```
> print(materias["sábado"])
> ```
>
> ```
> Traceback (most recent call last):
> File "<stdin>", line 1, in <module>
> KeyError: 'sábado'
> ```

Para evitar tratar de acceder a una clave que no existe, podemos verificar si una clave se encuentra o no en el diccionario haciendo uso del operador `in`:

```
if "sábado" in materias:
    print(materias["sábado"])
else:
    print("No hay clases el sábado")
```

```
No hay clases el sábado
```

También podemos usar la función `get`, que recibe una clave `k`y un valor por omisión `v`, y devuelve el valor asociado a la clave `k`, en caso de existir, o el valor `v` en caso contrario.

```
print(materias.get("sábado", "Error de clave: sábado"))
```

```
Error de clave: sábado
```

```
print(materias.get("domingo",[]))
```

```
[]
```

Como vemos el valor por omisión puede ser de cualquier tipo.

4.8.3 Iterando Elementos del Diccionario

4.8.3.1 Por Claves

Para iterar sobre las claves de un diccionario, podemos usar un ciclo `for`:

```
for dia in materias:
    print(f"El {dia} tengo que cursar las materias {materias[dia]}")
```

```
El lunes tengo que cursar las materias [6103, 7540]
El martes tengo que cursar las materias [6201]
El miércoles tengo que cursar las materias [6103, 7540]
El jueves tengo que cursar las materias []
El viernes tengo que cursar las materias [6201]
```

También podemos obtener las claves del diccionario como una lista usando el método `keys()`:

```
for dia in materias.keys():
    print(f"El {dia} tengo que cursar las materias {materias[dia]}")
```

```
El lunes tengo que cursar las materias [6103, 7540]
El martes tengo que cursar las materias [6201]
El miércoles tengo que cursar las materias [6103, 7540]
El jueves tengo que cursar las materias []
El viernes tengo que cursar las materias [6201]
```

4.8.3.2 Por Valores

Para iterar sobre los valores de un diccionario, podemos usar el método `values()`:

```
for codigos in materias.values():
    print(codigos)
```

```
[6103, 7540]
[6201]
[6103, 7540]
[]
[6201]
```

Nótese que en este último ejemplo, no podemos obtener la clave a partir de los valores. Por eso no imprimimos los `días`.

4.8.3.3 Por Clave-Valor

Para iterar sobre los pares clave-valor de un diccionario, podemos usar el método `items()`, que nos devuelve un conjunto de tuplas donde el primer elemento de cada una es una clave y el segundo, su valor asociado `(clave,valor)`:

```
for tupla in materias.items():
  dia = tupla[0]
  codigos = tupla[1]
  print(f"El {dia} tengo que cursar las materias {codigos}")
```

```
El lunes tengo que cursar las materias [6103, 7540]
El martes tengo que cursar las materias [6201]
El miércoles tengo que cursar las materias [6103, 7540]
El jueves tengo que cursar las materias []
El viernes tengo que cursar las materias [6201]
```

También podemos desempaquetar las tuplas como vimos previamente:

```
for dia, codigos in materias.items():
  print(f"El {dia} tengo que cursar las materias {codigos}")
```

```
El lunes tengo que cursar las materias [6103, 7540]
El martes tengo que cursar las materias [6201]
El miércoles tengo que cursar las materias [6103, 7540]
El jueves tengo que cursar las materias []
El viernes tengo que cursar las materias [6201]
```

> **Note**
>
> Como los diccionarios no son secuencias, no tienen orden interno específico, por lo que no podemos obtener porciones de un diccionario usando *slices* o `[:]` como hacíamos con otras estructuras de datos.

> **Acerca de la Iteración de un Diccionario**
>
> El mayor beneficio de los diccionarios es que podemos acceder a sus valores de forma eficiente, a través de sus claves.
> Si la única funcionalidad que necesitamos de un diccionario es iterarlo, entonces no estamos aprovechando su potencial. En ese caso, es preferible usar una o más listas o tuplas,

que es más simple y más eficiente.
Iterar un diccionario es una funcionalidad adicional que nos brinda Python, pero no es su principal uso.

4.8.4 Usos de un Diccionario

Los diccionarios son muy versátiles. Se puede utilizar un diccionario para, por ejemplo, contar cuántas apariciones de cada palabra hay en un texto, o cuántas apariciones por cada letra.

También se puede usar un diccionario para tener una agenda de contactos, donde la clave es el nombre de la persona y el valor el número de teléfono.

Hashmaps: Dato interesante

Los diccionarios de Python son implementados usando una estructura de datos llamada *hashmap*.
Para cada clave, se le calcula un valor numérico llamado *hash*, que es el que se usa para acceder al valor asociado a esa clave.
Cuando se recibe una clave, se le calcula su *hash* y se busca en el diccionario el valor asociado a ese *hash*.

4.8.5 Operaciones de los Diccionarios

Operación	Descripción
`d[k]`	Devuelve el valor asociado a la clave `k`
`d[k] = v`	Asigna el valor `v` a la clave `k`. Si la clave no existe, la agrega al diccionario. Si ya existe, le actualiza el valor asociado.
`del d[k]`	Elimina la clave `k` y su valor asociado del diccionario `d`
`k in d`	Devuelve `True` si la clave `k` se encuentra en el diccionario `d`, `False` en caso contrario
`len(d)`	Devuelve la cantidad de pares clave-valor del diccionario `d`
`d.keys()`	Devuelve una lista con las claves del diccionario `d`
`d.values()`	Devuelve una lista con los valores del diccionario `d`

Operación	Descripción
`d.items()`	Devuelve una lista de tuplas con los pares clave-valor del diccionario `d`
`d.copy()`	Devuelve una copia del diccionario `d`
`d.pop(k)`	Elimina la clave `k` y su valor asociado del diccionario `d`, y devuelve el valor asociado
`d.get(k, v)`	Devuelve el valor asociado a la clave `k` si la clave existe, o el valor `v` en caso contrario

> **Note**
>
> Existen más métodos de diccionarios, pero estos son los más utilizados y los que vamos a ver en la materia. Recomendamos que pruebes cada uno de ellos con los diccionarios que vimos en este capítulo.

4.8.6 Diccionarios y Funciones

Los diccionarios son mutables, por lo que podemos pasarlos como parámetros a funciones y modificarlos dentro de la función.

```
def agregar_alumno(alumnos, nombre, legajo):
  alumnos[nombre] = legajo

alumnos = {}
agregar_alumno(alumnos, "Juan", 1234)
agregar_alumno(alumnos, "María", 5678)
print(alumnos)
```

```
{'Juan': 1234, 'María': 5678}
```

4.8.7 Ordenamiento de Diccionarios

Tenemos algunas operaciones que nos permiten ordenar un diccionario:

Operación	Descripción
`dict()`	Crea un diccionario vacío
`sorted(d)`	Devuelve una lista ordenada con las claves del diccionario `d`

Operación	Descripción
`dict(sorted(d.items()))`	Devuelve un diccionario ordenado con las claves del diccionario `d`

Si lo que necesitamos es ordenar diccionarios entre sí (por ejemplo, teniendo una lista de diccionarios), vamos a usar el parámetro `key` de la función `sorted`:

```
def obtener_nombre(alumno):
  return alumno["nombre"]

alumnos = [
  {"nombre": "Priscila", "legajo": 1234},
  {"nombre": "Iara", "legajo": 5678},
  {"nombre": "Agostina", "legajo": 9012}
]
alumnos_ordenados = sorted(alumnos, key=obtener_nombre)
print(alumnos_ordenados)
```

```
[{'nombre': 'Agostina', 'legajo': 9012}, {'nombre': 'Iara', 'legajo': 5678}, {'nombre': 'Pris
```

Note

En el ejemplo de arriba, estamos ordenando una lista de diccionarios por el valor de la clave `"nombre"`.
El parámetro `key` recibe una función que se va a aplicar a cada elemento de la lista antes de ordenar. En este caso, la función `obtener_nombre` devuelve el valor de la clave `"nombre"` de cada diccionario, y es lo que se usa para ordenar.

Ejercicio Desafío

Escribir una función que reciba una lista de diccionarios y una clave, y devuelva una lista con los diccionarios ordenados según la clave.

Ejemplo:
Si se tiene `[{"nombre": "Priscila", "legajo": 1234}, {"nombre": "Iara", "legajo": 5678}, {"nombre": "Agostina", "legajo": 9012}]` y se recibe la clave `nombre`
una vez ejecutada, la función debe devolver:
`[{"nombre": "Agostina", "legajo": 9012}, {"nombre": "Iara", "legajo": 5678}, {"nombre": "Priscila", "legajo": 1234}]`

5 Entrada y Salida

5.1 Archivos

Cuando un programa se está ejecutando los datos están en la memoria, pero cuando el programa termina los datos se pierden.

Para almacenar los datos de forma permanente se hace uso de **archivos**. Cada archivo se identifica con un nombre único dentro de directorio o carpeta en que se encuentre. Por ejemplo dentro la carpeta *Documentos* puede existir solo un archivo con el nombre *Apuntes.txt*.

Los archivos se utilizan para organizar los datos e intercambiarlos para distintos fines. El modo de trabajar con archivos es como trabajar con libros, se pueden abrir, leer, escribir y cerrar. Además se puede leer en orden o secuencialmente o yendo a un lugar específico.

> **Note**
>
> Toda la organización de las computadoras está basada en archivos y directorios.

5.1.1 Abriendo un Archivo

En python para abrir un archivo utilizamos la función `open`

```
ruta_archivo = "alumnos.txt"
archivo = open(ruta_archivo)
```

Esta función intentara abrir el archivo “alumnos.txt” y si tiene éxito en la variable archivo quedara un tipo de dato que nos á manipularlo.

5.1.2 Leyendo un Archivo

La operación más frecuente con los archivos es leerlos de forma secuencial

```
archivo = open(ruta_archivo)
línea = archivo.readline()

while línea != '':
  # hacer algo con la línea
  línea = archivo.readline()

archivo.close()
```

Este último bloque de código lee todas las líneas (renglones) del archivo hasta que no queden más.

La variable `archivo`, que mencionamos más arriba como un "tipo de dato que nos permitirá manipularlo" guarda cuál es la siguiente posición que debe leer y cuando se ejecuta `archivo.readline()` lee esa posición y avanza una posición más.

La función `close()` cierra el archivo, esta operación es importante para mantener la consistencia de la información. Volveremos más adelante sobre este tema.

Ejemplo "alumnos.txt"

```
DNI;Nombre;Nota
45000001;Mariana Szischik;9
46000001;Emilia Duzac;8
46000001;Lucia Capon;9
```

En el ejemplo anterior leímos el archivo línea por línea, pero existe otra forma de leer un archivo. Veamos otro ejemplo.

```
archivo = open(ruta_archivo)
líneas = archivo.readlines()
archivo.close()

for línea in líneas:
  # hacer algo con la línea
  print(línea)
```

```
línea número 0
línea número 1
línea número 2
línea número 3
línea número 4
```

¿Que diferencias hay entre el ejemplo de más arriba y éste?

La diferencia principal y que condiciona el resto de los cambios es que en lugar de leer línea por línea utilizamos la función `readlines()`. Esta función lee *todo* el contenido del archivo y devuelve una lista donde cada elemento de la lista es un renglón. Por otro lado se llama a la función `close()` inmediatamente después de leer todo el archivo. ¿Por qué? ¿Te animás a analizar todas las diferencias?

5.1.2.1 Resumen

- `read()`: Lee todo el archivo y lo devuelve como una cadena de texto.
- `readline()`: Lee una línea del archivo y la devuelve como una cadena de texto. Cuando se llega al final del archivo, devuelve una cadena vacía.
- `readlines()`: Lee todas las líneas del archivo y las devuelve como una lista de cadenas de texto.

5.1.3 Escribiendo en un archivo

Python también tiene métodos para escribir archivos, los más comunes son:

- `write()`: Escribe una cadena de texto en el archivo.
- `writelines()`: Escribe una lista de cadenas de texto en el archivo.

```
ruta_archivo_nuevo = "saludo.txt"
archivo = open(ruta_archivo_nuevo, 'w')

archivo.write("Hola!\n")
archivo.writelines(["¿Cómo estás?\n", "Espero que bien.\n"])
archivo.close()
```

> **Tip**
>
> En este ejemplo se puede ver el uso de `\n`. Este caracter es lo que indica a los medios de salida de información que lo que se escribió finaliza con una nueva línea. Ninguno de los métodos de escritura agrega automáticamente un salto de línea al final de lo que se escribe, a menos que se lo indiquemos explícitamente.
> Cuando leemos un archivo tenemos que tener en cuenta que el último caracter de cada línea va a ser `\n`

Pero no todos los archivos pueden ser escritos, por ejemplo los archivos que se encuentran en modo lectura ('r'). ¿De qué depende? Depende del tipo de acceso con el que se abrió el archivo.

5.1.4 Tipos de acceso

Cuando se abre un archivo hay que especificar para qué lo estamos abriendo, las opciones en general son: leer o escribir. Por defecto, si no especificamos nada, tal como vimos en los ejemplos anteriores, se abre para leer.

operación/modo	r	w	a	r+	w+
leer	si	no	no	si	si
escribir	no	si	si	si	si
posición inicial	inicio	inicio	fin	inicio	inicio
observaciones	Si el archivo no existe da error	Si el archivo existe borra el contenido, sino lo crea	Si el archivo no existe lo crea	Si el archivo no existe da error	Si el archivo existe borra el contenido, sino lo crea
caso de uso	Leer un archivo	Iniciar un nuevo archivo	Agregar más lineas a un archivo existente	Agregar, editar y leer	Agregar, editar y leer

Figure 5.1: Resumen de los tipos de acceso con los que se puede abrir un archivo.

Veamos ejemplos de los casos más comunes

Ejemplo Write (w)

```
ruta_archivo_nuevo = "alumnos_nuevo.txt"
archivo = open(ruta_archivo_nuevo, 'w')

for x in range(5):
  # hacer algo con la línea
  archivo.write(f"línea número {x} \n")

archivo.close()
```

Ejemplo Read (r)

```
archivo = open(ruta_archivo_nuevo, 'r')
líneas = archivo.readlines()
archivo.close()

for línea in líneas:
```

```
  # hacer algo con la línea
  print(línea)
```

Ejemplo Append (a)

```
archivo = open(ruta_archivo_nuevo, 'a')
archivo.write("línea número 5 \n") # agrega una nueva línea al final del archivo
archivo.close()
```

5.1.5 Close

Al terminar de trabajar con un archivo, es importante cerrarlo, por diversos motivos: en algunos sistemas los archivos sólo pueden ser abiertos de a un programa por la vez; en otros, lo que se haya escrito no se guardará realmente hasta no cerrar el archivo.

```
archivo = open(ruta_archivo)
líneas = archivo.readlines()
archivo.close()
```

> **Warning**
>
> Cuando abrimos un archivo, queremos dejarlo abierto siempre la **menor cantidad de tiempo posible**. Si abrimos un archivo y no lo cerramos, estamos ocupando recursos del sistema que podrían ser utilizados por otros programas. Por lo que tenemos que pensar muy bien la forma de armar nuestro código para que los archivos se abran y cierren sólo cuando los necesitamos usar.

Una forma de asegurarse de que un archivo se cierre es utilizar la sentencia `with`. Esta sentencia se encarga de cerrar el archivo automáticamente al finalizar el bloque de código que se le pasa.

```
with open(ruta_archivo) as archivo:
  líneas = archivo.readlines()

# Acá el archivo ya se cerró sólo
```

> **Tip**
>
> ¿Te animás a probar que pasa si intentas escribir en un archivo que fue abierto para lectura ('r') y a leer en uno que fue abierto para escritura ('w')?

5.1.6 Ejemplos

Veamos un ejemplo en el que trabajaremos con dos archivos.

> **Ejemplo** Obtener el promedio de un archivo de notas recibido por parámetro y guardarlo en un nuevo archivo llamado "promedio.txt"

Ejemplo de notas.txt:

```
4
7
10
5
```

```
# Abrimos el archivo de notas
def calcular_guardar_promedio(ruta_notas): # La función puede recibir "notas.txt"
  archivo = open(ruta_notas, 'r')
  líneas = archivo.readlines()
  archivo.close()

  # Leemos línea por línea cada nota
  suma_notas = 0
  cantidad_notas = 0
  for línea in líneas[1:]: # la primer línea no contiene datos, solo los nombres de los campo
    nota = línea.split(";")[2].strip('\n') # nos quedamos con la nota
    suma_notas += int(nota)
    cantidad_notas += 1

  # Guardamos el promedio en un nuevo archivo
  ruta_archivo_promedios = "promedio.txt"
  archivo = open(ruta_archivo_promedios, 'w')
  archivo.write(str(suma_notas/cantidad_notas))
  archivo.close()
```

Otra forma de resolverlo podría haber sido:

```
# Abrimos el archivo de notas
def calcular_guardar_promedio(ruta_notas): # La función puede recibir "notas.txt"
  archivo = open(ruta_notas, 'r')
  líneas = archivo.readlines()
  archivo.close()
```

```
  # Leemos línea por línea cada nota
  notas = []
  for línea in líneas[1:]: # la primer línea no contiene datos, solo los nombres de los campo
    nota = línea.split(";")[2].strip('\n') # nos quedamos con la nota
    notas.append(int(nota))

  # Guardamos el promedio en un nuevo archivo
  ruta_archivo_promedios = "promedio.txt"
  archivo = open(ruta_archivo_promedios, 'w')
  archivo.write(str(sum(notas)/len(notas)))
  archivo.close()
```

Veamos el contenido del archivo "promedio.txt"

```
ruta_archivo = "promedio.txt"
archivo = open(ruta_archivo, 'r')
línea = archivo.readline() # o read
archivo.close()
print(línea)
```

```
6.5
```

En el ejemplo anterior hay al menos dos cosas que vale la pena remarcar: el uso de la función `split()`[1] nos permite separar cada línea en una lista que tiene 3 elementos, a nosotros nos interesa el elemento que está en la posición 2, la nota; por otro lado también utilizamos la función `strip()`[2], esto remueve el caracter de nueva línea `\n` y nos permite leer la nota como un número.

Un detalle que no hay que evadir cómo se recorre la lista teniendo en cuenta que la primer línea del archivo no nos interesa, ya que contiene los nombres de cada campo. Esto se explica en unidad 4.

5.1.7 Tipos de archivos

En la sección anterior utilizamos para todos los archivos la extensión '.txt' el uso de extensiónes es una **convención**, una manera de nombrar las cosas que nos da una idea de lo que hay en el contenido del archivo.

Comúnmente a los archivos que estuvimos usando como ejemplo se los nombra con la extensión '.csv' las siglas de "comma separated values"[3].

[1] Split
[2] Strip
[3] CSV

5.1.8 Conclusiones

- Para utilizar un archivo desde un programa, es necesario abrirlo, y cuando ya no se lo necesite, se lo debe cerrar.
- Las instrucciones más básicas para manejar un archivo son leer y escribir.
- Los archivos de texto se procesan generalmente línea por línea y sirven para intercambiar información entre diversos programas o entre programas y humanos.

5.2 Colab y Archivos

Para usar y crear archivos en Colab, se debe ingresar al último item del menú derecho, que tiene forma de carpeta.

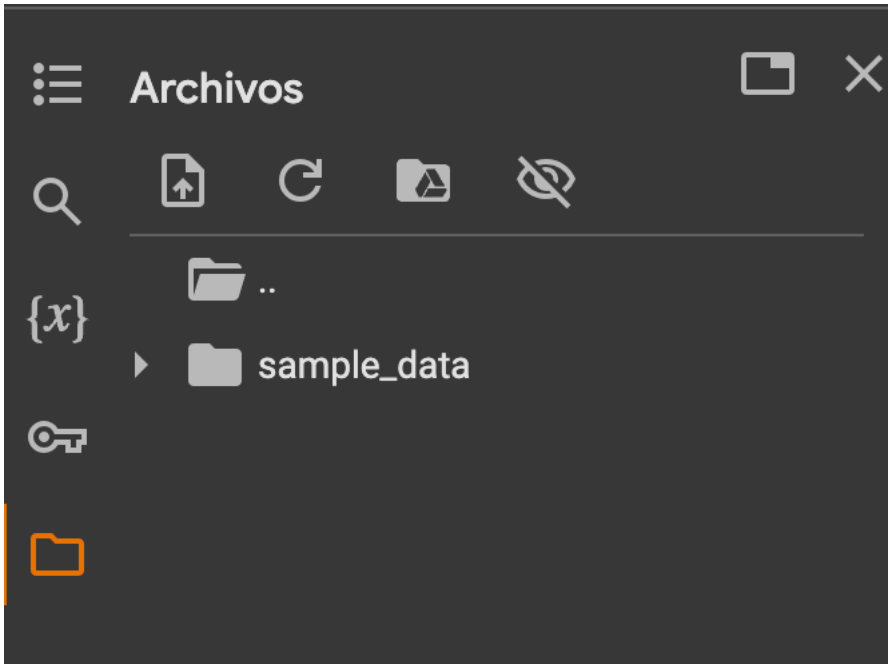


Figure 5.2: Menu Derecho

Allí, con el menú interno de archivos vamos a poder:

- Subir un archivo que tengamos en la computadora / celular
- Refrescar los archivos para ver el contenido actualizado
- Subir un archivo desde Drive
- Ver archivos ocultos (no vamos a usar esto)