Pensamiento Computacional

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

2023-12-27

Table of contents

Pensamiento Computacional **6**
Docentes de la Cátedra 6

La Materia **8**
Fundamentación 8
Objetivos Generales 8

Regimen de Cursada, Calendario y Cursos **10**
Calendario y Cursos 10
Formas de Evaluación 10
Aprobación de la Cursada/Materia 10
Regularización 11
Promoción 11
Examen Final Integrador 11

Videos y Diapositivas **13**
Taller de Herramientas 13

1 Introducción a la Algoritmia y a la Programación **14**
1.1 Introducción 14
1.1.1 La Computadora 14
1.1.2 Software y Hardware 14
1.1.3 Sistema Operativo 15
1.1.4 Algoritmo 15
1.1.5 Programa 16
1.1.6 Lenguaje de Programación 16
1.1.7 Entorno de Desarrollo 16
1.2 Lenguaje Python 16
1.2.1 Hola, Mundo! 17
1.3 Anexo: Google Colab 17
1.3.1 Cómo usar Google Colab 17

2 Tipos de Datos, Expresiones y Funciones **22**
2.1 Sentencias Básicas 22
2.1.1 Flujo de Control de un Programa 22
2.1.2 Valores y Tipos 23

2.1.3 Variables . . . 25
2.1.4 Funciones . . . 26
2.1.5 Ingreso de Datos por Consola . . . 30
2.2 Buenas Prácticas de programación . . . 33
2.2.1 Sobre Comentarios . . . 33
2.2.2 Sobre Convención de Nombres . . . 33
2.2.3 Sobre Ordenamiento de Código . . . 34
2.2.4 Sobre uso de Parámetros en Funciones . . . 34
2.3 Tipos de Datos . . . 36
2.3.1 Datos Simples . . . 36
2.3.2 Operadores Numéricos . . . 37
2.3.3 Operadores de Texto . . . 38
2.3.4 Input y Casteo . . . 39
2.4 Bonus Track: Algunas Funciones Predefinidas de Python . . . 42

3 Estructuras de Control **44**
3.1 Decisiones . . . 44
3.1.1 Expresiones Booleanas . . . 44
3.1.2 Expresiones de Comparación . . . 45
3.1.3 Operadores Lógicos . . . 46
3.1.4 Comparaciones Simples . . . 48
3.1.5 Múltiples decisiones consecutivas. . . . 51
3.2 Ciclos y Rangos . . . 55
3.2.1 Ciclo `for` . . . 56
3.2.2 Ciclo `while` . . . 59
3.2.3 Break, Continue y Return . . . 61
3.2.4 Consideraciones del While . . . 64

4 Tipos de Estructuras de Datos **70**
4.1 Introducción: Secuencias . . . 70
4.2 Rangos . . . 70
4.3 Cadenas de Caracteres . . . 70
4.3.1 Métodos de Cadenas de Caracteres . . . 71
4.4 Tuplas . . . 77
4.4.1 Tuplas como Secuencias . . . 78
4.4.2 Tuplas como Inmutables . . . 78
4.4.3 Longitud de una Tupla . . . 78
4.4.4 Empaquetado y desempaquetado de tuplas . . . 79
4.5 Listas . . . 80
4.5.1 Longitud de una Lista . . . 80
4.5.2 Listas como Secuencias . . . 81
4.5.3 Listas como Mutables . . . 81
4.5.4 Referencias de Listas . . . 84

4.5.5 Búsqueda de Elementos en una Lista . . . 84
4.5.6 Iterando sobre Listas . . . 85
4.5.7 Ordenando Listas . . . 85
4.5.8 Listas anidadas . . . 87
4.6 Listas y Cadenas . . . 89
4.7 Operaciones de las Secuencias . . . 90
4.7.1 Map . . . 92
4.7.2 Filter . . . 93
4.8 Diccionarios . . . 96
4.8.1 Diccionarios en Python . . . 97
4.8.2 Accediendo a los Valores de un Diccionario . . . 97
4.8.3 Iterando Elementos del Diccionario . . . 99
4.8.4 Usos de un Diccionario . . . 101
4.8.5 Operaciones de los Diccionarios . . . 101
4.8.6 Diccionarios y Funciones . . . 102
4.8.7 Ordenamiento de Diccionarios . . . 102

5 Entrada y Salida **104**
5.1 Archivos . . . 104
5.1.1 Abriendo un Archivo . . . 104
5.1.2 Leyendo un Archivo . . . 104
5.1.3 Escribiendo en un archivo . . . 106
5.1.4 Tipos de acceso . . . 107
5.1.5 Close . . . 108
5.1.6 Ejemplos . . . 109
5.1.7 Tipos de archivos . . . 110
5.1.8 Conclusiones . . . 111
5.2 Colab y Archivos . . . 111
5.3 Manejo de errores . . . 113
5.3.1 Validaciones . . . 115
5.3.2 Varias Excepciones . . . 116
5.3.3 Conclusiones . . . 116
5.3.4 Bonus Track: Tipos de Errores . . . 117

6 Bibliotecas de Python **118**
6.1 Introducción . . . 118
6.1.1 ¿Cómo se utilizan las bibliotecas? . . . 118
6.2 Adaptaciones para Colab . . . 119
6.3 NumPy . . . 119
6.3.1 **Arrays** . . . 120
6.3.2 Operaciones aritméticas utilizando **array** . . . 126
6.4 Pandas . . . 135
6.4.1 Serie . . . 136

6.4.2 DataFrame . . . 138
6.4.3 Conclusiones . . . 156
6.5 Matplotlib . . . 156
6.5.1 Introducción . . . 156
6.5.2 Creando una figura . . . 157
6.5.3 Títulos . . . 163
6.5.4 Referencias . . . 164
6.5.5 Gráficos múltiples . . . 165
6.5.6 Uniendo Bibliotecas . . . 167
6.5.7 Bonus Track: Personalización (opcional) . . . 170

Guía de Ejercicios **174**
Recomendaciones al realizar las guías . . . 174
Guía 1: Introducción a la Algoritmia y la Programación . . . 175
Guía 2: Tipos de Datos, Expresiones y Funciones . . . 176
Guía 3: Estructuras de Control . . . 178
1. Decisiones . . . 178
2. Ciclos . . . 180
Guía 4: Tipos de Estructuras de Datos . . . 184
1. Cadenas de caracteres . . . 184
2. Rangos, Tuplas y Listas . . . 186
3. Diccionarios . . . 191
Guía 5: Entrada y Salida . . . 195
1. Archivos . . . 195
2. Manejo de Errores . . . 199

Guía 6: Bibliotecas de Python **202**
1. Numpy . . . 202
2. Pandas . . . 204
3. Matplotlib . . . 207

Contacto **212**
Discord . . . 212
Dejanos Feedback! . . . 212
Ser Docente . . . 212

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
- Bulacios, Juan
- Cabibbo Arteaga, Nehuén Daniel
- Cáceres, Fernando
- Capón, Lucía
- Corti, Bautista
- Duchen, Leonardo
- Duzac, Emilia
- Fulco, Victoria
- Juarez Goldemberg, Mariana
- Lourengo Caridade, Lucía Gabriela
- Maciel, Laura
- Maxwell, Julian
- Notari, Pablo
- Ortielli, Bruno
- Pratto, Florencia
- Ponti, Julieta
- 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

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.

Formas de Evaluación

La cursada de la materia cuenta con dos parciales:

- Primer Parcial
  - Unidad 1
  - Unidad 2
  - Unidad 3
  - Unidad 4 (rangos, cadenas, tuplas y listas)
- Segundo Parcial
  - Unidad 4 (diccionarios)
  - 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.

Videos 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.

Las clases teóricas son virtuales y asincrónicas. Los videos se encuentran en el canal de Youtube de la materia, cada uno con su lista de reproducción correspondiente.

Las clases prácticas son presenciales. Las diapositivas usadas en la práctica se encuentran en siguiente link y están organizadas por curso.

Taller de Herramientas

La materia cuenta con un "Taller de Herramientas" en el canal de Youtube.

En esta lista de videos, se explica cómo utilizar las herramientas de la materia como Discord, la página oficial y Google Colab.

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:

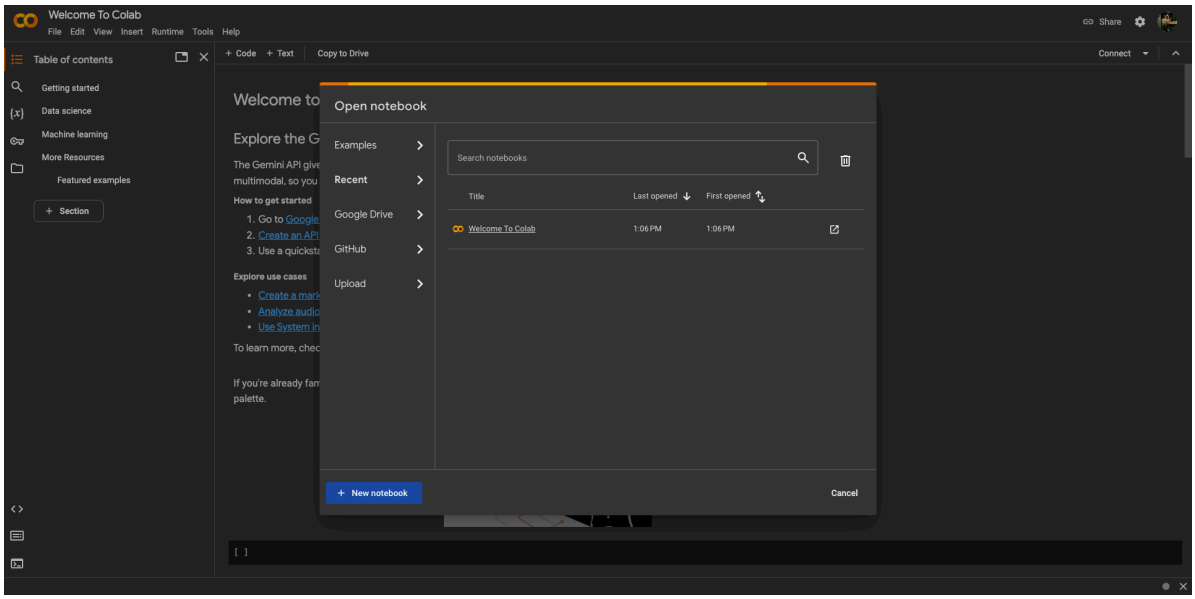


Figure 1.1: Inicio en Google Colab

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.

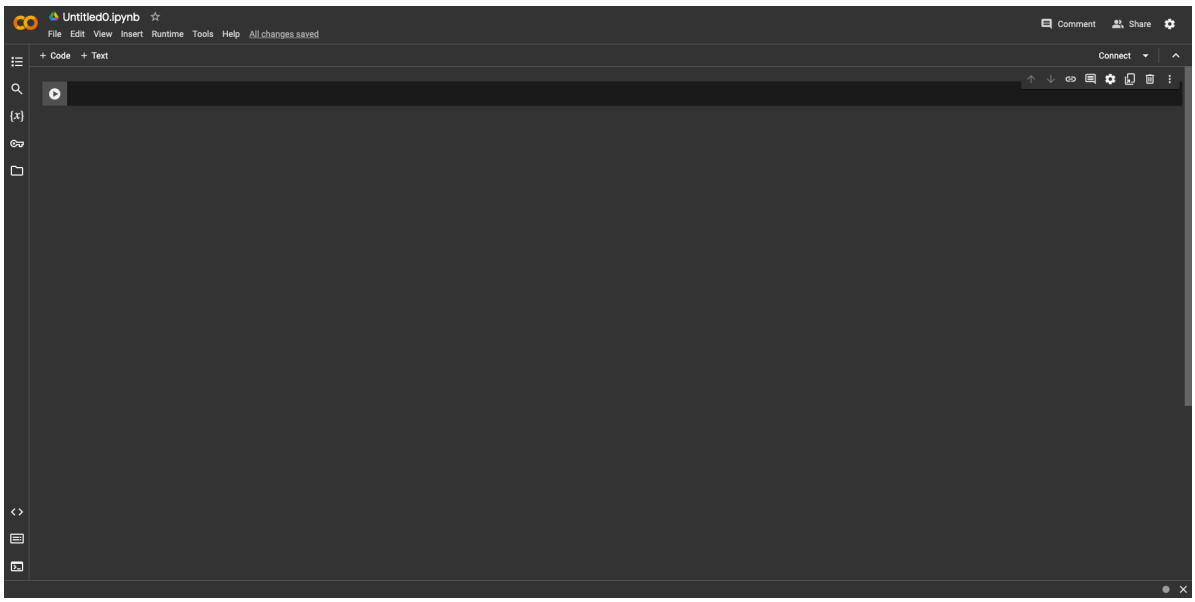


Figure 1.2: Archivo nuevo

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.

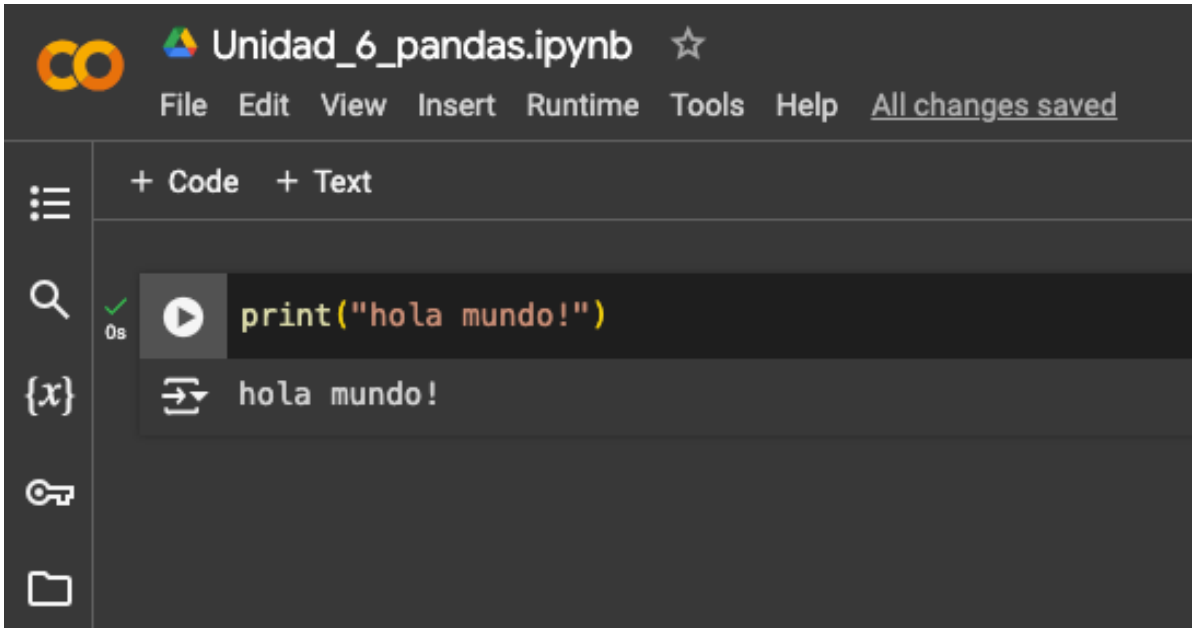


Figure 1.3: Ejecución de una celda de código

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.

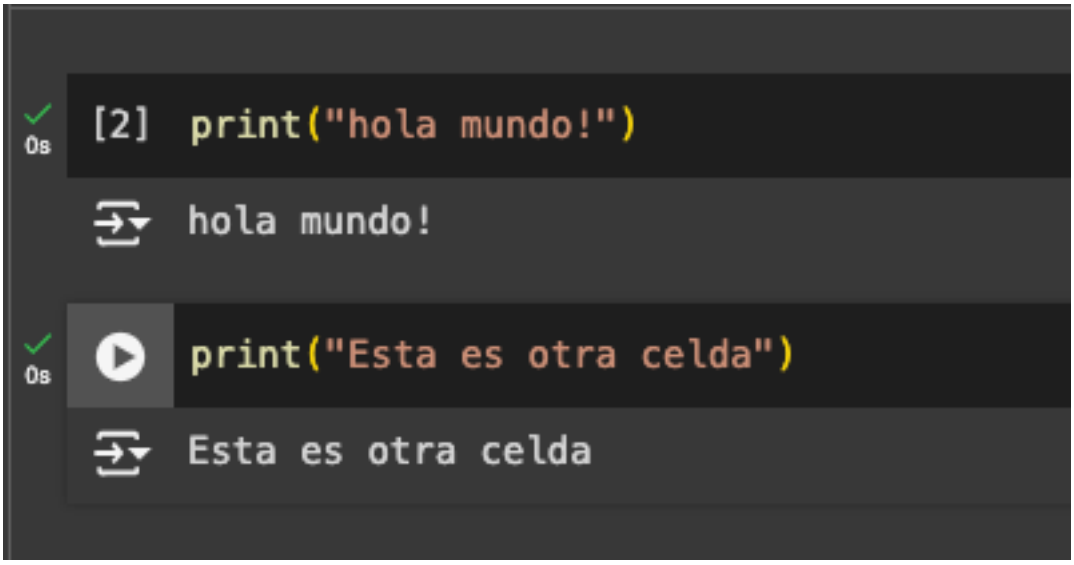
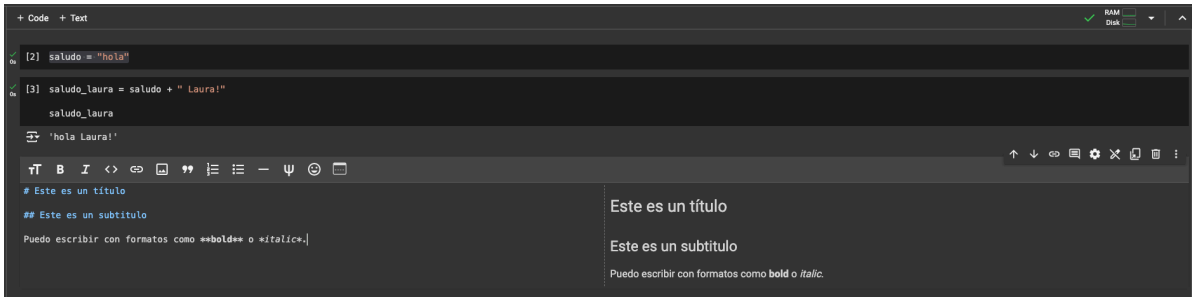


Figure 1.4: Ejecución simultánea de dos celdas de código

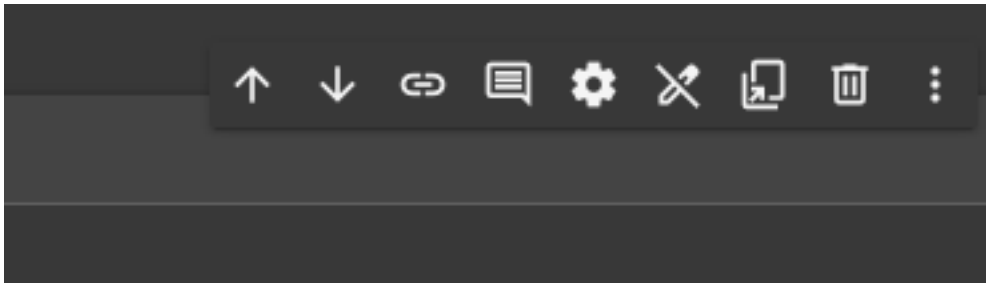
1.3.1.2 Celdas de Texto

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.

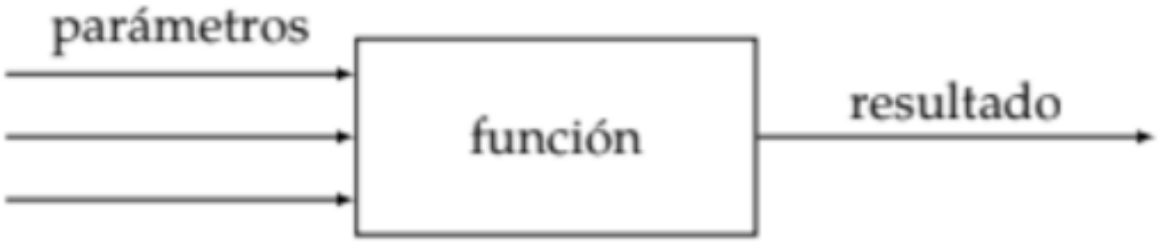


Figure 2.1: Una función recibe parámetros y devuelve un resultado

> **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!".

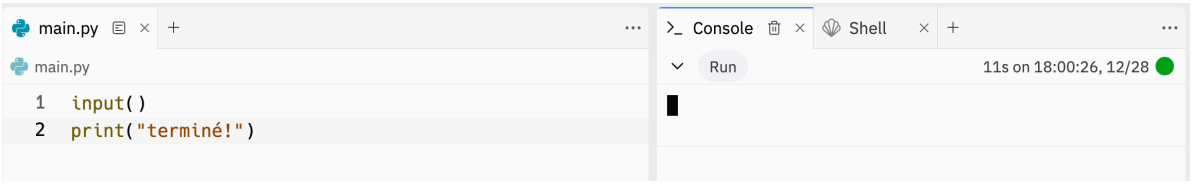


Figure 2.2: Input bloquea el flujo del programa

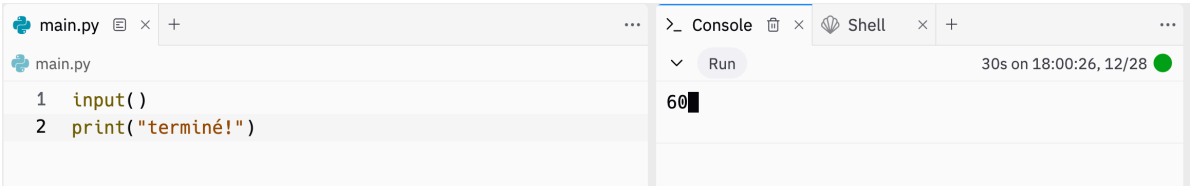


Figure 2.3: Ingresamos un valor (puede ser un número, texto, o ambos)

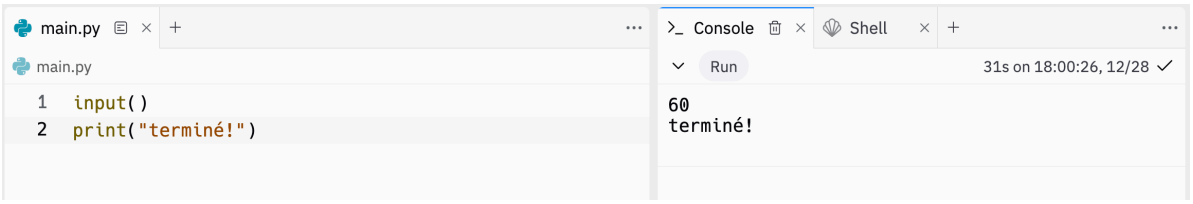


Figure 2.4: Al presionar Enter, la consola muestra el mensaje "terminé!"

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 + "!")
```

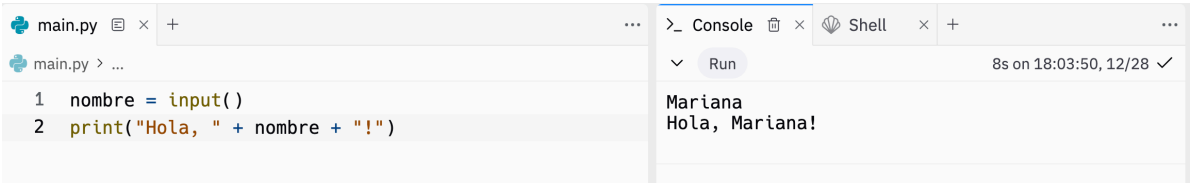


Figure 2.5: Ingresamos “Mariana” y presionamos Enter.

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 + "!")
```

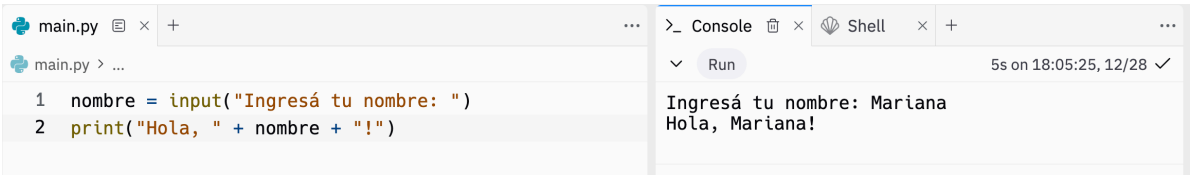


Figure 2.6: Ingresamos “Mariana” y presionamos Enter.

> ⚠ ¡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)
```

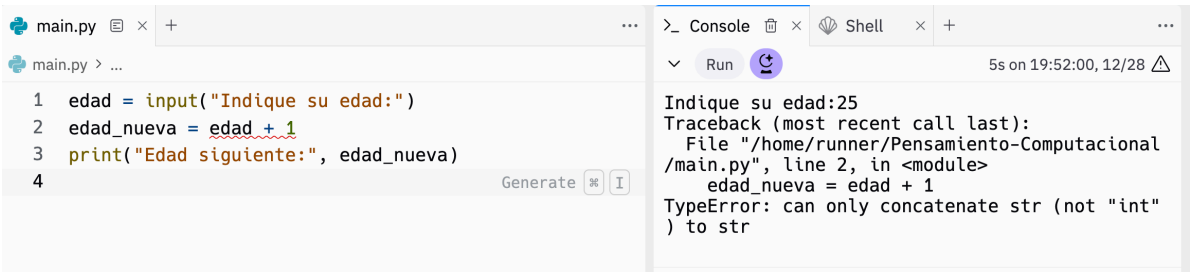


Figure 2.7: Ejecución del bloque de código

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.

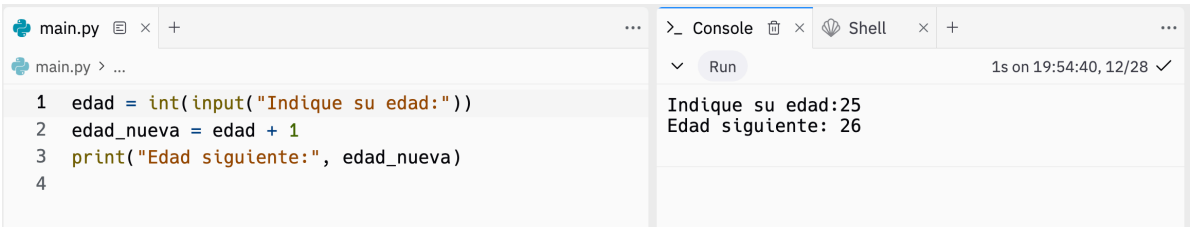


Figure 2.8: Ejecución del bloque de código

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')
```

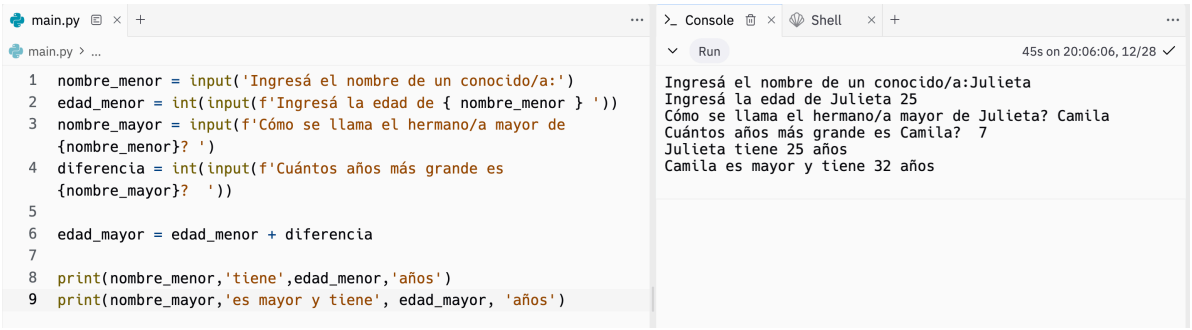


Figure 2.9: Ejecución del bloque de código

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.

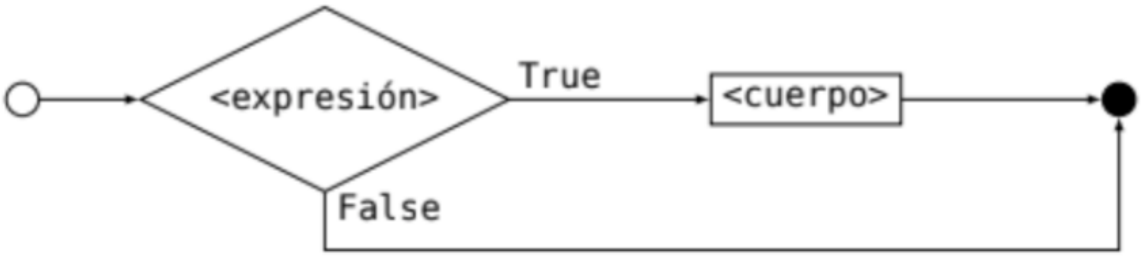


Figure 3.1: Diagrama de Flujo para la instrucción `if`

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`.

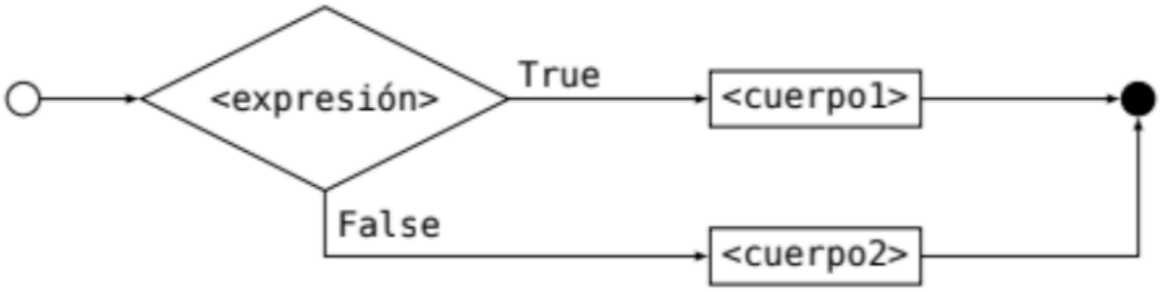


Figure 3.2: Diagrama de Flujo para la instrucción `if-else`

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:

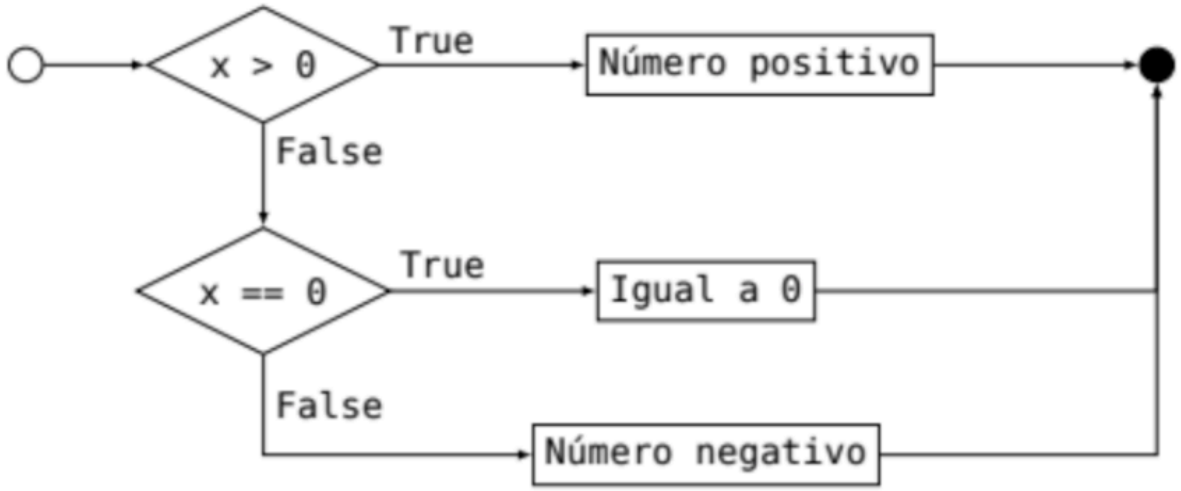


Figure 3.3: Diagrama de Flujo para la instrucción `if-elif-else` del ejemplo

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

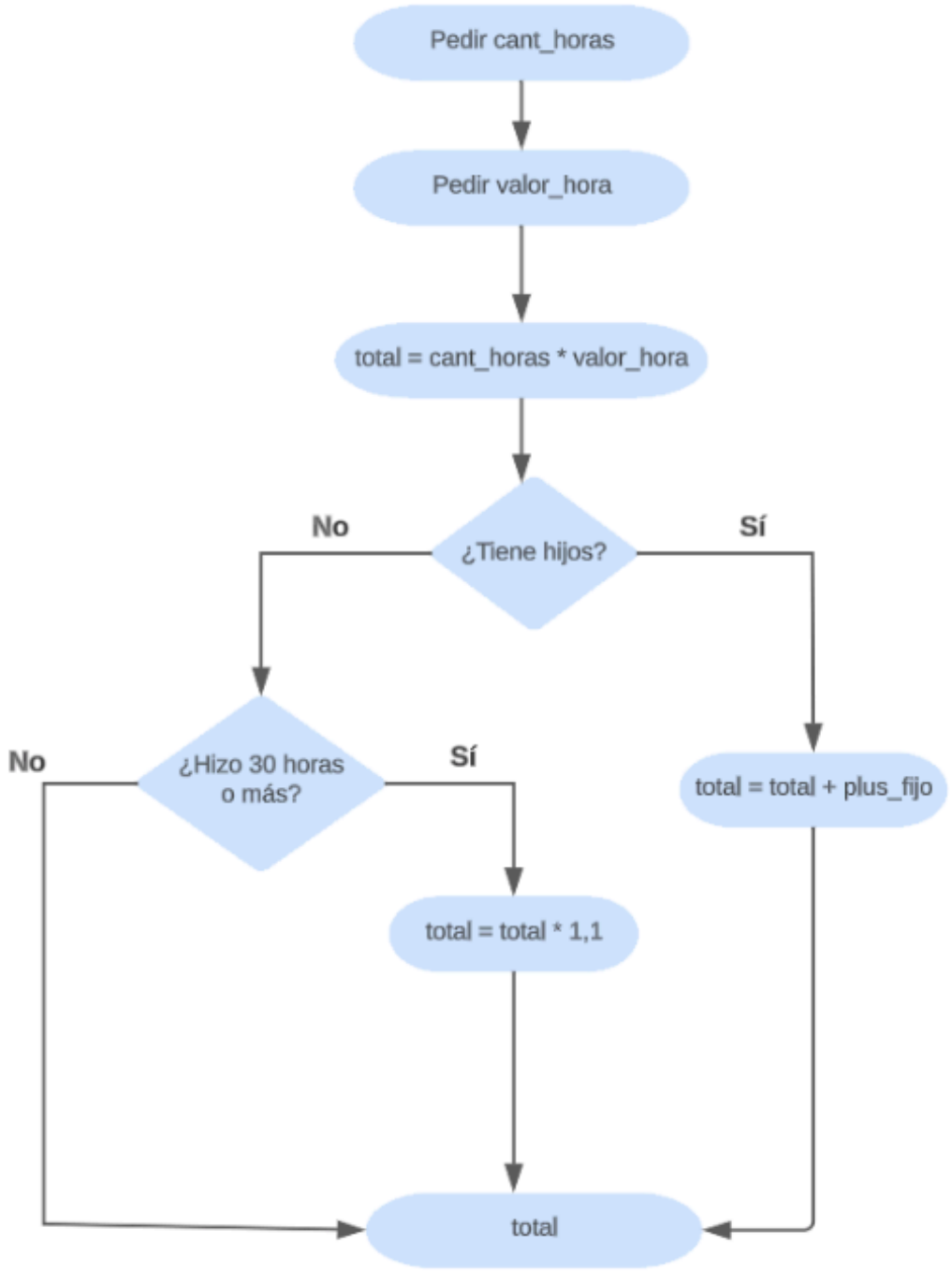


Figure 3.4: Diagrama de Flujo para el desafío

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.

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.

```python
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`:

```python
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.

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.

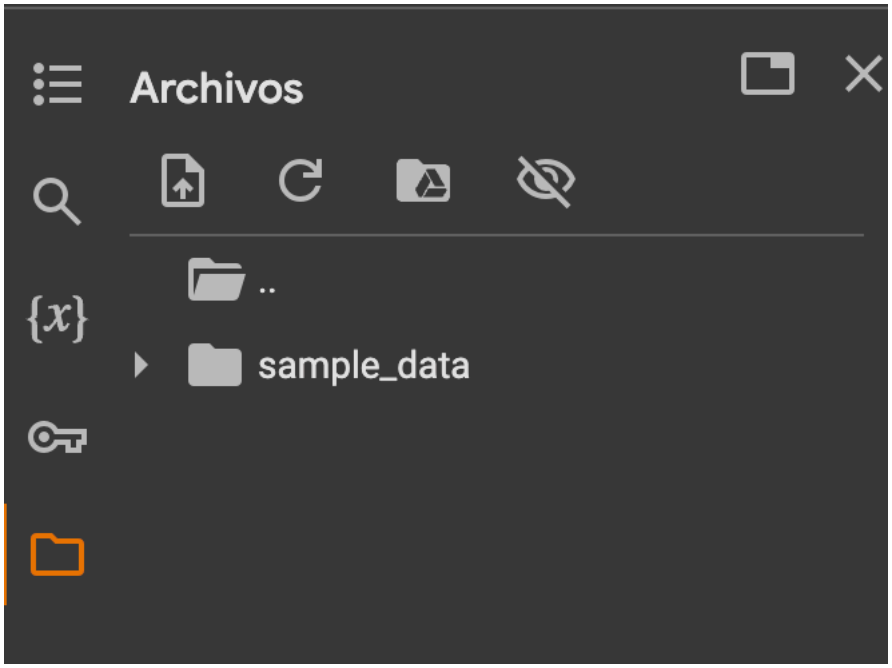


Figure 5.2: Menu Derecho

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)

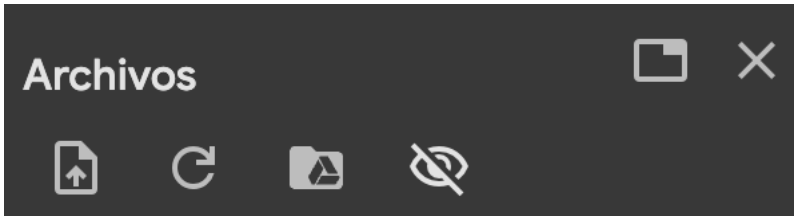


Figure 5.3: Menú Archivos

También podemos crear un nuevo archivo usando el menú de "sample_data" (los tres puntitos) y seleccionando "Nuevo Archivo".

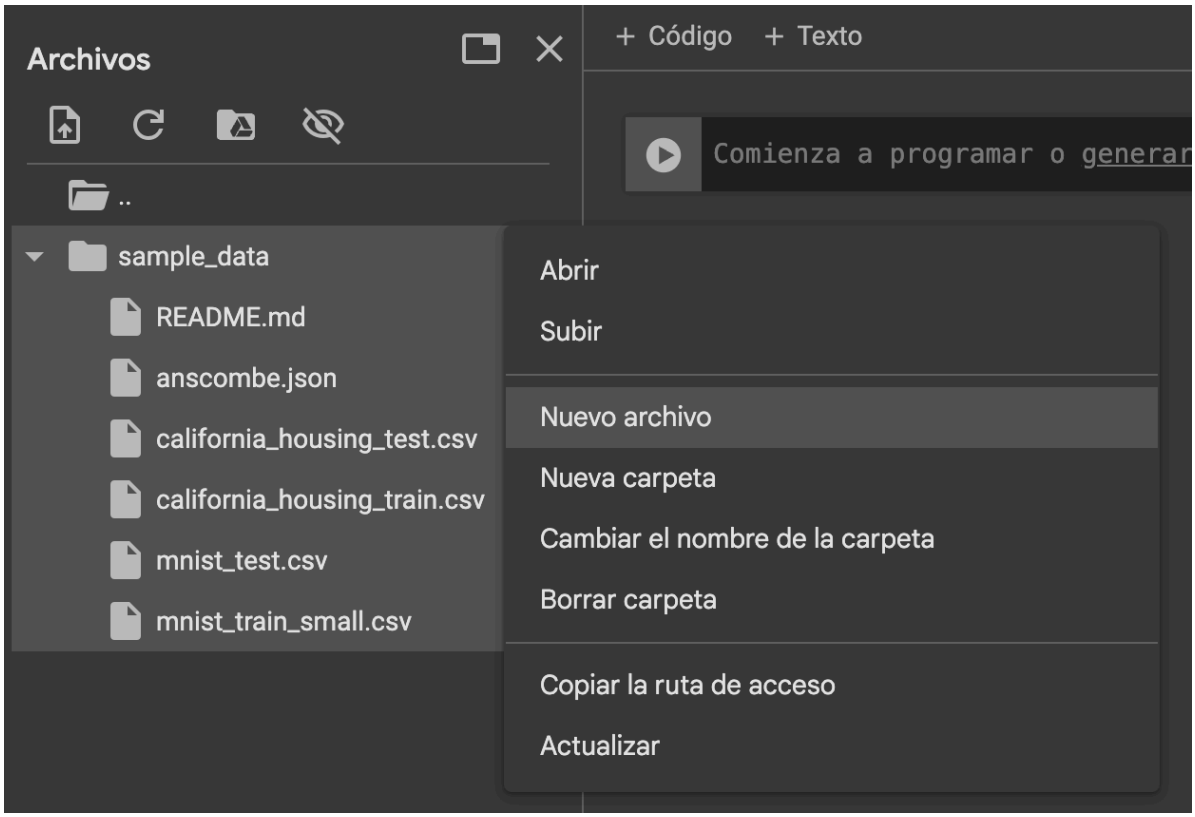


Figure 5.4: Nuevo archivo en sample data

El archivo luego hay que arrastrarlo *fuera* de la carpeta de sample data para poder usar como ruta su nombre directamente.

De esta forma, podemos abrirlo usando como ruta su nombre: `open("archivo_nuevo")`.

Si hacemos doble clic en el archivo, podemos verlo. Si además el archivo es del tipo txt, podemos editarlo. Los archivos CSV no pueden editarse, pero un truco sería cambiarle la extensión a txt, editarlo, y luego volver a cambiarle la extensión a CSV. No es lo ideal, pero nos permite realizar cambios manuales en un archivo para poder trabajar con mayor comodidad.

> **Warning**
>
> ¡Los archivos creados en Colab no se guardan para siempre!
> Te recomendamos que si los vas a necesitar o vas a querer trabajar con ellos en un futuro, los descargues. Lo podés haciendo con clic derecho sobre el archivo, o con el menú del mismo (los tres puntitos).

Con estos tips, ya podrías manejarte correctamente para hacer la guía de archivos en Colab.

5.3 Manejo de errores

Cuando cometemos un error de tipeo o utilizamos mal una sentencia el intérprete nos muestra un error de sintaxis. En la práctica lo vemos como un `SintaxisError`, este tipo de errores se los llama errores sintácticos, la manera de resolverlo es revisar la sintaxis y corregirlo.

Ejemplo: Función mal definida

```
deff incrementar(n):
  return n + 1
```

```
 File ...., line 1
    deff incrementar(n):
         ^^^^^^^^^^^
SyntaxError: invalid syntax
```

Cuando un programa se está ejecutando y ocurre un error se crea una excepción, normalmente el programa detiene su ejecución y se imprime un mensaje. Este tipo de errores se los llama **errores de ejecución**, vamos a ver como manejarlos.

Ejemplo: División por cero

```
dividendo = 10
divisor = 0
resultado = dividendo/divisor
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
File ...
      1 dividendo = 10
      2 divisor = 0
----> 3 resultado = dividendo/divisor
ZeroDivisionError: division by zero
```

Ejemplo: Acceso a un elemento que no existe

```
lista = ["a","b"]
segundo_elemento = lista[2]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
File /...
      1 lista = ["a","b"]
----> 2 segundo_elemento = lista[2]
IndexError: list index out of range
```

Ejemplo: Abrir un archivo que no existe

```
archivo = open("archivo_falso.txt","r")
```

```
FileNotFoundError                         Traceback (most recent call last)
File ...
----> 1 archivo = open("archivo_falso.txt","r")
FileNotFoundError: [Errno 2] No such file or directory: 'archivo_falso.txt'
```

En cada caso el mensaje de error tiene dos partes, la primera indica el tipo de error:

- `ZeroDivisionError`
- `IndexError`
- `FileNotFoundError`

La segunda tiene una descripción:

- `division by zero`
- `list index out of range`
- `No such file or directory`

Además nos da información contextual que puede indicar en la ejecución de qué línea se dio el error:

- línea 3: `----> 3 resultado = dividendo/divisor`.
- línea 2: `----> 2 segundo_elemento = lista[2]`.
- línea 1: `----> 1 archivo = open("archivo_falso.txt","r")`.

En algunas ocasiones es parte del programa manejar operaciones que puedan lanzar este tipo de excepciones sin que el programa detenga su ejecución, para estos casos Python nos provee las sentencias `try`y `except`.

Ejemplo

```
dividendo = 10
divisor = 0
try:
  resultado = dividendo/divisor
except ZeroDivisionError:
  print("No se puede dividir por cero.")
```

```
No se puede dividir por cero.
```

Como se ve en el ejemplo se "envuelve" la operación que puede generar ese tipo de excepción para que lo que resulte de esa operación se pueda controlar. Como vimos más arriba hay distintos tipos de excepciones, la lista completa se puede ver en excepciones.

> Tip
>
> Es **extremadamente importante** que el bloque `try` **sólo tenga dentro** la porción de código que **tiene posibilidad de romper**. No es correcto colocar todo nuestro código dentro del try, porque si tenemos varios posibles puntos de falla, deberían tratarse por separado. Adicionalmente, es una mala práctica tener demasiado código dentro de un bloque try, cuando es innecesario.

5.3.1 Validaciones

Las validaciones son técnicas que permiten asegurar que los valores con los que se vaya a operar estén dentro de determinado conjunto de posibilidades o que tengan ciertas características.

Si bien quien invoca una función debe preocuparse de cumplir con las precondiciones de ésta, si las validaciones están hechas correctamente pueden devolver información valiosa para que el invocante pueda actuar en consecuencia.

También se debe tener en cuenta qué hará nuestro código cuando una validación falle, ya que queremos darle información al invocante que le sirva para procesar el error. El error producido tiene que ser fácilmente reconocible.

Ejemplo:

```
def convertir_a_int(ingreso_usuario):
  try:
    return int(ingreso_usuario)
  except ValueError:
    print("El valor ingresado no es un número")

convertir_a_int("cuarenta")
```

```
El valor ingresado no es un número
```

> **Note**
>
> En Python también tenemos una forma de *arrojar* nosotros un error, es decir, hacer que la función falle y devuelva un mensaje de error. Esto se hace con la sentencia `raise`. No lo vemos en la materia, pero se puede leer más en la documentación de Python.
> Arrojar o levantar excepciones es un tema complejo, porque además de arrojarlas, debemos ser capaces de capturarlas y manejarlas. Es por esto que no se incluye en el temario de la materia.

5.3.2 Varias Excepciones

Una misma línea podría arrojar varios tipos de excepciones distintos. Existe una forma de atajar varios casos de error sobre un mismo try, pero no lo vemos en la materia. Cada par try-except debería hacer referencia a un sólo tipo de error.

5.3.3 Conclusiones

- El manejo de errores es un parte fundamental en el desarrollo de software, tan importante como la funciónalidad que se está programando.
- Los errores que se dan en tiempo de ejecución los podemos "atrapar" con el bloque `try`.
- Hay tipos de excepciones que describen distintos tipos de errores de ejecución.

5.3.4 Bonus Track: Tipos de Errores

A continuación se presenta una tabla con los errores más comunes que se pueden encontrar al programar en Python.

Tipo de error	Descripción	Más Información
SyntaxError	Error de sintaxis	Suele ser un error de tipeo o de uso incorrecto de una sentencia
ZeroDivisionError	División por cero	Se produce cuando se intenta dividir por cero
NameError	Variable no definida	Se produce cuando se intenta utilizar una variable que no fue definida
IndexError	Índice fuera de rango	Se produce cuando se intenta acceder a un elemento de una secuencia, que no existe
FileNotFoundError	Archivo no encontrado	Se produce cuando se intenta abrir un archivo que no existe
TypeError	Tipo de dato incorrecto	Se produce cuando se intenta realizar una operación con un tipo de dato incorrecto
ValueError	Valor incorrecto	Se produce cuando se intenta realizar una operación con un valor incorrecto
KeyError	Clave no encontrada	Se produce cuando se intenta acceder a un elemento de un diccionario que no existe
IOError	Error de entrada/salida	Se produce cuando se intenta realizar una operación de entrada/salida que no se puede realizar (por ejemplo, intentar acceder a un archivo)

6 Bibliotecas de Python

6.1 Introducción

Python es un lenguaje de programación muy popular, poderoso y versátil que cuenta con una amplia gama de bibliotecas que ayudan a que la programación sea más fácil y eficiente. Pero, **¿qué son las bibliotecas?** Las bibliotecas son conjuntos de módulos que contienen funciones, clases y variables relacionadas, que permiten realizar tareas sin tener que escribir el código desde cero y este se puede reutilizar en múltiples programas y proyectos.

Entre las bibliotecas disponibles se encuentran las estándares, que se incluye con cada instalación de Python, y las de código abierto creadas por la gran comunidad de desarrolladores, que constantemente genera nuevas bibliotecas y mejora las existentes. Por ello es aconsejable que, al momento de utilizarlas, se verifique si existe alguna actualización en las guías de usuario.

Asimismo, estas bibliotecas se pueden clasificar según su aplicación y funcionalidad en: procesamiento de datos, visualización, aprendizaje automático, desarrollo web, procesamiento de lenguaje y de imágenes, entre otras. En este capítulo se analizarán tres de las bibliotecas más reconocidas y ampliamente utilizadas de Python: **NumPy** y **Pandas** para procesamiento de datos y **Matplotlib**, para visualización.

6.1.1 ¿Cómo se utilizan las bibliotecas?

Para acceder a una biblioteca y sus funciones, se debe instalar por única vez y luego, importar cada vez que la necesitemos.

En la parte superior de nuestro código debemos correr `import {nombre_de_biblioteca} as {nombre_corto_de_biblioteca}`. El alias o nombre corto de la biblioteca se suele agregar para lograr una mayor legibilidad del código, pero no es mandatorio.

```
import numpy as np
```

> **Note**
>
> En nuestro caso, la instalación no es necesaria, ya que vamos a utilizar Google Colab, pero en caso de usar otro IDE (como por ejemplo, Visual Studio Code), se realiza desde el símbolo del sistema (o en inglés: “Command Prompt”, o terminal o consola), corriendo:

`pip install -nombre_de_biblioteca` o similares.

6.2 Adaptaciones para Colab

A partir de esta guía es probable que debamos reutilizar ciertas cosas o adaptar la forma en la que venimos trabajando. Para esto, vamos a hacer una modificación:

Si tenemos variables o imports definidos en otra celda y no la ejecutamos, al intentar usarla nos va a dar error. Es importante entonces o correr todas las celdas, o que cada celda tenga la información necesaria para poder correrse de forma independiente. Acá vemos cómo, al querer ejecutar sólo la segunda celda, nos da error porque la variable `saludo` está definida en la celda anterior (que no se ejecutó).

6.3 NumPy

NumPy es una biblioteca de código abierto muy utilizada en el campo de la ciencia y la ingeniería. Permite trabajar con datos numéricos, matrices multidimensionales, funciones matemáticas y estadísticas avanzadas.

Como ya se mencionó anteriormente, para utilizarse se debe instalar e importar. Por convención, se suele importar como:

```
import numpy as np
```

NumPy incorpora una estructura de datos propia llamados **arrays** que es similar a la lista de Python, pero puede almacenar y operar con datos de manera mucho más eficiente: **el procesamiento de los arrays es hasta 50 veces más rápido**. Esta diferencia de velocidad se debe, en parte, a que **los arrays contienen datos homogéneos**, a diferencia de las listas que pueden contener distintos tipos de datos dentro.

6.3.1 Arrays

Un **array** es un conjunto de elementos del mismo tipo, donde cada uno de ellos posee una posición y esta es única para cada elemento. Como dijimos arriba, en Python vimos las listas, que es lo más parecido a un Array.

Analicemos el siguiente ejemplo: si pensamos en una matriz, lo primero que nos viene a la mente es una tabla con valores ordenados en filas y columnas, donde una fila es la línea horizontal y una columna es la vertical. Es decir, una matriz es un conjunto de elementos que posee una posición o índice determinado por la fila y la columna, por lo que sería un array.

En este capítulo se trabajará principalmente con vectores y matrices, ya que consideramos que les será útil para aplicar los conocimientos de Numpy en otras materias.

6.3.1.1 Creación de un Array

Un array se crea usando la función `array()` a partir de listas o tuplas. Por ejemplo:

```
a = np.array([1, 2, 3])
print(a)
```

```
[1 2 3]
```

También, se pueden crear arrays particulares, constituidos por ceros con `zeros()` o por unos con `ones()`:

```
# Creo un array de ceros con dos elementos
a_ceros = np.zeros(2)
print(a_ceros)
```

```
[0. 0.]
```

```
# Creo un array de unos con dos elementos
a_unos = np.ones(2)
print(a_unos)
```

```
[1. 1.]
```

Además, se pueden crear arrays con un rango de números, utilizando `arange()` o `linspace()`:

```
# Creo un array con un rango que empieza en 2 hasta 9 y va de 2 en 2.
a_rango = np.arange(2, 9, 2)
print(a_rango)
```

```
[2 4 6 8]
```

```
# Creo un array con un rango formado por 4 números
# que empieza en 2 hasta 10 (incluidos).
a_rango_2 = np.linspace(2, 10, num=4)
print(a_rango_2)
```

```
[ 2.          4.66666667  7.33333333 10.        ]
```

Esto es muy parecido a los rangos que ya vimos en Python, con la sutil diferencia de que el `final` del rango en este caso **sí se incluye**.

Finalmente, para crear arrays de más dimensiones, se utilizan varias listas:

```
matriz = np.array([[1, 2, 3], [4, 5, 6]])

print(matriz)
```

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

6.3.1.2 Atributos de un array

6.3.1.3 Dimensión

Para caracterizar un array es necesario conocer sus dimensiones, utilizando `ndim`. De esta forma, se puede confirmar que el array llamado **matriz**, definido anteriormente, es bidimensional:

```
# Número de ejes o dimensiones de la matriz
matriz.ndim
```

```
2
```

6.3.1.4 Forma

Otra característica de interés es su forma o `shape`: para las matrices bidimensionales, se muestra una tupla (n, m) con el número de filas n y de columnas m:

```
# (n = filas, m = columnas)
matriz.shape
```

```
(2, 3)
```

6.3.1.5 Tamaño

El tamaño de un array es el número total de elementos que contiene, y se obtiene con `size`:

```
# Número total de elementos de la matriz: 2 filas x 3 columnas = 6 elementos
matriz.size
```

```
6
```

6.3.1.6 Posiciones

Al elemento de una matriz A que se encuentra en la **fila i-ésima y la columna j-ésima** se llama a_{ij}. Así, para acceder a un elemento de un array se debe indicar primero la posición de la fila y luego, de la columna:

```
print('Elemento de la primera fila y segunda columna: ', matriz[0, 1])
```

```
Elemento de la primera fila y segunda columna:  2
```

Nótese la diferencia con las matrices (listas de listas) de Python, donde se accedía a un elemento por separado, primero a la fila y luego a la columna: `matriz[0][1]`.

También se puede elegir un rango de elementos en una fila o columna particular:

```
print('Los elementos de la primera fila, columnas 0 y 1: ', matriz[0, 0:2])
```

```
Los elementos de la primera fila, columnas 0 y 1:  [1 2]
```

```
print('Los elementos de la segunda columna, filas 0 y 1: ', matriz[0:2, 1])
```

```
Los elementos de la segunda columna, filas 0 y 1:  [2 5]
```

6.3.1.7 Modificar arrays

De forma similar a lo aprendido con las listas de Python, se pueden modificar los arrays utilizando ciertas funciones. Para entender y aplicar las mismas, definamos un vector llamado a:

```
a = np.array([2, 1, 5, 3, 7, 4, 6, 8])

print(a)
```

```
[2 1 5 3 7 4 6 8]
```

6.3.1.7.1 Reshape

A este vector, se le puede modificar la forma: pasando de ser `(8,)` a `(4,2)`, por dar un ejemplo:

```
a_reshape = a.reshape(2, 4) # 2 filas y 4 columnas

print(a_reshape)
```

```
[[2 1 5 3]
 [7 4 6 8]]
```

6.3.1.7.2 Insert

También, se podría insertar una fila (`axis = 0`) o una columna (`axis = 1`) en una determinada posición. Por ejemplo:

```
# Agregar fila de cincos en posición 1:
print(np.insert(a_reshape, 1, 5, axis=0))
```

```
[[2 1 5 3]
 [5 5 5 5]
 [7 4 6 8]]
```

A la función `insert()`, se le debe indicar:

- El array que se desea modificar
- La posición de la fila o columna que se desea agregar
- Los valores a insertar. **¡Ojo con las dimensiones!** Para el ejemplo anterior, a_reshape tenía 2 filas, por lo que se debe agregar una columna con 2 elementos o una fila con 4.
- El eje que se agrega: una fila (axis = 0) o una columna (axis = 1)

```
# Agregar columna de cincos en posición 1:
print(np.insert(a_reshape, 1, 5, axis=1))
```

```
[[2 5 1 5 3]
 [7 5 4 6 8]]
```

O lo que es equivalente:

```
# Agregar columna de cincos en posición 1:
print(np.insert(a_reshape, 1, [5, 5], axis=1))
```

```
[[2 5 1 5 3]
 [7 5 4 6 8]]
```

6.3.1.7.3 Append y Delete

También podríamos agregar una fila o una columna utilizando `append()` al final, como ocurría con las listas:

```
# Agregar una última fila
a_modificada = np.append(a_reshape, [[1, 2, 3, 4]], axis=0)
print(a_modificada)
```

```
[[2 1 5 3]
 [7 4 6 8]
 [1 2 3 4]]
```

O eliminarlas con `delete()`

```
# Eliminar la fila de la posición 2.
print(np.delete(a_modificada, 2, axis=0))
```

```
[[2 1 5 3]
 [7 4 6 8]]
```

6.3.1.7.4 Concatenate y Sort

Finalmente, podemos concatenar arrays, como los siguientes:

```
a = np.array([2, 1, 5, 3])
b = np.array([7, 4, 6, 8])

# Concatenar a y b:
c = np.concatenate((a, b))
print(c)
```

```
[2 1 5 3 7 4 6 8]
```

Y ordenar los elementos de un array como numérico o alfabético, ascendente o descendente.

```
print(np.sort(c))
```

```
[1 2 3 4 5 6 7 8]
```

6.3.2 Operaciones aritméticas utilizando array

Como se ha mencionado anteriormente, Numpy tiene un gran potencial para realizar operaciones, muy superior al de las listas de Python. Por ejemplo, si quisiéramos sumar dos listas de python necesitaríamos realizar un `for`:

```
# Definir listas
a = [2, 1, 5, 3]
b = [7, 4, 6, 8]
c = []

# Sumar el primer elemento de a con el primero de b, el segundo elemento de a con el segundo
for i in range(len(a)):
  c.append(a[i] + b[i])
print(c)
```

```
[9, 5, 11, 11]
```

Utilizando las funciones de Numpy, esto ya no es más necesario:

```
# add() para sumar elemento a elemento de a y b
c = np.add(a, b)
print(c)
```

```
[ 9  5 11 11]
```

También podemos realizar otras operaciones, como la resta, multiplicación y división. Usar un arreglo dentro de una ecuación nos devuelve otro arreglo, donde cada elemento es el resultado de aplicar la operación a los elementos correspondientes de los arreglos originales.

```
x = np.array([0,  1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = 3 * x + 2
print(y)
```

```
[ 2  5  8 11 14 17 20 23 26 29 32]
```

De esta forma, podemos realizar operaciones aritméticas con arrays de Numpy de forma muy sencilla y rápida. Antes, en Python, para realizar estas operaciones debíamos recurrir a un ciclo `for` o a el uso de `map`. Numpy se ocupa de ahorrarnos el trabajo y calcular, para cada elemento del array, el resultado.

Además, tenemos operaciones básicas que vienen predefinidas por Numpy. Las vamos a ver a continuación.

6.3.2.1 Operaciones básicas:

A continuación se muestra una lista con las operaciones básicas junto con sus operadores asociados, funciones y ejemplos.

Operación	Operador	Función
Suma	`+`	`add()`
Resta	`-`	`subtract()`
Multiplicación	`*`	`multiply()`
División	`/`	`divide()`
Potencia	`**`	`power()`

Definimos los vectores a y b con los que operaremos y veremos ejemplos:

```
a = np.array([1, 3, 5, 7])
b = np.array([1, 1, 2, 2])
```

- Suma:

```
resultado_1 = a + b
print("Suma usando +:", resultado_1)

resultado_2 = np.add(a, b)
print("Suma usando add():", resultado_2)
```

```
Suma usando +: [2 4 7 9]
Suma usando add(): [2 4 7 9]
```

- Resta:

```
resultado_1 = a - b
print("Resta usando -:", resultado_1)

resultado_2 = np.subtract(a, b)
print("Resta usando subtract():", resultado_2)
```

```
Resta usando -: [0 2 3 5]
Resta usando subtract(): [0 2 3 5]
```

- Multiplicación:

```
resultado_1 = a * b
print("Multiplicación usando *:", resultado_1)

resultado_2 = np.multiply(a, b)
print("Multiplicación usando multiply():", resultado_2)
```

```
Multiplicación usando *: [ 1  3 10 14]
Multiplicación usando multiply(): [ 1  3 10 14]
```

- División:

```
resultado_1 = a / b
print("División usando /:", resultado_1)

resultado_2 = np.divide(a, b)
print("División usando divide():", resultado_2)
```

```
División usando /: [1.  3.  2.5 3.5]
División usando divide(): [1.  3.  2.5 3.5]
```

- Potencia:

```
resultado_1 = a ** b
print("Potencia usando **:", resultado_1)

resultado_2 = np.power(a, b)
print("Potencia usando power():", resultado_2)
```

```
Potencia usando **: [ 1  3 25 49]
Potencia usando power(): [ 1  3 25 49]
```

> **Note**
>
> Note que si quisiéramos operar con un vector `b` de elementos iguales, podríamos utilizar un escalar.

```
b = np.array([2, 2, 2, 2])

resultado_1 = a * b
print("Usando un vector b = [2, 2, 2, 2]:", resultado_1)

resultado_2 = a * 2
print("Usando un escalar b = 2:", resultado_2)
```

```
Usando un vector b = [2, 2, 2, 2]: [ 2  6 10 14]
Usando un escalar b = 2: [ 2  6 10 14]
```

6.3.2.2 Logaritmo:

NumPy provee funciones para los logaritmos de base 2, 10 y e:

Base	Función
2	`log2()`
10	`log10()`
e	`log()`

Por ejemplo:

```
# Ejemplo log2()
print("Logaritmo base 2:", np.log2([2, 4, 8, 16]))
# Ejemplo log10()
print("Logaritmo base 10:", np.log10([10, 100, 1000, 10000]))
# Ejemplo log()
print("Logaritmo base e:", np.log([1, np.e, np.e**2]))
```

```
Logaritmo base 2: [1. 2. 3. 4.]
Logaritmo base 10: [1. 2. 3. 4.]
Logaritmo base e: [0. 1. 2.]
```

> **Note**
>
> Note que el número de Euler o número e es una constante incluida en NumPy como: `np.e`

```
np.e
```

```
2.718281828459045
```

> **Funciones trigonométricas (opcional)**
>
> Esta parte del apunte es opcional, es decir, no se evalúa en los exámenes. A continuación, una lista con las funciones trigonométricas más utilizadas, que toman los valores en radianes:

Función trigonométrica	Función
seno	`sin()`
coseno	`cos()`
tangente	`tan()`
arcoseno	`arcsin()`
arcocoseno	`arccos()`
arcotangente	`arctan()`

Por ejemplo:

```
# Ejemplo de seno
print("Seno de   / 2:", np.sin(np.pi / 2))

# Ejemplo de arcoseno
print(np.arcsin(1))
```

```
Seno de   / 2: 1.0
1.5707963267948966
```

```
# Ejemplo de coseno
print("Coseno de  :", np.cos(np.pi))

# Ejemplo de arcococoseno
print("Arcoseno de -1:", np.arccos(-1))
```

```
Coseno de  : -1.0
Arcoseno de -1: 3.141592653589793
```

```
# Ejemplo de tangente:
print("Tangente de 0:", np.tan(0))

# Ejemplo de arcotangente:
print("Arcotangente de 0:", np.arctan(0))
```

```
Tangente de 0: 0.0
Arcotangente de 0: 0.0
```

> **Note**
>
> Note que el número es una constante incluida en NumPy como: `np.pi`

```
np.pi
```

```
3.141592653589793
```

Para convertir los radianes a grados y viceversa, se utiliza `deg2rad()` y `rad2deg()` respectivamente:

```
print("De grados [90, 180, 270, 360] a radianes:",
      np.deg2rad([90, 180, 270, 360]))

print("De radianes [ /2,  , 1.5* , 2* ] a grados:",
      np.rad2deg([np.pi/2, np.pi, 1.5*np.pi, 2*np.pi]))
```

```
De grados [90, 180, 270, 360] a radianes: [1.57079633 3.14159265 4.71238898 6.28318531]
De radianes [ /2,  , 1.5* , 2* ] a grados: [ 90. 180. 270. 360.]
```

6.3.2.3 Operaciones con matrices:

A continuación, una lista con las operaciones que les pueden ser de interés mientras estudian álgebra matricial:

Función	Descripción	Comentario
`dot()`	Producto escalar	Se utiliza para obtener el producto escalar entre dos vectores. El resultado es un número.
`dot()`	Producto vectorial	También se utiliza para multiplicar matrices. El resultado es una matriz
`transpose()`	Traspuesta	Cambia las filas por las columnas y viceversa
`linalg.inv()`	Inversa	Inversa de una matriz
`linalg.det()`	Determinante	Determinante de una matriz
`eye()`	Matriz identidad	Matriz cuadrada con unos en la diagonal principal y ceros en el resto

Definimos los arreglos 1 y 2, y matrices 1 y 2 con los que operaremos y veremos ejemplos:

```
# Crear arreglos
arreglo_1 = np.array([1, 2])
arreglo_2 = np.array([3, 4])

# Crear matrices
matriz_1 = np.array([[1, 3], [5, 7]])
matriz_2 = np.array([[2, 6], [4, 8]])
```

```
print("Producto escalar entre el array 1 y 2: \n", np.dot(arreglo_1, arreglo_2))
```

```
Producto escalar entre el array 1 y 2:
 11
```

```
print("Producto vectorial entre la matriz 1 y 2: \n", np.dot(matriz_1, matriz_2))
```

```
Producto vectorial entre la matriz 1 y 2:
 [[14 30]
 [38 86]]
```

```
print("Traspuesta de la matriz 1: \n", np.transpose(matriz_1))
```

```
Traspuesta de la matriz 1:
 [[1 5]
 [3 7]]
```

```
print("Inversa de la matriz 1: \n", np.linalg.inv(matriz_1))
```

```
Inversa de la matriz 1:
 [[-0.875  0.375]
 [ 0.625 -0.125]]
```

```
print("Determinante de la matriz 1: \n", np.linalg.det(matriz_1))
```

```
Determinante de la matriz 1:
 -7.999999999999998
```

> **Note**
>
> Note que así como existen constantes numéricas, existen las matrices particulares como las compuestas por ceros `np.zeros()`, por unos `np.ones()` y la matriz identidad `np.eyes`.

```
print("Matriz de identidad de 3x3: \n", np.eye(3))
```

```
Matriz de identidad de 3x3:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

6.3.2.4 Más operaciones útiles:

Operaciones	Función	Descripción
Máximo	`max()`	Valor máximo del array o del eje indicado
Mínimo	`min()`	Valor mínimo del array o del eje indicado
Suma	`sum()`	Suma de todos los elementos o del eje indicado
Promedio	`mean()`	Promedio de todos los elementos o del eje indicado

Utilizando la matriz **data** como ejemplo:

```
data = np.array([[1, 2], [5, 3], [4, 6]])
```

- Valor máximo

```
print("Valor máximo de todo el array: ", data.max())
print("Valores máximos de cada columna: ", data.max(axis=0))
```

```
Valor máximo de todo el array:  6
Valores máximos de cada columna:  [5 6]
```

- Valor mínimo

```
print("Valor mínimo de todo el array: ", data.min())
print("Valores mínimos de cada fila: ", data.min(axis=1))
```

```
Valor mínimo de todo el array:  1
Valores mínimos de cada fila:  [1 3 4]
```

- Suma de elementos:

```
print("Suma de todos los elementos del array: ", data.sum())
print("Suma de los elementos de cada fila: ", data.sum(axis=1))
```

```
Suma de todos los elementos del array:  21
Suma de los elementos de cada fila:  [ 3  8 10]
```

- Promedio:

```
print("Promedio de todos los elementos del array: ", data.mean())
print("Promedio de los elementos de cada columna: ", data.mean(axis=0))
```

```
Promedio de todos los elementos del array:  3.5
Promedio de los elementos de cada columna:  [3.33333333 3.66666667]
```

> **Note**
>
> Numpy te va a ser muy útil cuando curses materias como Análisis Matemático, Álgebra, Física, Estadística, entre otras. Te va a permitir realizar operaciones de manera rápida y eficiente, y te va a ayudar a entender mejor los conceptos.

6.4 Pandas

Pandas es una biblioteca de código abierto diseñada específicamente para la manipulación y el análisis de datos en Python. Es una herramienta poderosa que puede ayudar a los usuarios a limpiar, transformar y analizar datos de una manera rápida y eficiente.

Al igual que Numpy, se debe importar. Por convención:

```
import pandas as pd
```

Los datos a trabajar en Pandas van a tener, en general, una forma muy similar a las tablas:

	Jurisdicción	Capital	Población (hab)	Superficie (km2)	PBI
0	Ciudad Autónoma de Buenos Aires		3075646	205.9	1.548638e+08
1	Buenos Aires	La Plata	17541141	305907.4	2.926899e+08
2	Catamarca	San Fernando del Valle de Catamarca	415438	101486.1	6.150949e+06
3	Chaco	Resistencia	1204541	99763.3	9.832643e+06
4	Chubut	Rawson	618994	224302.3	1.774785e+07

Figure 6.1: Esquema de figuras y axes

Pandas es una biblioteca un poco más pesada, por lo que vamos a poder aprovechar mucho más Google Colab, que maneja grandes datos de forma rápida y eficiente.

6.4.1 Serie

Pandas incorpora dos estructuras de datos nuevas llamadas: **Series** y **DataFrames**.

Una **serie** es un vector capaz de contener cualquier tipo de dato, como por ejemplo, números enteros o decimales, strings, objetos de Python, etc.

6.4.1.1 Creación de una Serie

Para crearlas, se puede partir de un escalar, una lista, un diccionario, etc., utilizando `pd.Serie()`:

```
# Crear serie partiendo de una lista:
lista = [1, "a", 3.5]

pd.Series(lista)
```

```
0      1
1      a
2    3.5
dtype: object
```

Esto es análogo a lo que se hacía con los arrays de Numpy, pero con la diferencia de que las series de Pandas pueden contener datos de distinto tipo.

Notemos que se ven dos líneas verticales de datos en el output. A la derecha se observa una columna con los elementos de la lista antes creada, y a la izquierda se encuentra el **índice**, formado por valores desde 0 a n-1, siendo n la cantidad de elementos. Este índice numérico es el predefinido, que nos muestra la posición de los elementos dentro de la Serie.
También podríamos cambiarlo si quisiéramos. En ese caso, se puede establecer utilizando `index`.

El índice es de vital importancia ya que permite acceder a los elementos de la serie (muy parecido a cómo las claves nos permiten acceden a los valores de un diccionario). Al elegir índices personalizados, tenemos que tener en cuenta que su longitud debe ser acorde al número de elementos de la Serie. De lo contrario, se mostrará un ValueError.

```
# Crear serie partiendo de una lista, indicando el índice
pd.Series(lista, index = ["x", "y", "z"])
```

```
x      1
y      a
z    3.5
dtype: object
```

También podemos crear Series utilizando diccionarios, y en ese caso, sus claves (o keys) pasan a formar el índice.

```
# Crear serie partiendo de un diccionario:
diccionario = {"x": 1, "y": "a", "z": 3.5}

a = pd.Series(diccionario)
a
```

```
x      1
y      a
z    3.5
dtype: object
```

6.4.1.2 Accediendo a un elemento

Como ya se debe estar imaginando, para acceder a un elemento de la serie, se debe indicar el valor del índice.

```
# Acceder al elemento de índice y:
a["y"]
```

```
'a'
```

Si los índices son numéricos, simplemente se le pasa entre corchetes un número.

6.4.1.3 Operaciones con Series

Así como pasaba con los arrays de NumPy, las Series no requieren recorrer valor por valor en un ciclo for si queremos realizar operaraciones. Podemos usar directamente operadores con Series.
Por ejemplo:

```
a + a
```

```
x      2
y     aa
z    7.0
dtype: object
```

```
a * 3
```

```
x       3
y     aaa
z    10.5
dtype: object
```

6.4.2 DataFrame

Un **DataFrame** es una estructura de datos tabular **(bidimensional, en forma de tabla)**, compuesta por filas y columnas, que se parece mucho a una hoja de cálculo de Excel.

6.4.2.1 Creando DataFrames

Para crearlos, se utiliza `DataFrame()` y se ingresan diferentes estructuras como arrays, diccionarios, listas, series u otros dataframes.

En el siguiente ejemplo, se crea un Dataframe partiendo de un diccionario llamado `data` para las columnas y de una lista `label` para el índice:

```
data = {'columna_1': ['a', 'b', 'c', 'd', 'e', 'f'],
        'columna_2': [2.5, 3, 0.5, None, 5, None],
        'columna_3': [1, 3, 2, 3, 2, 3]}

labels = ['a1', 'a2', 'a3', 'a4', 'a5', 'a6']

df = pd.DataFrame(data, index=labels)
df
```

	columna_1	columna_2	columna_3
a1	a	2.5	1
a2	b	3.0	3

	columna_1	columna_2	columna_3
a3	c	0.5	2
a4	d	NaN	3
a5	e	5.0	2
a6	f	NaN	3

Lo que vemos acá es el output de ejecutar el código de arriba en una celda: una tabla donde tenemos 3 columnas, y el índice de cada fila son los elementos de la lista `labels`.

En vez de tener un valor `None`, Pandas lo recibe y lo transforma en `NaN`, que significa "Not a Number". Esto es muy útil para trabajar con datos faltantes, ya que Pandas nos permite realizar operaciones con ellos sin que nos de error (a diferencia de Python).

6.4.2.2 Atributos y descripción de un Dataframe

A continuación, se observa una tabla con métodos que nos permiten conocer las características de un determinado DataFrame.

Método o atributo	Descripción
`.info()`	Resume la información del DataFrame
`.shape`	Devuelve una tupla con el número de filas y columnas
`.size`	Número de elementos, aunque también puede usarse `len`
`.columns`	Lista con los nombres de las columnas
`.index`	Lista con los nombres de las filas
`.head()`	Muestra las primeras filas
`.tail()`	Muestra las últimas filas
`.describe()`	Brinda métricas de las columnas numéricas

Para ejemplificar los métodos y las funciones de Pandas, usaremos el **DataFrame df** definido en el siguiente bloque de código.

```
data = {'nombre': ['José Martínez', 'Rosa Díaz', 'Javier García', 'Carmen López', 'Marisa C
                   'Pilar González', 'Pedro Tenorio', 'Santiago Manzano', 'Macarena Álvarez'
        'edad': [18, 32, 24, 35, 46, 68, 51, 22, 35, 46, 53, 58, 27, 20],
        'genero': ['H', 'M', 'H', 'M', 'X', 'H', 'H', 'M', 'H', 'X', 'M', 'H', 'H', 'M'],
        'peso': [85.0, 65.0, None, 65.0, 51.0, 66.0, 62.0, 60.0, 90.0, 75.0, 55.0, 78.0, 109
        'altura': [1.79, 1.73, 1.81, 1.7, 1.58, 1.74, 1.72, 1.66, 1.94, 1.85, 1.62, 1.87, 1.9
        'colesterol': [182.0, 232.0, 191.0, 200.0, 148.0, 249.0, 276.0, None, 241.0, 280.0, 2

df = pd.DataFrame(data)
df
```

	nombre	edad	genero	peso	altura	colesterol
0	José Martínez	18	H	85.0	1.79	182.0
1	Rosa Díaz	32	M	65.0	1.73	232.0
2	Javier Garcíaz	24	H	NaN	1.81	191.0
3	Carmen López	35	M	65.0	1.70	200.0
4	Marisa Collado	46	X	51.0	1.58	148.0
5	Antonio Ruiz	68	H	66.0	1.74	249.0
6	Antonio Fernández	51	H	62.0	1.72	276.0
7	Pilar González	22	M	60.0	1.66	NaN
8	Pedro Tenorio	35	H	90.0	1.94	241.0
9	Santiago Manzano	46	X	75.0	1.85	280.0
10	Macarena Álvarez	53	M	55.0	1.62	262.0
11	José Sanz	58	H	78.0	1.87	198.0
12	Miguel Gutiérrez	27	H	109.0	1.98	210.0
13	Carolina Moreno	20	M	61.0	1.77	194.0

> **Note**
>
> Muchas veces no vamos a querer modificar el dataframe original, pero sin manipularlo. En ese caso, podemos hacer una copia del dataframe original, de esta forma: `df_copy = df.copy()`

6.4.2.2.1 Info, dtypes, columns e index

Con `info()` se puede ver:

- el índice en la primera línea, que es un rango de 0 a 13
- el número total de columnas en la segunda línea
- el uso de la memoria en la última
- una tabla con los nombres de las columnas en **Column**, la cantidad de valores no nulos en **Non-Null Count** y el tipo de dato en **Dtype** para cada una de ellas.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14 entries, 0 to 13
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
```

```
---  ------      --------------  -----
 0   nombre      14 non-null     object
 1   edad        14 non-null     int64
 2   genero      14 non-null     object
 3   peso        13 non-null     float64
 4   altura      14 non-null     float64
 5   colesterol  13 non-null     float64
dtypes: float64(3), int64(1), object(2)
memory usage: 800.0+ bytes
```

Note que utilizando `columns` e `index` se obtiene parte de esta información:

```
# Nombre de cada columna
df.columns
```

```
Index(['nombre', 'edad', 'genero', 'peso', 'altura', 'colesterol'], dtype='object')
```

```
# índice
df.index
```

```
RangeIndex(start=0, stop=14, step=1)
```

6.4.2.2.2 Shape y Size

La forma del DataFrame es de 14 filas y 6 columnas, por lo que contiene 84 elementos.

```
# Forma del DataFrame (filas, columnas)
df.shape
```

```
(14, 6)
```

```
# Número de elementos del DataFrame
df.size
```

```
84
```

6.4.2.2.3 Head y Tail

Asimismo, cuando no conocemos un DataFrame, puede ser importante ver las primeras 5 filas con `head()` o las últimas con `tail()`. Si se quisiera observar un número determinado, sólo hay que especificarlo, por ejemplo:

```
# Mostrar las primeras 3 filas.
df.head(3)
```

	nombre	edad	genero	peso	altura	colesterol
0	José Martínez	18	H	85.0	1.79	182.0
1	Rosa Díaz	32	M	65.0	1.73	232.0
2	Javier Garcíaz	24	H	NaN	1.81	191.0

```
# Mostrar las últimas 5 filas.
df.tail()
```

	nombre	edad	genero	peso	altura	colesterol
9	Santiago Manzano	46	X	75.0	1.85	280.0
10	Macarena Álvarez	53	M	55.0	1.62	262.0
11	José Sanz	58	H	78.0	1.87	198.0
12	Miguel Gutiérrez	27	H	109.0	1.98	210.0
13	Carolina Moreno	20	M	61.0	1.77	194.0

6.4.2.2.4 Describe

Por otro lado, `describe()` devuelve un resumen descriptivo de las columnas de valores numéricos, como “edad”, “peso”, “altura” y “colesterol”.

```
df.describe()
```

	edad	peso	altura	colesterol
count	14.000000	13.000000	14.000000	13.000000
mean	38.214286	70.923077	1.768571	220.230769
std	15.621379	16.126901	0.115016	39.847948

	edad	peso	altura	colesterol
min	18.000000	51.000000	1.580000	148.000000
25%	24.750000	61.000000	1.705000	194.000000
50%	35.000000	65.000000	1.755000	210.000000
75%	49.750000	78.000000	1.840000	249.000000
max	68.000000	109.000000	1.980000	280.000000

Estas métricas podrían obtenerse de forma puntual (tanto para todo el DataFrame como para sólo una columna) utilizando funciones determinadas, como:

- `count()`: contabiliza los valores no nulos
- `mean()`: promedio
- `min()`: valor mínimo
- `max()`: valor máximo

Por ejemplo:

```
df.count()
```

```
nombre        14
edad          14
genero        14
peso          13
altura        14
colesterol    13
dtype: int64
```

```
df.min()
```

```
nombre        Antonio Fernández
edad                         18
genero                        H
peso                       51.0
altura                     1.58
colesterol                148.0
dtype: object
```

```
df.max()
```

```
nombre        Santiago Manzano
edad                        68
genero                       X
peso                     109.0
altura                    1.98
colesterol               280.0
dtype: object
```

6.4.2.3 Accediendo a filas de un DataFrame

Finalmente, como ocurre con las series, para acceder a los elementos de un DataFrame se puede indicar la posición o el nombre de la fila o columna.

Para acceder a una fila en particular, utilizamos `iloc[]`. Podemos pasarle a `iloc` un entero, una lista de enteros, un rango de números (que indican las posiciones) o directamente el valor del índice.

```
# Mostrar la fila de posición 0, usando doble corchete [[]]
# Recibe una lista de elementos a mostrar (que contiene sólo al 0)
df.iloc[[0]]
```

	nombre	edad	genero	peso	altura	colesterol
0	José Martínez	18	H	85.0	1.79	182.0

Si queremos mostrar un rango, lo hacemos así:

```
# Mostrar las filas de posición 0 y 3, usando doble corchete [[]]
# Recibe una lista de elementos a mostrar
df.iloc[[0, 3]]
```

	nombre	edad	genero	peso	altura	colesterol
0	José Martínez	18	H	85.0	1.79	182.0
3	Carmen López	35	M	65.0	1.70	200.0

```
# Mostrar las filas de posiciones entre 0 hasta 3 (exclusive)
# Usa slices
df.iloc[:3]
```

	nombre	edad	genero	peso	altura	colesterol
0	José Martínez	18	H	85.0	1.79	182.0
1	Rosa Díaz	32	M	65.0	1.73	232.0
2	Javier Garcíaz	24	H	NaN	1.81	191.0

```
# El equivalente a df.iloc[:3] es el uso de head(3)
df.head(3)
```

	nombre	edad	genero	peso	altura	colesterol
0	José Martínez	18	H	85.0	1.79	182.0
1	Rosa Díaz	32	M	65.0	1.73	232.0
2	Javier Garcíaz	24	H	NaN	1.81	191.0

6.4.2.4 Accediendo a columnas de un DataFrame

A veces no queremos sólo acceder a filas, sino a columnas. Por ejemplo, para calcular sumas, promedios, máximos o mínimos, etc.

Para acceder a una **columna** se pueden utilizar una lista con los nombres de las columnas que se quieren mostrar: `DataFrame[[columna1, columna2]]`.

```
# Mostrar la columna "nombre"
df[['nombre']]
```

	nombre
0	José Martínez
1	Rosa Díaz
2	Javier Garcíaz
3	Carmen López
4	Marisa Collado
5	Antonio Ruiz
6	Antonio Fernández
7	Pilar González
8	Pedro Tenorio
9	Santiago Manzano
10	Macarena Álvarez
11	José Sanz
12	Miguel Gutiérrez

	nombre
13	Carolina Moreno

```
# Mostrar más de una columna: "nombre" y "edad":
df[['nombre', 'edad']]
```

	nombre	edad
0	José Martínez	18
1	Rosa Díaz	32
2	Javier Garcíaz	24
3	Carmen López	35
4	Marisa Collado	46
5	Antonio Ruiz	68
6	Antonio Fernández	51
7	Pilar González	22
8	Pedro Tenorio	35
9	Santiago Manzano	46
10	Macarena Álvarez	53
11	José Sanz	58
12	Miguel Gutiérrez	27
13	Carolina Moreno	20

De esta forma, podríamos hacer algo así:

```
# calculamos el promedio para los valores de la columna 'edad'
df[['edad']].mean()
```

```
edad    38.214286
dtype: float64
```

6.4.2.5 Modificar un Dataframe

A la hora de modificar un DataFrame, tenemos distintas posibilidades:

- Cambiar la estructura del mismo, como los nombres de las columnas y de los índices,
- Agregar una nueva filas o columna
- Reemplazar un dato en una determinada posición.

A continuación, se enumeran distintos métodos para llevar a cabo estos cambios.

Método	Descripción
`rename()`	Renombra las columnas
`insert()`	Agrega columnas
`drop()`	Elimina columnas y filas
`loc[fila]`	Agrega una fila en un índice dado
`loc[fila, columna]`	Modifica un valor particular dado un índice y una columna
`map()`	Busca un valor dado en una columna y lo reemplaza
`replace()`	Reemplaza un valor dado en una columna

6.4.2.5.1 Rename, insert y drop

Para renombrar una columna, se utiliza un diccionario: `rename(columns={"nombre_columna": "nuevo_nombre_columna"})`

```
# Reemplazo "nombre" por "nombre y apellido"
df = df.rename(columns={"nombre": "nombre y apellido"})
df.head() # para que veamos el cambio de nombre en la columna
```

	nombre y apellido	edad	genero	peso	altura	colesterol
0	José Martínez	18	H	85.0	1.79	182.0
1	Rosa Díaz	32	M	65.0	1.73	232.0
2	Javier Garcíaz	24	H	NaN	1.81	191.0
3	Carmen López	35	M	65.0	1.70	200.0
4	Marisa Collado	46	X	51.0	1.58	148.0

Para agregar una nueva columna, existe el método `insert()`, que requiere indicar la posición de la nueva columna, el nombre de la nueva columna, y los valores de la misma.

Vamos a crear una lista llamada **direccion** con 14 valores, para cada una de las personas del DataFrame, y luego agregarla en la posición 3:

```
# Valores de la nueva columna
direccion = ["CABA", "Bs As", "Bs As", "Bs As", "CABA", "Bs As", "CABA", "CABA", "CABA", "CA

# Insertar la columna "direccion" en la posición 3 de columnas:
df.insert(3, "direccion", direccion)
df.head() # para que veamos que la columna nueva se agregó
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol
0	José Martínez	18	H	CABA	85.0	1.79	182.0
1	Rosa Díaz	32	M	Bs As	65.0	1.73	232.0
2	Javier Garcíaz	24	H	Bs As	NaN	1.81	191.0
3	Carmen López	35	M	Bs As	65.0	1.70	200.0
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0

Para agregar una nueva columna también podemos hacerlo directamente, como si fuera un diccionario: `df['nueva_columna'] = valores`.

Por ejemplo, supongamos que queremos ingresar una columna con el índice de masa corporal de las personas., que se calcula de la siguiente manera:

$$IMC = \frac{Peso(kg)}{Altura(m)^2}$$

Esto podemos hacerlo directamente trabajando sobre las columnas, como trabajábamos sobre los arrays de NumPy; y guardando el resultado en una nueva columna llamada "IMC".

```
# Crear la columna "IMC"
df["IMC"] = df["peso"] / df["altura"]**2
df.head()
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
0	José Martínez	18	H	CABA	85.0	1.79	182.0	26.528510
1	Rosa Díaz	32	M	Bs As	65.0	1.73	232.0	21.718066
2	Javier Garcíaz	24	H	Bs As	NaN	1.81	191.0	NaN
3	Carmen López	35	M	Bs As	65.0	1.70	200.0	22.491349
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418

Esto lo que va a hacer es tomar todos los valores de peso y altura de cada fila, calcular el IMC y guardarlo en una nueva columna de la línea, bajo el nombre "IMC".

De manera similar, se puede crear la columna **dni** sin utilizar `insert()`, usando la lista **dni**:

```
df_copy = df.copy() # Hacemos una copia, para que no nos afecte el original
dni = [12345678, 23456789, 34567890, 45678901, 56789012, 67890123, 78901234, 89012345, 90123
df_copy["dni"] = dni
df_copy.head()
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC	dni
0	José Martínez	18	H	CABA	85.0	1.79	182.0	26.528510	12345678
1	Rosa Díaz	32	M	Bs As	65.0	1.73	232.0	21.718066	23456789
2	Javier Garcíaz	24	H	Bs As	NaN	1.81	191.0	NaN	34567890
3	Carmen López	35	M	Bs As	65.0	1.70	200.0	22.491349	45678901
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418	56789012

Por lo que vemos que no es indispensable usar `insert` para poder agregar una nueva columna, pero `insert` nos permite decir *en dónde* queremos agregarla.

Para eliminar una fila `(axis=0)` o columna `(axis=1)`, se utiliza `drop()`:

```
# Elimino una columna llamada "direccion". También podría hacer: `del df["direccion"]`
df = df.drop('direccion', axis=1)
```

```
# Elimino la fila 14
df = df.drop(14, axis=0)
```

6.4.2.6 Insertar filas

Para agregar una nueva fila, se utiliza `loc[]`, que nos pide indicar el índice y los valores de la misma. Para ello, creamos una lista llamada **nueva_fila** con valores para cada columna del DataFrame.

```
# Valores de la nueva fila
nueva_fila = ['Carlos Rivas', 30, 'H', "CABA", 70.0, 1.75, 203.0, 22]

# Insertamos al final del DataFrame
largo = len(df.index) # o también: largo = df.shape[0], para obtener la cantidad de filas
df.loc[largo] = nueva_fila
df.tail() # Para ver que se agregó al final
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
10	Macarena Álvarez	53	M	CABA	55.0	1.62	262.0	20.957171
11	José Sanz	58	H	Bs As	78.0	1.87	198.0	22.305471
12	Miguel Gutiérrez	27	H	CABA	109.0	1.98	210.0	27.803285
13	Carolina Moreno	20	M	CABA	61.0	1.77	194.0	19.470778
14	Carlos Rivas	30	H	CABA	70.0	1.75	203.0	22.000000

> ¿Por qué usamos 'len' arriba?
>
> A diferencia de Python, podemos agregar una fila en una posición que no existe aún. Por eso arriba pudimos hacer `df.loc[largo]`.

6.4.2.7 Modificar un valor

Finalmente, **para cambiar un valor determinado** también se utiliza `loc[]`, como por ejemplo, agregar el peso de Javier García (tercera fila), que antes tenía `NaN`:

```
df.loc[2, 'peso'] = 92
df.head()
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
0	José Martínez	18	H	CABA	85.0	1.79	182.0	26.528510
1	Rosa Díaz	32	M	Bs As	65.0	1.73	232.0	21.718066
2	Javier Garcíaz	24	H	Bs As	92.0	1.81	191.0	NaN
3	Carmen López	35	M	Bs As	65.0	1.70	200.0	22.491349
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418

Para transformar los valores de una columna entera, podemos utilizar `map()` pasando un diccionario del estilo `{valor_viejo: valor_nuevo}`. Por ejemplo, modificar la columna "genero" reemplazando "H" por "M", "M" por "F":

```
df['genero'] = df['genero'].map({'H': 'M', 'M': 'F', 'X': 'X'})
df.head()
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
0	José Martínez	18	M	CABA	85.0	1.79	182.0	26.528510
1	Rosa Díaz	32	F	Bs As	65.0	1.73	232.0	21.718066
2	Javier Garcíaz	24	M	Bs As	92.0	1.81	191.0	NaN
3	Carmen López	35	F	Bs As	65.0	1.70	200.0	22.491349
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418

Otra manera sería utilizando `replace()`. Por ejemplo, en la columna "direccion" vamos a modificar "Bs As" por "Buenos Aires".

```
df['direccion'] = df['direccion'].replace('Bs As', 'Buenos Aires')
df.head()
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
0	José Martínez	18	M	CABA	85.0	1.79	182.0	26.528510
1	Rosa Díaz	32	F	Buenos Aires	65.0	1.73	232.0	21.718066
2	Javier Garcíaz	24	M	Buenos Aires	92.0	1.81	191.0	NaN
3	Carmen López	35	F	Buenos Aires	65.0	1.70	200.0	22.491349
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418

6.4.2.8 Filtrar un Dataframe

Para filtrar los elementos de un DataFrame se suelen utilizar condiciones lógicas de la siguiente forma: `DataFrame[ condicion ]`.

Por ejemplo:

```
# Seleccionar aquellas personas menores de 40 años
# La condición es que la columna 'edad' del dataframe tenga valor menor a 40
df[ df['edad'] < 40 ]
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
0	José Martínez	18	M	CABA	85.0	1.79	182.0	26.528510
1	Rosa Díaz	32	F	Buenos Aires	65.0	1.73	232.0	21.718066
2	Javier Garcíaz	24	M	Buenos Aires	92.0	1.81	191.0	NaN
3	Carmen López	35	F	Buenos Aires	65.0	1.70	200.0	22.491349
7	Pilar González	22	F	CABA	60.0	1.66	NaN	21.773842
8	Pedro Tenorio	35	M	CABA	90.0	1.94	241.0	23.913275
12	Miguel Gutiérrez	27	M	CABA	109.0	1.98	210.0	27.803285
13	Carolina Moreno	20	F	CABA	61.0	1.77	194.0	19.470778
14	Carlos Rivas	30	M	CABA	70.0	1.75	203.0	22.000000

Cuando se requieren múltiples condiciones, se puede adicionar usando símbolos como `&` para **and** y `|` para **or**. Por ejemplo:

```
# Seleccionar aquellas personas de genero femenino y menores de 40 años:
df[ (df['edad'] < 40) & (df['genero'] == 'F') ]
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
1	Rosa Díaz	32	F	Buenos Aires	65.0	1.73	232.0	21.718066
3	Carmen López	35	F	Buenos Aires	65.0	1.70	200.0	22.491349
7	Pilar González	22	F	CABA	60.0	1.66	NaN	21.773842
13	Carolina Moreno	20	F	CABA	61.0	1.77	194.0	19.470778

```
# Seleccionar aquellas personas cuyo peso es 60kg o 90kg:
df[(df['peso'] == 60.0) | (df['peso'] == 90.0)]
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
7	Pilar González	22	F	CABA	60.0	1.66	NaN	21.773842
8	Pedro Tenorio	35	M	CABA	90.0	1.94	241.0	23.913275

También puedo filtrar las fillas por el valor `NaN`.
Para eso, se utiliza la función `isnull()` que devuelve `True` si el valor de la columna es nulo o NaN. Por ejemplo:

```
df['IMC'].isnull()
```

```
0     False
1     False
2      True
3     False
4     False
5     False
6     False
7     False
8     False
9     False
10    False
11    False
12    False
13    False
14    False
Name: IMC, dtype: bool
```

Para visualizar aquellas filas donde el índice de masa corporal es nulo, filtramos:

```
df[df['IMC'].isnull()]
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
2	Javier Garcíaz	24	M	Buenos Aires	92.0	1.81	191.0	NaN

El método opuesto es `notnull()`, que devuelve `True` si el valor de la columna no es nulo o NaN. Por ejemplo:

```
df[df['IMC'].notnull()]
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
0	José Martínez	18	M	CABA	85.0	1.79	182.0	26.528510
1	Rosa Díaz	32	F	Buenos Aires	65.0	1.73	232.0	21.718066
3	Carmen López	35	F	Buenos Aires	65.0	1.70	200.0	22.491349
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418
5	Antonio Ruiz	68	M	Buenos Aires	66.0	1.74	249.0	21.799445
6	Antonio Fernández	51	M	CABA	62.0	1.72	276.0	20.957274
7	Pilar González	22	F	CABA	60.0	1.66	NaN	21.773842
8	Pedro Tenorio	35	M	CABA	90.0	1.94	241.0	23.913275
9	Santiago Manzano	46	X	CABA	75.0	1.85	280.0	21.913806
10	Macarena Álvarez	53	F	CABA	55.0	1.62	262.0	20.957171
11	José Sanz	58	M	Buenos Aires	78.0	1.87	198.0	22.305471
12	Miguel Gutiérrez	27	M	CABA	109.0	1.98	210.0	27.803285
13	Carolina Moreno	20	F	CABA	61.0	1.77	194.0	19.470778
14	Carlos Rivas	30	M	CABA	70.0	1.75	203.0	22.000000

6.4.2.9 Ordenando y Contando

A continuación, se muestra una lista con métodos que resultan muy útiles a la hora de analizar datos:

Método	Descripción
`sort_values(by, ascending)`	Ordena el DataFrame considerando los valores de la o las columnas determinadas y devuelve un DataFrame nuevo (no modifica el original)

Método	Descripción
`value_counts()`	Indica los valores únicos de una determinada columna y el número de veces que aparece en ella

6.4.2.9.1 Sort value:

Para utilizar la función `sort_values(by, ascending)`, se debe indicar en el parámetro `by` una lista con las columnas para ordenar el DataFrame y en `ascending`, `True` si el orden deseado es creciente o `False` para decreciente.

En el siguiente ejemplo ordenamos por "nombre y apellido" en forma alfabética:

```
df.sort_values(by=['nombre y apellido'], ascending=[True])
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
6	Antonio Fernández	51	M	CABA	62.0	1.72	276.0	20.957274
5	Antonio Ruiz	68	M	Buenos Aires	66.0	1.74	249.0	21.799445
14	Carlos Rivas	30	M	CABA	70.0	1.75	203.0	22.000000
3	Carmen López	35	F	Buenos Aires	65.0	1.70	200.0	22.491349
13	Carolina Moreno	20	F	CABA	61.0	1.77	194.0	19.470778
2	Javier Garcíaz	24	M	Buenos Aires	92.0	1.81	191.0	NaN
0	José Martínez	18	M	CABA	85.0	1.79	182.0	26.528510
11	José Sanz	58	M	Buenos Aires	78.0	1.87	198.0	22.305471
10	Macarena Álvarez	53	F	CABA	55.0	1.62	262.0	20.957171
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418
12	Miguel Gutiérrez	27	M	CABA	109.0	1.98	210.0	27.803285
8	Pedro Tenorio	35	M	CABA	90.0	1.94	241.0	23.913275
7	Pilar González	22	F	CABA	60.0	1.66	NaN	21.773842
1	Rosa Díaz	32	F	Buenos Aires	65.0	1.73	232.0	21.718066
9	Santiago Manzano	46	X	CABA	75.0	1.85	280.0	21.913806

¿Qué ocurre cuando ordenamos siguiendo varias columnas? Los valores del DataFrame se ordenan siguiendo la primera columna en primer lugar, luego la segunda, y así sucesivamente.

```
df.sort_values(by=['genero', 'nombre y apellido'], ascending=[True, True])
```

	nombre y apellido	edad	genero	direccion	peso	altura	colesterol	IMC
3	Carmen López	35	F	Buenos Aires	65.0	1.70	200.0	22.491349
13	Carolina Moreno	20	F	CABA	61.0	1.77	194.0	19.470778
10	Macarena Álvarez	53	F	CABA	55.0	1.62	262.0	20.957171
7	Pilar González	22	F	CABA	60.0	1.66	NaN	21.773842
1	Rosa Díaz	32	F	Buenos Aires	65.0	1.73	232.0	21.718066
6	Antonio Fernández	51	M	CABA	62.0	1.72	276.0	20.957274
5	Antonio Ruiz	68	M	Buenos Aires	66.0	1.74	249.0	21.799445
14	Carlos Rivas	30	M	CABA	70.0	1.75	203.0	22.000000
2	Javier Garcíaz	24	M	Buenos Aires	92.0	1.81	191.0	NaN
0	José Martínez	18	M	CABA	85.0	1.79	182.0	26.528510
11	José Sanz	58	M	Buenos Aires	78.0	1.87	198.0	22.305471
12	Miguel Gutiérrez	27	M	CABA	109.0	1.98	210.0	27.803285
8	Pedro Tenorio	35	M	CABA	90.0	1.94	241.0	23.913275
4	Marisa Collado	46	X	CABA	51.0	1.58	148.0	20.429418
9	Santiago Manzano	46	X	CABA	75.0	1.85	280.0	21.913806

En el ejemplo de arriba, primero se ordena de manera creciente por genero, resultando en tres grupos: “genero” = “F”, “M” y “X”. Luego, cada uno de esos grupos se ordena por “nombre y apellido” de forma creciente.

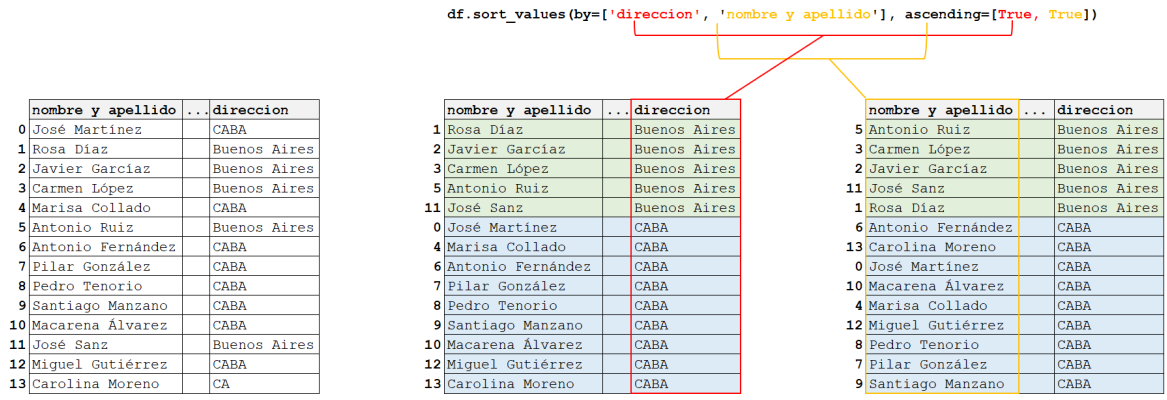


Figure 6.2: Ejemplo de sort_values()

6.4.2.9.2 Value Count:

Utilizando `value_counts()` se pueden contar las filas en cada grupo según “direccion”. Es decir, cuenta cuántas apariciones hay de cada valor en una columna.

```
df['direccion'].value_counts()
```

```
direccion
CABA            10
Buenos Aires     5
Name: count, dtype: int64
```

6.4.3 Conclusiones

Pandas nos permite trabajar con datos de una manera muy sencilla y eficiente. Nos permite importar datos de distintas fuentes, limpiarlos, transformarlos y analizarlos. Además, nos permite visualizar los datos de una manera muy amigable, lo que nos va a permitir entender mejor los datos con los que estamos trabajando.

6.5 Matplotlib

> **Note**
>
> Para Matplotlib también vamos a usar Google Colab

Matplotlib es probablemente la biblioteca de Python más usada para crear gráficos, también llamados **plots**. Esta biblioteca provee una forma rápida de graficar datos en varios formatos de alta calidad que pueden ser compartidos y/o publicados.

El primer paso es importar la biblioteca. Por convención:

```
import matplotlib.pyplot as plt
```

6.5.1 Introducción

Para crear un gráfico con matplotlib, se deben seguir los siguientes pasos:

1. **Crear la figura** que contendrá el gráfico, utilizando las funciones `subplots()` o `figure()`. Se recomienda la primera, es la que va a utilizarse en la materia.
2. **Graficar los datos**, utilizando distintas funciones dependiendo del tipo de gráfico que se desea realizar:

Función	Tipo de Gráfico
`plot()`	Gráfico de línea
`scatter()`	Gráfico de puntos
`bar()`	Gráfico de barras verticales
`barh()`	Gráfico de barras horizontales
`pie()`	Gráfico de torta

3. **Personalizar el gráfico**. Este paso es muy recomendado para lograr un mejor entendimiento de la visualización. Esto incluye agregar etiquetas a los ejes, títulos, leyendas, etc. También podemos modificar el aspecto de las líneas, los puntos, los colores, y más (esto se deja en el final del apunte, y es opcional).
4. **Mostrar el gráfico**, utilizando la función `show()`

```
plt.show()
```

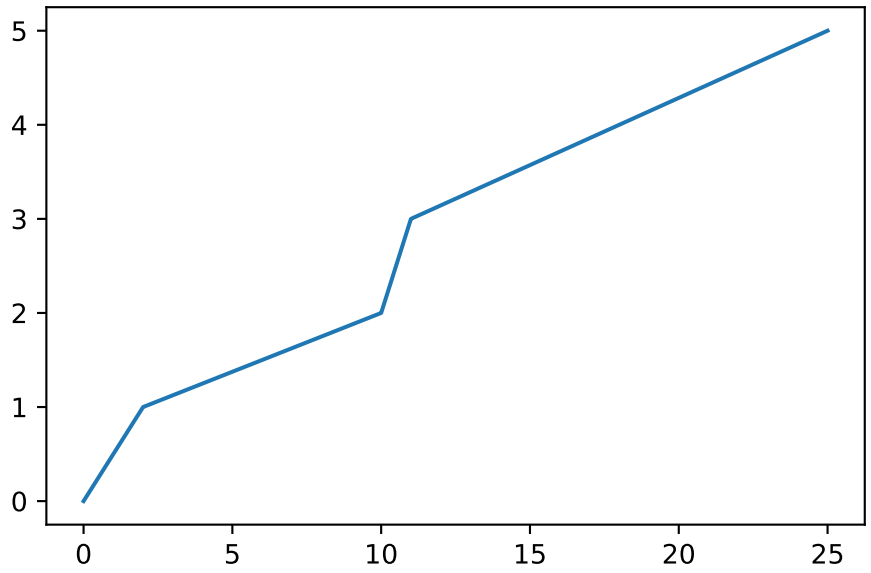
6.5.2 Creando una figura

Para crear una figura, la función de gráfico recibe los datos a graficar y los parámetros necesarios para personalizarlo.

6.5.2.1 Gráfico de línea

El gráfico de línea permite visualizar cambios en los valores lo largo de un rango continuo (tendencias), como puede ser el tiempo o la distancia.
Para crear un gráfico de línea, se utiliza la función `plot()`.

```
# Grafico elemental
x = [0,2,10,11,18,25]
y = [0,1,2,3,4,5]

fig, ax = plt.subplots()

# Gráfico de línea
ax.plot(x, y)
plt.show()
```

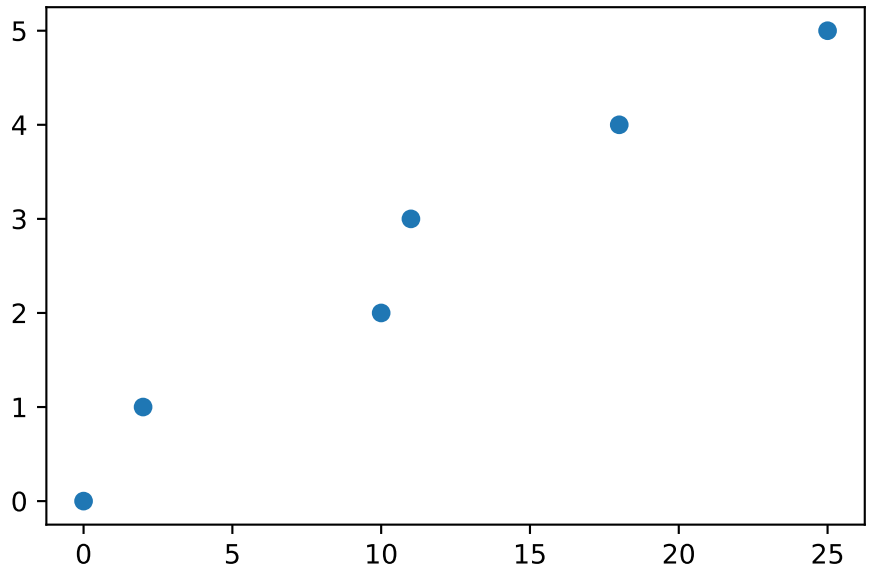


En este caso, se creó un gráfico de línea con los valores de **x** e **y**, tal que los puntos (0,0), (2,1), (10,2), (11,3), (18,4) y (25,5) están unidos por una línea recta.

6.5.2.2 Gráfico de puntos

El gráfico de dispersión o puntos permite visualizar la relación entre las variables.
Para crearlo, se utiliza la función `scatter()`:

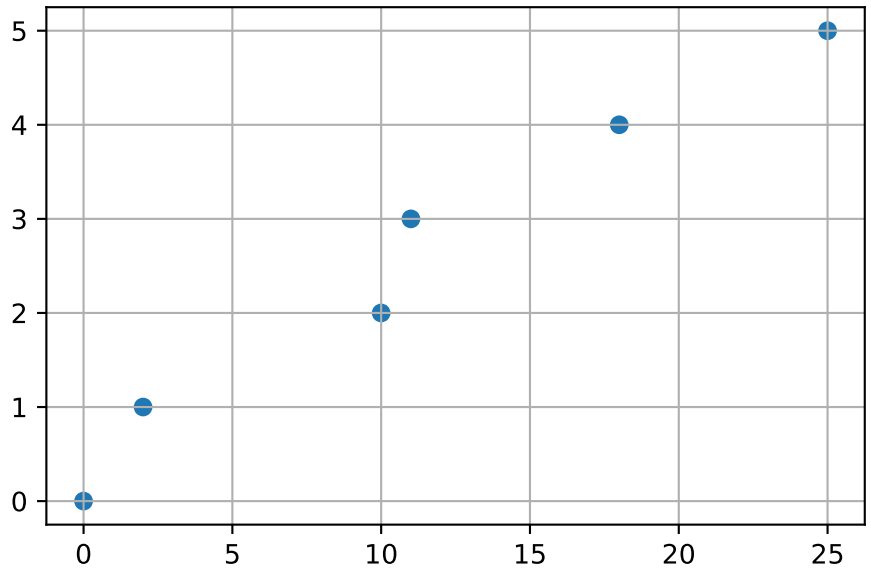
```
# Gráfico de puntos
fig2, ax2 = plt.subplots()
ax2.scatter(x, y)
plt.show()
```



! Grillas

¿Ves cómo en el gráfico de puntos quizás no se entiende bien la ubicación de cada punto? Esto es porque no tenemos una guía que nos ayude. Para eso, vamos a agregar una grilla. La grilla se puede agregar con al función `grid()`, y es una buena forma de darle legibilidad a un gráfico como puede ser el de línea y el de puntos.

```
# Gráfico de puntos
fig, ax = plt.subplots()
ax.scatter(x, y)
ax.grid()
plt.show()
```

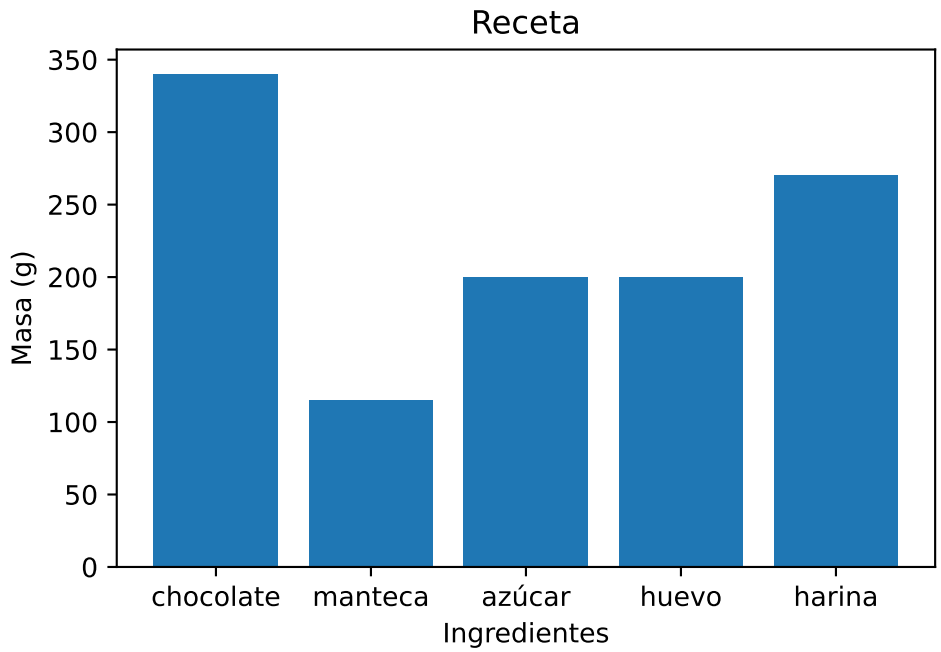


6.5.2.3 Gráficos de Barras

El gráfico de barras permite visualizar proporciones, comparando dos o más valores entre sí. Para crearlo, se utiliza la función `bar()`, la cual primero recibe, en primer lugar, las etiquetas de las barras que se van a mostrar y en segundo lugar, la altura correspondiente a cada una de estas barras.

En el caso de este tipo de gráfico, **no hace falta que los valores de las etiquetas sean numéricos**.

```
peso = [340, 115, 200, 200, 270]
ingredientes = ['chocolate', 'manteca', 'azúcar', 'huevo', 'harina']

fig, ax = plt.subplots()

ax.bar(ingredientes, peso) # Acá podría usarse también barh

ax.set_xlabel('Ingredientes')
ax.set_ylabel('Masa (g)')

ax.set_title("Receta")

plt.show()
```

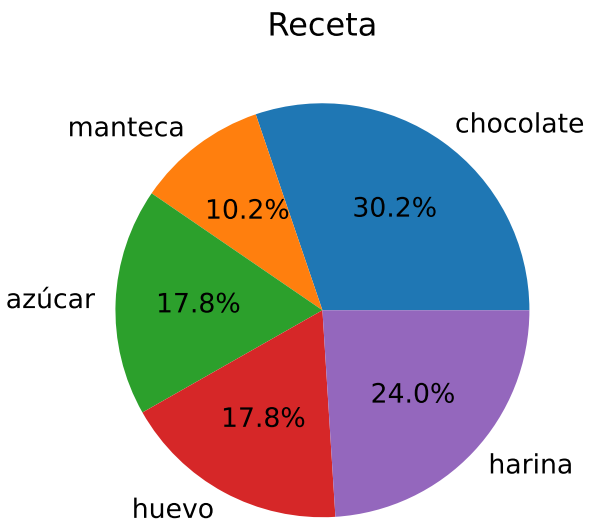


6.5.2.4 Gráfico de torta

Finalmente, tenemos el gráfico de torta. El gráfico de torta, como el de barras, permite visualizar y comparar proporciones pero de manera circular y como partes de un todo.

Para crearlo, se utiliza la función `pie()`, que recibe los valores de las porciones y las etiquetas de cada una. Para las etiquetas, se debe indicar la variable `label`, y si además queremos que se muestren los porcentajes, se puede utilizar `autopct='%1.1f%%'`. `'%1.1f%%'` significa que se mostrará un decimal y el símbolo de porcentaje.

```
peso = [340, 115, 200, 200, 270]
ingredientes = ['chocolate', 'manteca', 'azúcar', 'huevo', 'harina']

fig, ax = plt.subplots()

ax.pie(peso, labels= ingredientes, autopct='%1.1f%%')

ax.set_title("Receta")

plt.show()
```

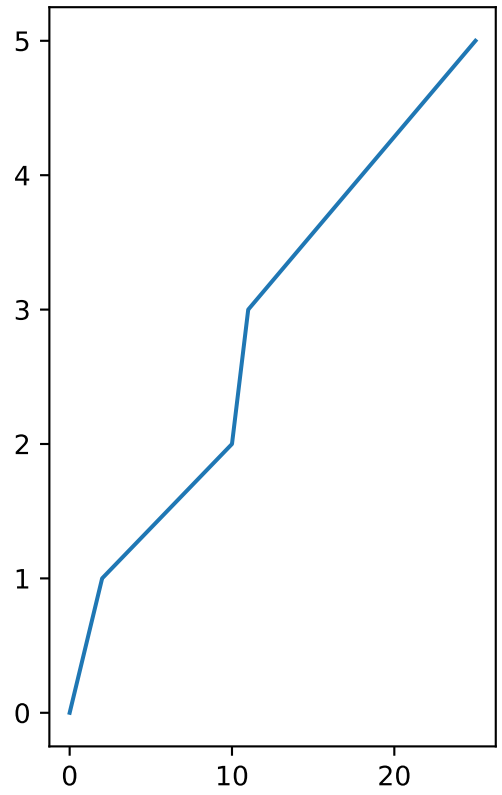


6.5.2.5 Cambio de Tamaño

Podemos establecer el tamaño de la figura con el parámetro `figsize` dentro de la función `subplots()`. Este parámetro recibe una tupla con dos valores: el ancho y el alto de la figura, en pulgadas.

Veamos este ejemplo: **x** es el tiempo medido en minutos e **y** una distancia en metros, entonces:

```
x = [0,2,10,11,18,25]   # Tiempo (min)
y = [0,1,2,3,4,5]       # Distancia (m)

fig, ax = plt.subplots(figsize=(3, 5))

ax.plot(x, y)

plt.show()
```



6.5.3 Títulos

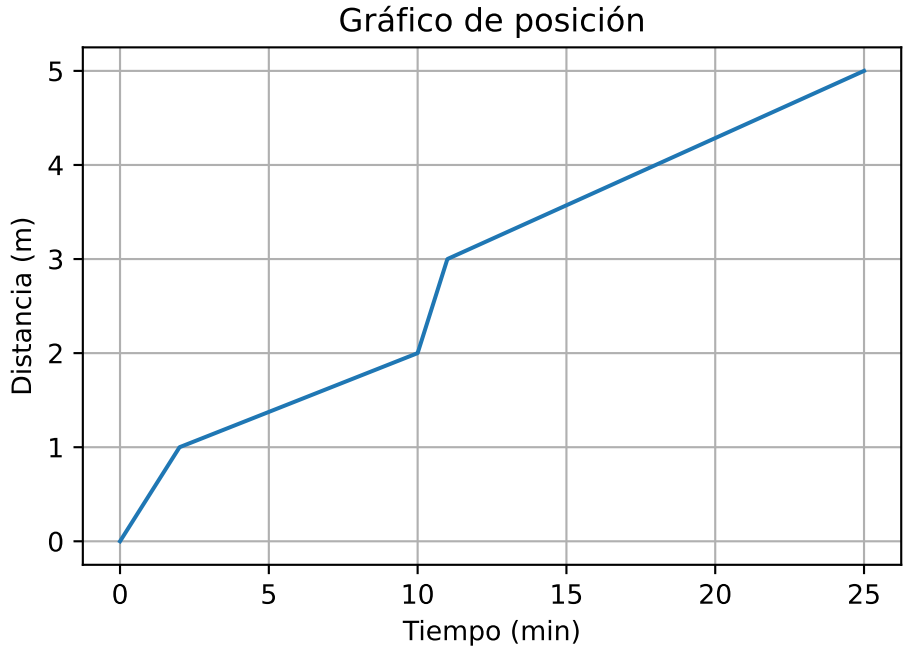
Así como los nombres de las variables en nuestro código, es importante que nuestro gráfico tenga un Título descriptivo que nos ayude a entender qué es lo que estamos viendo. Lo mismo pasa con los ejes: queremos poder entender qué representa cada uno.

Para setear un título, vamos a usar la función `set_title()`. Para los ejes, usaremos `set_xlabel()` y `set_ylabel()`. Cada una recibe un string que se usará como etiqueta del eje X, etiqueta del eje Y o título, respectivamente.

Siguiendo el ejemplo anterior, vamos a agregar títulos a los ejes y al gráfico:

```
x = [0,2,10,11,18,25]   # Tiempo (min)
y = [0,1,2,3,4,5]       # Distancia (m)

fig, ax = plt.subplots()

ax.plot(x, y)
```

```
# Mostrar el título del gráfico
ax.set_title("Gráfico de posición")

# Mostrar el título de los ejes
ax.set_xlabel('Tiempo (min)')
ax.set_ylabel('Distancia (m)')


ax.grid()
plt.show()
```

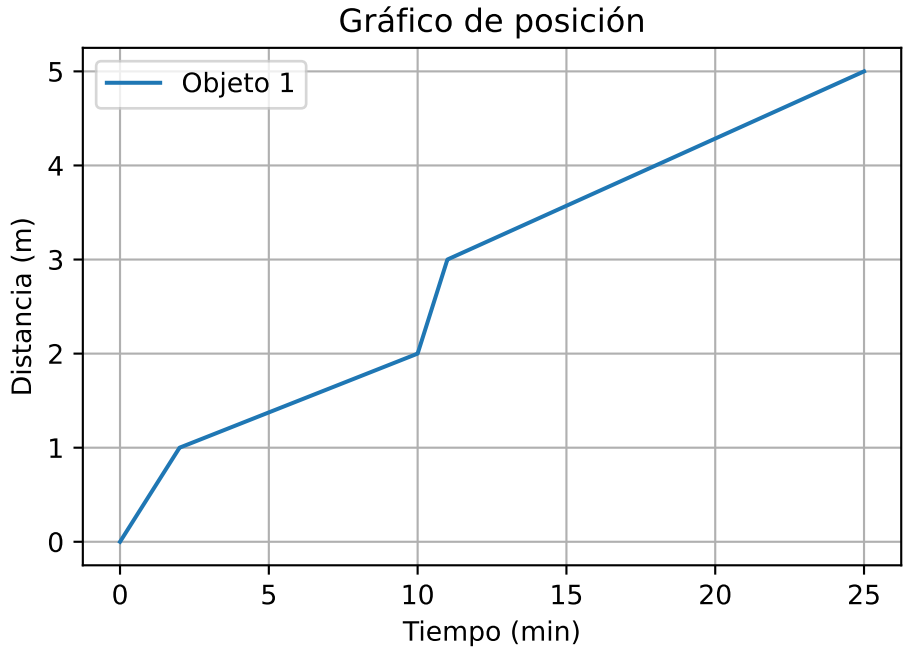


6.5.4 Referencias

El gráfico con el que estamos trabajando sólo tiene una línea, pero si contara con más de una (lo vamos a ver más adelante en este apunte), el uso de referencias sería muy importante para lograr el entendimiento del mismo. Para rotular las líneas, dentro de `plot()` se debe definir la referencia como `label`. Luego se coloca `legend()`

```
x = [0,2,10,11,18,25]    # Tiempo (min)
y = [0,1,2,3,4,5]        # Distancia (m)
```

```python
fig, ax = plt.subplots()

ax.plot(x, y, label='Objeto 1') # Agregar el label

ax.set_title("Gráfico de posición")
ax.set_xlabel('Tiempo (min)')
ax.set_ylabel('Distancia (m)')

# Agregar la refencia
ax.legend()

ax.grid()
plt.show()
```



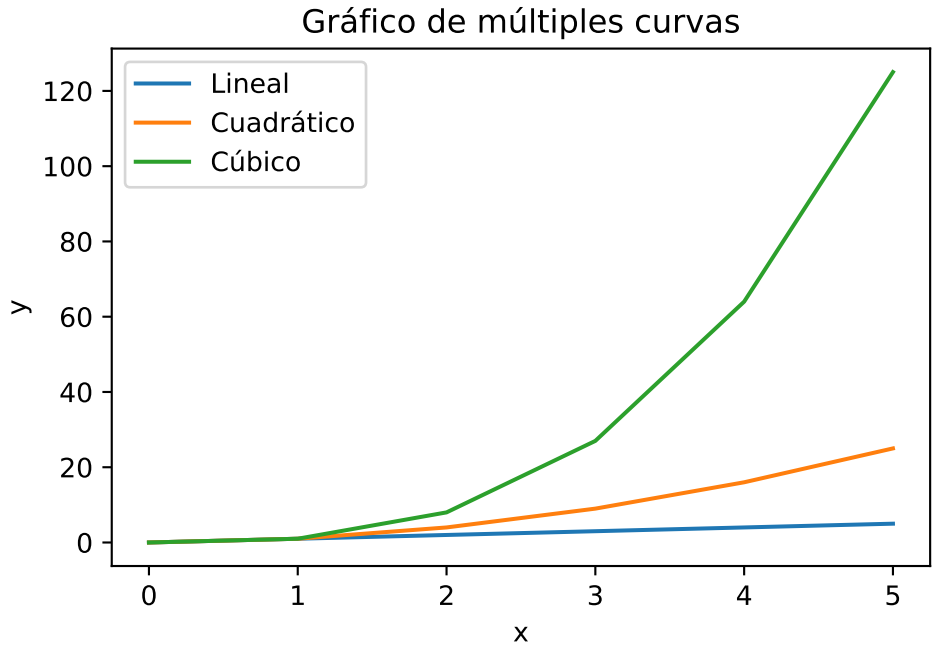
6.5.5 Gráficos múltiples

En los casos anteriores, creamos siempre un sólo gráfico con una curva, en una figura. Pero **También podríamos graficar varias curvas en un mismo gráfico**.

Para esto vamos a seguir el mismo procedimiento que antes, pero vamos a agregar más de un `plot()` al mismo axes. También vamos a darles un label a cada uno, y luego vamos a

agregar una leyenda. Automáticamente, al estar en el mismo axes, matplotlib los va a agregar al mismo gráfico con distintos colores.

```
# Valores que se desean graficar
x = [0, 1, 2, 3, 4, 5]
y_linear = [0, 1, 2, 3, 4, 5]
y_quadratic = [0, 1, 4, 9, 16, 25]
y_cubic = [0, 1, 8, 27, 64, 125]

fig, ax = plt.subplots()

# Usamos distintos tipos de y, con distintas labels
ax.plot(x, y_linear, label='Lineal')
ax.plot(x, y_quadratic, label='Cuadrático')
ax.plot(x, y_cubic, label='Cúbico')

ax.set_title("Gráfico de múltiples curvas")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()

plt.show()
```



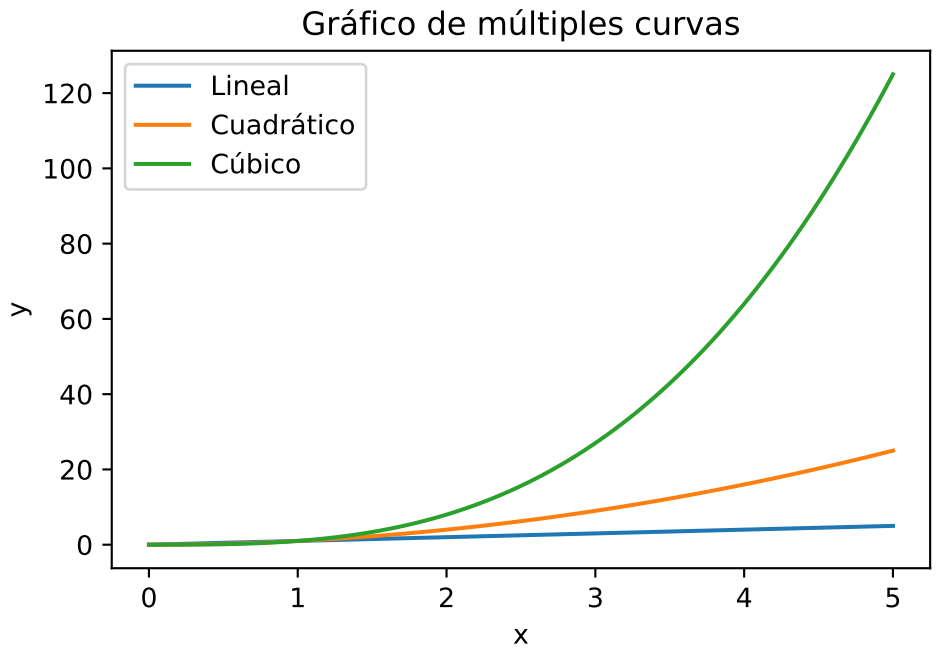
Note que se agregan nuevos datos al mismo axes, por lo que siempre usamos `plot()` pero con distintos valores de `y`. Asimismo, se estableció un tamaño de la figura con `figsize=(width, height)`

6.5.6 Uniendo Bibliotecas

6.5.6.1 Matplotlib y Numpy

Podemos usar arrays de numpy para simplificarnos el trabajo.
Repetimos el gráfico anterior pero usando numpy, y vamos a ver que pudimos obtener muchos más valores sin tener que calcularlos nosotros y escribirlos en una lista de Python.

```
import numpy as np

x = np.linspace(0, 5, 100) # Creamos un array de 100 valores entre 0 y 5
y_linear = x # usamos x
y_quadratic = x**2 # usamos x al cuadrado
y_cubic = x**3 # usamos x al cubo

fig, ax = plt.subplots()

# Usamos distintos tipos de y
ax.plot(x, y_linear, label='Lineal')
ax.plot(x, y_quadratic, label='Cuadrático')
ax.plot(x, y_cubic, label='Cúbico')

ax.set_title("Gráfico de múltiples curvas")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()

plt.show()
```



6.5.6.2 Matplotlib y Pandas

También podemos usar columnas de Pandas como datos para crear un gráfico.

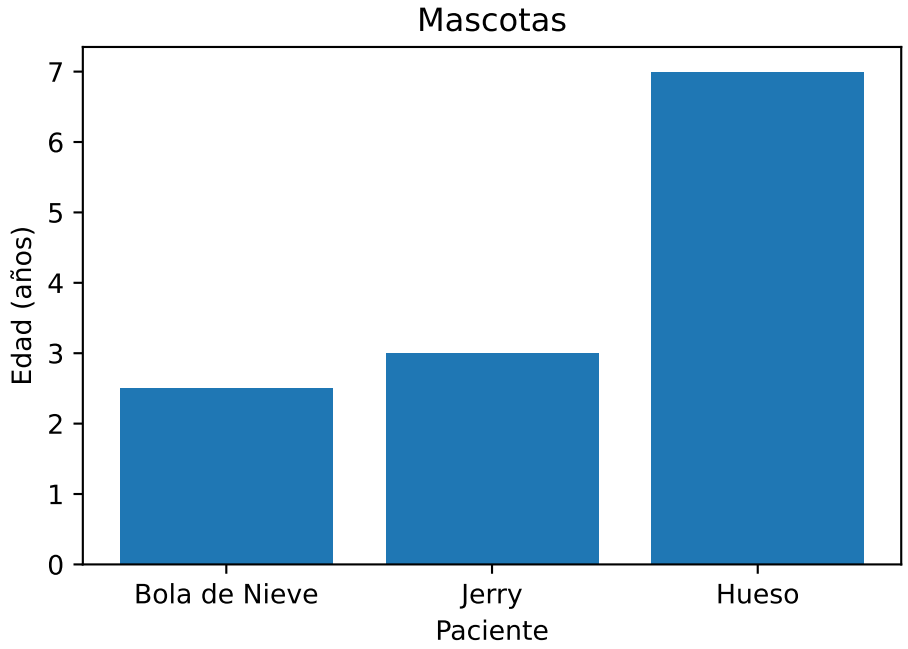
Supongamos el siguiente dataframe de los datos de las mascotas en una veterinaria:

```
data = { 'nombre': ['Bola de Nieve', 'Jerry', 'Hueso'],
         'especie': ['gato','chinchilla', 'perro'],
         'edad': [2.5, 3, 7],
         'visitas': [1, 3, 2],
         'prioridad': ['si', 'si', 'no']}

df = pd.DataFrame(data)
df
```

	nombre	especie	edad	visitas	prioridad
0	Bola de Nieve	gato	2.5	1	si
1	Jerry	chinchilla	3.0	3	si
2	Hueso	perro	7.0	2	no

Podemos ahora crear un gráfico usando las columnas del dataframe:

```python
# Determino las columnas del DataFrame que queremos graficar
x_values = df['nombre']
y_values = df['edad']

fig, ax = plt.subplots()

ax.bar(x_values, y_values)

ax.set_xlabel('Paciente')
ax.set_ylabel('Edad (años)')

ax.set_title("Mascotas")

plt.show()
```



i ¿Qué pasa si nuestros labels no llegan a verse? (opcional)

Puede pasar, sobre todo si tenemos muchos datos con nombres largos, que nuestros labels no lleguen a verse de forma correcta si los presentamos de forma horizontal.
Por ejemplo:
En esos casos, podemos pedirle a matplotlib que los presente de forma vertical, para que no se superpongan. Para eso, usamos `xticks()` y `rotation`:

`plt.xticks(rotation=90)`.
El ángulo de rotación se mide en grados, por lo que `rotation=90` significa que se rotarán 90 grados. De esta forma, los labels se presentarán de forma vertical. Podríamos también usar otro ángulo, y se mostrarían los labels de forma inclinada.
Esta es la forma en que se visualizan los datos con los labels rotados:

6.5.7 Bonus Track: Personalización (opcional)

> **i** ¿Qué significa opcional?
>
> Significa que esta parte del apunte no es obligatoria. Pero si quieren leerla, les va a permitir hacer gráficos más bonitos y personalizados. También les va a permitir llevarse el conocimiento de qué otras herramientas tienen disponibles, y volver a este apunte a buscar información en un futuro si es que la necesitan.

Esta imagen, fue obtenida de la referencia de matplotlib y resume de manera fácil y visual las modificaciones que podemos hacerla a las figuras creadas.

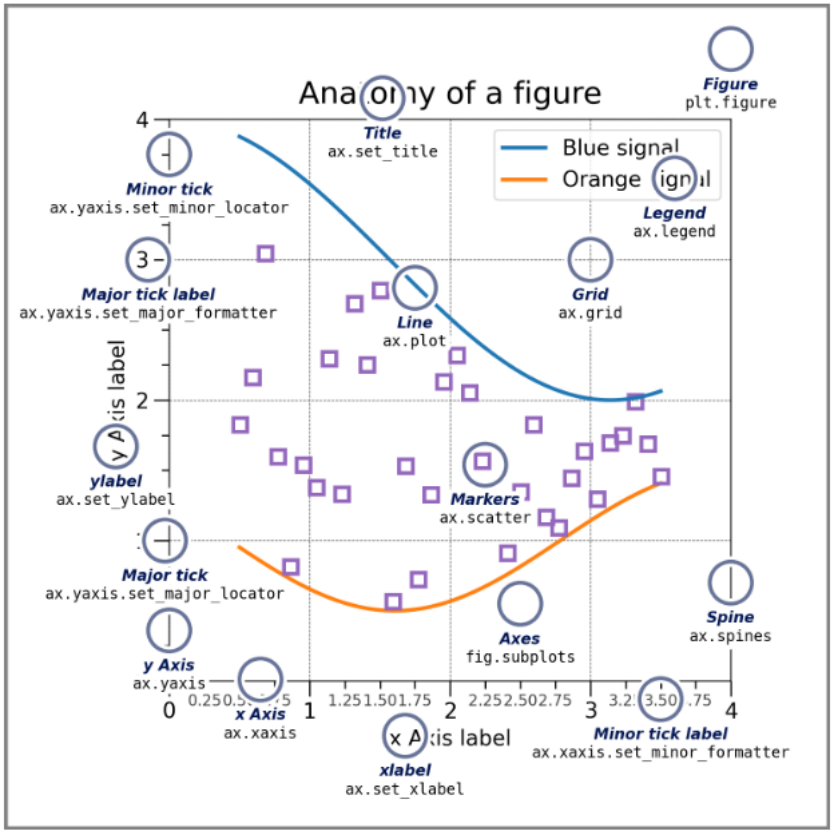


Figure 6.3: Partes de una Figura. Si querés conocer más detalle, podés ingresar a este link.

Como dijimos más arriba, las figuras pueden ser personalizadas de muchas maneras. Algunas son:

- Colores
- Estilos de línea
- Marcadores
- Grilla personalizada

¡y más!

6.5.7.1 Cambiando colores y estilos

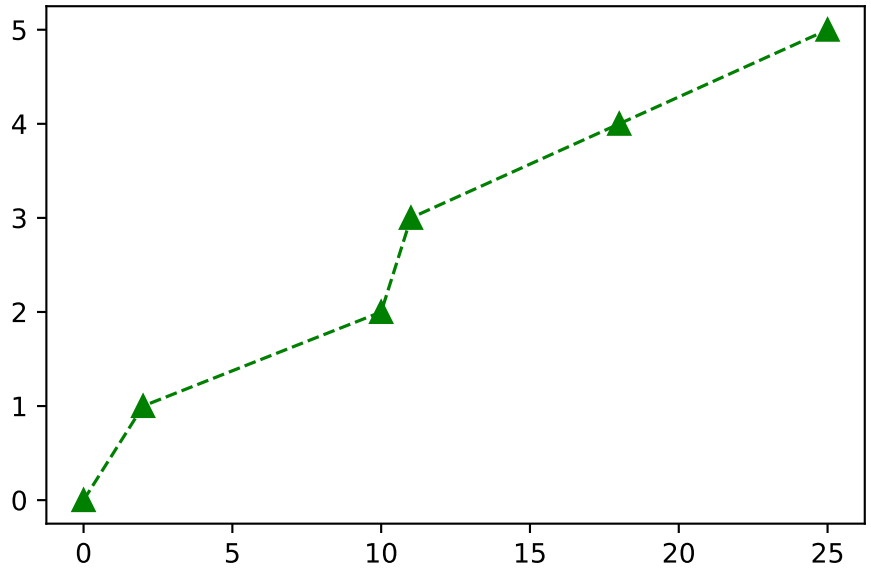
Para cambiar los colores de los gráficos, podemos utilizar los parámetros `color`, `marker`, `linestyle`, `markersize` y `linewidth` dentro de la función `plot()`.

- **color** = nombre del color, por ejemplo: `'blue'`, `'green'`, `'red'`, etc.

- **marker** = forma de los puntos o marcadores, por ejemplo: `'^', 'o', 'v'`, etc.
- **linestyle** = estilo de línea, por ejemplo: `'solid', 'dashed', 'dotted'` o sus equivalentes:`'-', '--', ':'`, entre otros.
- **markersize, linewidth** = con un número, establecemos el tamaño del marcador y el espesor de la línea respectivamente.

Si no le asignamos un valor, se establecen los predefinidos.

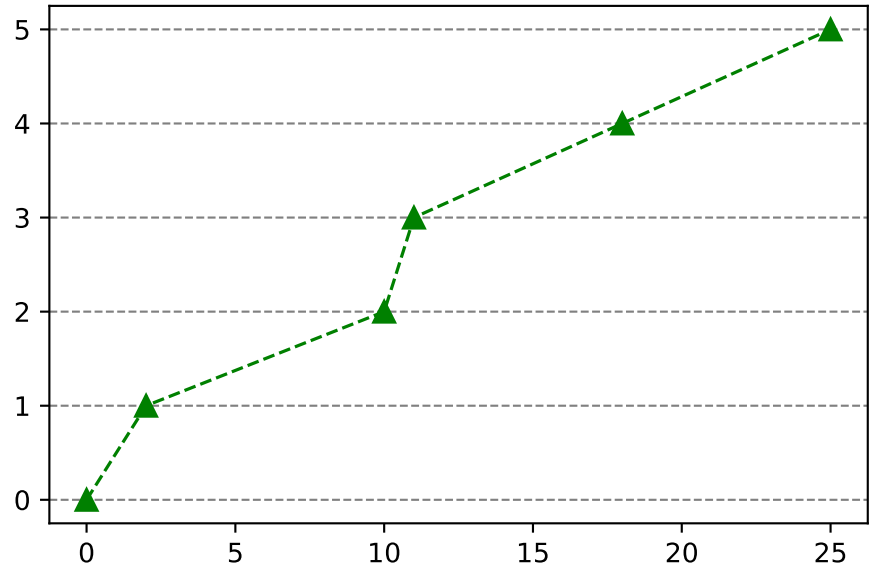
```
x = [0,2,10,11,18,25]   # Tiempo (min)
y = [0,1,2,3,4,5]       # Distancia (m)

fig, ax = plt.subplots()

ax.plot(x, y, color='green', marker='^', linestyle='--', markersize=8, linewidth=1.2)
plt.show()
```



6.5.7.2 Grilla personalizada:

Si deseamos modificarle a una grilla, por ejemplo, el color, el estilo de línea, o sólo queremos ver uno de los ejes, podemos indicarlo utilizando parámetros muy similares a los vistos anteriormente pero en la funcion `grid()`.

```
x = [0,2,10,11,18,25]    # Tiempo (min)
y = [0,1,2,3,4,5]        # Distancia (m)

fig, ax = plt.subplots()

ax.plot(x, y, color='green', marker='^', linestyle='--', markersize=8, linewidth=1.2)

#Grilla modificada
ax.grid(axis = 'y', color = 'gray', linestyle = 'dashed')
plt.show()
```



Con esto finalizamos los temas de la materia.
¡Esperamos que hayas disfrutado de este recorrido!
Recordá que si querés dejarnos feedback podés hacerlo en el formulario que está en la parte de Contacto.

Si te interesa ser docente de la materia, podés ver los requisitos y escribirnos en la sección Ser Docente.

¡Muchos éxitos!

Guía de Ejercicios

Recomendaciones al realizar las guías

- Prestá atención al leer el enunciado. En particular:
  - Si se pide una función que *devuelva* o *calcule* un valor, la función debe tener una función `return`.
  - Si se pide una función que *imprima* un valor, la función debe tener un `print`.
  - Si se pide una función que *pida* o *pregunte* algo al usuario, la función debe tener un `input`.
  - A menos que se diga específicamente "pedirle al usuario", no es necesario que el programa contenga `input`. En todo caso, hacer que la función reciba el o los datos por parámetro.
- Cada ejercicio puede tener muchas soluciones posibles. Una vez que encuentres una solución, en lugar de pasar al siguiente ejercicio, pensá si se te ocurre una solución cuya codificación sea más simple.
- Es muy importante que el código sea lo más claro y legible posible.
  - En particular, nombres de funciones y variables deben ser descriptivos.
  - También prestá atención a los espacios en blanco y a la indentación.
- No documentes en exceso, pero tampoco ahorres documentación necesaria.
- Probá siempre que el código cumpla con lo solicitado.

> **Discord**
>
> La materia usa Discord como plataforma adicional para la resolución de los ejercicios de las guías.
> Tengan a bien leer con atención el mensaje de bienvenida y las reglas de convivencia. Pueden ingresar al servidor a través del siguiente link.

Guía 1: Introducción a la Algoritmia y la Programación

> **Recomendación**
>
> En esta guía nos dedicaremos a introducirnos en los conceptos de programación y algoritmo. Para los primeros seis ejercicios, te recomendamos ver este video para recordar cómo entiende la computadora nuestras instrucciones.

1. Se tiene que explicar a una máquina exactamente cómo servir un vaso de jugo (de los que vienen en cartón) de la heladera. Recordando la definición de algoritmo, hacer una descripción paso a paso de lo que se tiene que hacer y usar para lograr el objetivo. Pista: No vas a necesitar nada de código en este ejercicio, sólo nombrar los pasos.

2. Se tiene que explicar a una máquina exactamente cómo hacer una tostada con queso, pensá qué ingredientes se necesitan con sus cantidades, cómo tiene que ser el espacio de trabajo y los elementos que va a necesitar usar. Recordando la definición de algoritmo, hacer una descripción paso a paso de lo que se tiene que hacer y usar para hacer una tostada con queso. Pista: No vas a necesitar nada de código en este ejercicio, sólo nombrar los pasos.

3. Se te pide que organices una colecta de alimentos no perecederos por la Ciudad de Buenos Aires. Contamos con algunos automóviles y camionetas de voluntarios, un listado de donaciones, listado de los alimentos a donar, la disponibilidad horaria y la dirección en la cual se dejan los alimentos. La colecta se realiza en un solo día. ¿Cómo la organizarías? Pista: No vas a necesitar nada de código en este ejercicio, sólo nombrar los pasos.

4. Tenés que enviar invitaciones personalizadas para tu cumpleaños. Cada invitación tiene que mencionar el nombre de la persona y la cantidad de invitados adicionales con los que puede venir. Contamos con una impresora a la que le das el texto a enviar, un listado con los nombres de los invitados y los invitados adicionales que pueden ellos traer. ¿Cómo redactarías el texto de la invitación? Pista: No vas a necesitar nada de código en este ejercicio, sólo nombrar los pasos.

5. Se te encargó definir qué datos son necesarios para el registro de estudiantes en un curso de inglés. ¿Qué datos crees que deberían ser obligatorios y cuáles opcionales? ¿Y si el curso es de cocina? Pista: No vas a necesitar nada de código en este ejercicio, sólo nombrar los pasos.

6. Contás con un listado de cosas a comprar y tenés que ir a un supermercado que cuenta con distintas góndolas o pasillos. Cada góndola o pasillo puede contar con varios, uno o ninguno de los productos de tu lista. ¿Cuál sería el listado de instrucciones para poder terminar lo más rápido posible? Pista: No vas a necesitar nada de código en este ejercicio, sólo nombrar los pasos.

7. Con el anexo de Google Colab de la Unidad 1, realizá tu primer programa: hacé que se imprima por pantalla un `"¡Hola mundo!"`.

Guía 2: Tipos de Datos, Expresiones y Funciones

1. Guardar el texto “Hola, Mundo!” en una variable e imprimirla por pantalla.
2. Guardar los números 1, 2 y 3 en tres variables distintas e imprimirlos por pantalla.
3. a. Guardar los números 1, 2 y 3 en tres variables distintas y luego sumarlos e imprimir el resultado por pantalla.

   b. Repetir con las distintas operaciones disponibles que se vieron en la unidad 2: resta, multiplicación, división, división entera, resto, potencia; combinando los números entre sí.
4. Crear un programa que le solicite al usuario:

   a. Su nombre y lo imprima por pantalla.
   b. Su edad y la imprima por pantalla.
   c. Su edad, le sume 1, y la imprima por pantalla.
5. Crear un programa que le solicite al usuario un número, y que devuelva el resto obtenido de dividirlo por 2.
   ¿Qué operador vimos para obtener el resto?
6. Escribir un programa que le pida al usuario su año de nacimiento, y que le diga qué edad tiene en el año actual.
7. Crear un programa que le solicite al usuario 5 enteros y que muestre por pantalla el promedio de ellos. Hacerlo de dos formas:

   a. Primero, usando 5 variables para cada entero.
   b. Después, usando una sola variable para almacenar la suma de los 5 enteros. ¿Cómo se te ocurre que podrías hacer?

A partir de ahora y en adelante, todo lo que se nos pida (incluso si dice “programa”) se debe realizar dentro de una o más funciones.

8. Crear una **función** que reciba un número y que devuelva el valor absoluto.
9. Crear una **función** que reciba un número y que devuelva `True` si es par, y `False` si es impar.
10. Crear una **función** que reciba un número y un string, y que devuelva ambos concatenados dentro de un nuevo string.

11. Crear una **función** que reciba dos enteros y que devuelva el resto y el cociente entre ellos.

12. Crear una **función** que le pida al usuario su nombre y apellido, e los imprima con el siguiente formato: "Apellido, Nombre".

13. Hacer una **función** que reciba una palabra y devuelva la cantidad de letras que tiene.

14. a. Hacer una **función** que reciba una palabra y que imprima los primeros 5 caracteres únicamente. Ejemplo: Si se recibe "pensamiento" se debe imprimir "pensa".
    b. Hacer una **función** que reciba una palabra y que imprima sólo los caracteres ubicados en posiciones pares. Ejemplo: Si se recibe "pensamiento" se debe imprimir "pnaino".
    c. Hacer una **función** que reciba una palabra y que imprima la palabra dada vuelta. Ejemplo: Si se recibe "materia" se debe imprimir "airetam".

15. Hacer una **funcion** que reciba una palabra, le borre todas las letras "a" e imprima el resultado por pantalla. Pista: usar una función predefinida de Python. Ejemplo: Si se recibe "casa" se debe imprimir "cs".

16. Analizar qué tipo de dato (o error) se obtiene al hacer las siguientes operaciones:

    a. `5 / 2`
    b. `5 // 2`
    c. `5 % 2`
    d. `5 ** 2`
    e. `5.0 / 2`
    f. `5.0 // 2`
    g. `5.0 % 2`
    h. `5.0 ** 2`
    i. `5 / 2.0`
    j. `5 // 2.0`
    k. `5 % 2.0`
    l. `5 ** 2.0`
    m. `5.0 / 2.0`
    n. `5.0 // 2.0`
    o. `5.0 % 2.0`
    p. `5.0 ** 2.0`
    q. `"Hola" * 2`
    r. `"Hola" + 2`
    s. `"Hola" + "2"`
    t.
    ```
    x = "Hola"
    x += " mundo"
    ```

17. a. Escribir una función que convierta un valor dado en grado Celsius, a Fahrenheit. Recordar que la fórmula para la conversión es: `F = 9/5 * C + 32`.

b. Escribir una función que convierta un valor dado en grados Fahrenheit, a Celsius. Usar la misma fórmula anterior.

18. Escribir una función que calcule el área de un triángulo recibiendo como parámetros su base y su altura.

19. Siendo el cálculo de la norma de un vector v en R^3:

$$||v|| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

Escribir una función que calcule la norma de un vector en R3 recibiendo como parámetros las 3 componentes v_1, v_2 y v_3 del mismo.

20. **Desafío** (no obligatorio): Calcular el área de un rectángulo (alineado con los ejes x e y), dadas sus coordenadas x_1, x_2, y_1 e y_2. Nota: considerar que tiene que funcionar para cualquier cuadrado, independientemente de los cuadrantes en los que tenga sus vértices.

Guía 3: Estructuras de Control

1. Decisiones

1. Escribir una función que, dado un número entero n, calcule si es impar o no.
2. Escribir una implementación propia de la función *abs*, que devuelva el valor absoluto de cualquier valor que reciba. Ejemplo: `mi_abs(5)` devuelve `5` y `mi_abs(-5)` devuelve `5`. Pista: No se puede usar la función predefinida `abs`.
3. Escribir una función que reciba un número y devuelva `True` si es entero y `False` si no lo es. Pista: no se puede usar la función `isinstance`.
4. Escribir una función para determinar si una letra recibida es vocal o no. La misma debe devolver un valor booleano. Luego, escribir una función para determinar si una letra es consonante o no.

   a. Resolver *sin* el uso de `in` ni `not in`.
   b. Resolver *usando* `in` y `not in`.
   c. Resolver para que la función identifique tanto mayúsculas como minúsculas. Pista: investigar los métodos `lower` y `upper` de string.

> Tip: `in` y `not in`
>
> ¿Conocés el uso de `in`?
> Para saber si un elemento está en una lista o en un string, podemos usar `in` y `not in`.
> Por ejemplo:

```
'a' in 'hola'
```

```
True
```

```
'w' in 'hola'
```

```
False
```

```
'w' not in 'hola'
```

```
True
```

```
'casa' in ['cama', 'mesa', 'silla']
```

```
False
```

5. Escribir funciones que resuelvan los siguientes problemas:

    a. Dado un año, que devuelva si es bisiesto. Nota: un año es bisiesto si es un número divisible por 4, pero no si es divisible por 100, excepto que también sea divisible por 400.
    b. Dado un mes y un año, que devuelva la cantidad de días correspondientes.
    c. Pedirle al usuario su día y mes de cumpleaños. El programa debe imprimir un mensaje indicando a qué signo corresponde el usuario.

```
Aries: 21 de marzo al 20 de abril.
Tauro: 21 de abril al 20 de mayo.
Géminis: 21 de mayo al 21 de junio.
Cáncer: 22 de junio al 23 de julio.
Leo: 24 de julio al 23 de agosto.
Virgo: 24 de agosto al 23 de septiembre.
Libra: 24 de septiembre al 22 de octubre.
Escorpio: 23 de octubre al 22 de noviembre.
Sagitario: 23 de noviembre al 21 de diciembre.
Capricornio: 22 de diciembre al 20 de enero.
Acuario: 21 de enero al 19 de febrero.
Piscis: 20 de febrero al 20 de marzo.
```

6. Piedra, papel o tijera: escribir un programa de "Piedra, papel o tijera" tal que sea imposible que el usuario gane. El usuario debe ingresar **R** (piedra), **P** (papel), o **T** (tijera) y la computadora debe siempre ganarle. Ejemplo:

```
¡Piedra (R), papel (P) o tijera (T)!
Ingrese jugada: R
¡Papel! ¡Gané!

¡Piedra (R), papel (P) o tijera (T)!
Ingrese jugada: P
¡Tijera! ¡Gané!

¡Piedra (R), papel (P) o tijera (T)!
Ingrese jugada: T
¡Piedra! ¡Gané!

¡Piedra (R), papel (P) o tijera (T)!
Ingrese jugada: M
Esa jugada no está disponible.
```

7. Suponiendo que el primer día del año fue lunes, escribir una función que reciba un número con el día del año (de 1 a 366) y devuelva el día de la semana que le toca. Por ejemplo: si se recibe '3', debe devolver "miércoles", y si se recibe '9', debe devolver "martes".

2. Ciclos

1. Escribir función que:

   a. Imprima por pantalla todos los números entre 10 y 20.
   b. Salude a todas las personas de esta lista `[Flaminia, Serena, Agustina, Priscila, Sol, Agostina, Iara, Lu]` con el mensaje `"Hola <nombre>! Vamos a aprender a programar"`.
   c. Le pida al usuario que ingrese 5 números y le muestre la suma total de todos ellos.
   d. Imprima por pantalla todos los números entre 100 y 199 que sean divisibles por 7.
   e. Reciba dos números, y recorra todos los números entre ellos, imprimiendo en pantalla si es par o impar. Por ejemplo, recibiendo 1 y 3, debe imprimir:

```
1 es impar
2 es par
3 es impar
```

2. Se quiere hacer un programa para enseñar a los niños las tablas de multiplicar del 1 al 10. Crear una función que reciba un número e imprima por pantalla la tabla de multiplicar de ese número. Ejemplo:

   ```
   mostrar_tablas_para(1)
   ```

   debe imprimir:

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

```
mostrar_tablas_para(-2)
```

debe imprimir:

```
Error: El número debe ser positivo y estar entre 1 y 10
```

3. Crear una función que cante el feliz cumpleaños. Dado un entero, debe imprimir 'Que los cumplas feliz' en distintas líneas por esa cantidad de veces.

4. a. Necesitamos escribir un programa de cobro en el supermercado. La función debe recibir un número entero que representa el monto a pagar y debe permitir al usuario que ingrese valores, hasta que el pago se haya realizado en su totalidad. Además, le debe ir indicando cuánto le queda por pagar. El programa no da vuelto.

   Ejemplo:

```
Su total a pagar es: 500
Ingrese el monto a pagar: 100
Pendientes: 400. Ingrese el monto a pagar: 200
Pendientes: 200. Ingrese el monto a pagar: 200
Pendientes: 0. Gracias por su compra.
```

   b. Hacer que el programa anterior dé vuelto:

   Ejemplo:

```
Su total a pagar es: 500
Ingrese el monto a pagar: 100
Pendientes: 400. Ingrese el monto a pagar: 200
Pendientes: 200. Ingrese el monto a pagar: 300
Pendientes: 0. Su vuelto es: 100. Gracias por su compra.
```

5. Escribir un programa que le pida al usuario que ingrese un número. Para ese número, se imprime la tabla de multiplicar del 1 al 10. Luego, se le vuelve a pedir otro número. Si el usuario ingresa "X", el programa debe terminar. El usuario debe poder ingresar números indefinidamente hasta que ingrese "X". Se puede reutilizar la función del ejercicio 9 de esta guía.

   Ejemplo:

```
Hola! Esto es Tablas de Multiplicar
Ingrese un número o "X" para salir: 1
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
Ingrese un número o "X" para salir: -2
Error: El número debe ser positivo y estar entre 1 y 10
Ingrese un número o "X" para salir: X
¡Adios!
```

6. **Manejo de contraseñas**

   a. Escribir un programa que contenga una contraseña inventada, que le pregunte al usuario la contraseña, y no le permita continuar hasta que la haya ingresado correctamente.
   b. Modificar el programa anterior para que solamente permita una cantidad fija de intentos.
   c. Modificar el programa anterior para que sea una función que devuelva si el usuario ingresó o no la contraseña correctamente, mediante un valor booleano (`True` o `False`).

7. a. Hacer una función que reciba un número del 1 al 10, y luego permita al usuario poder adivinar ese número, ingresando valores repetidamente. Para cada ingreso del usuario, el programa debe indicarle si su número es menor o mayor al número a adivinar. Una vez que el usuario ingresa el número correcto, lo felicita y termina.

   b. Repetir permitiendo únicamente 3 intentos.

   c. Repetir generando el número aleatoriamente de la siguiente forma dentro de la función, sin recibirlo por parámetro:

```python
import random
numero_a_adivinar = random.randint(1, 10)
print(numero_a_adivinar)
```

```
9
```

> **Tip: Bibliotecas**
>
> ¿Sabías que Python tiene muchas bibliotecas que podés usar para hacer cosas más complejas? Por ejemplo, la biblioteca `random` tiene funciones para generar números aleatorios. También hay otras bibliotecas como `Pandas` para trabajar con datos, `Matplotlib` para hacer gráficos, `Numpy` para trabajar con matrices, y muchas más. Vamos a estar viendo estas tres en la última unidad de la materia.
> Una biblioteca es un conjunto de funciones que alguien más escribió y que podemos usar en nuestros programas. Para usar una biblioteca, primero tenemos que importarla. Por ejemplo, para usar la biblioteca `random`, tenemos que poner `import random` al principio de nuestro programa (arriba de todo en nuestro archivo). Luego, podemos usar las funciones de la biblioteca, como `random.randint(1, 10)`.

8. a. Queremos modelar una máquina de sacar juguetes. Debemos hacer una función que reciba un número que representa la cantidad de fichas x que necesita la máquina para funcionar. Se debe imprimir un mensaje en pantalla que indique "Ingresá x fichas para comenzar". El usuario deberá ingresar entonces letras "F", que representan a las fichas. Notar que si se ingresa algo distinto a "F", se ignora.

   Se debe seguir solicitando fichas siempre que no se haya alcanzado la cantidad necesaria para funcionar. Cuando se haya alcanzado la cantidad necesaria, se debe imprimir un mensaje que indique "¡A jugar!". Ejemplo:

   ```
   Ingresá 2 fichas para comenzar: F
   Ingresá 2 fichas para comenzar: B
   Ingresá 2 fichas para comenzar: Hola
   Ingresá 2 fichas para comenzar: F
   ¡A jugar!
   ```

   b. Modificar el programa anterior para que vaya mostrando la cantidad de fichas que faltan para comenzar a jugar. Ejemplo:

   ```
   Ingresá 2 fichas para comenzar: F
   Ingresá 1 fichas para comenzar: B
   Ingresá 1 fichas para comenzar: ficha
   Ingresá 1 fichas para comenzar: F
   ¡A jugar!
   ```

9. Crear una función que calcule si un número es primo o no. Un número es primo cuando solamente es divisible por sí mismo y por 1. Pista: usar el operador módulo `%`.

10. **Desafío** (obligatorio): Crear una función que reciba un número entero e imprima los números primos entre 0 y el número ingresado.

11. **Desafío** (obligatorio):

    a. Crear una función que reciba dos números, y devuelva la suma de todos los números múltiplos de 7 entre esos dos números. Por ejemplo, si recibe 3 y 25, debe devolver 7 + 14 + 21 = 42. Si recibe 3 y 4, debe devolver 0, ya que no hay múltiplos de 7 entre esos dos números.

    b. Repetir calculando el promedio en vez de la suma.

    c. Repetir calculando únicamente el promedio entre los primeros 3 múltiplos de 7 encontrados. Pista: usar `break`.

    d. Repetir calculando únicamente el promedio entre los múltiplos de 7 encontrados que no sean múltiplos de 2. Pista: usar `continue`.

12. **Desafío** (obligatorio):

    a. Escribir una función que dada la cantidad de ejercicios de un examen, y el porcentaje de ejercicios bien resueltos necesario para aprobar dicho examen, revise un grupo de exámenes.

    Para ello, en cada paso debe preguntarle al usuario la cantidad de ejercicios resueltos por el alumno, o pedirle que ingrese "*" para salir. Debe mostrar por pantalla el porcentaje correspondiente a la cantidad de ejercicios resueltos respecto a la cantidad de ejercicios del examen y una leyenda que indique si aprobó o no.

    b. Adicional al punto anterior: imprimir un mensaje informándole al usuario la cantidad de ejercicios y el % de aprobación.
    Validar que el usuario siempre ingrese números positivos y menor o iguales a la cantidad de ejercicios del examen, o "*". De lo contrario, mostrar un mensaje de error y volver a pedirle el dato al usuario.

Guía 4: Tipos de Estructuras de Datos

1. Cadenas de caracteres

1. Escribir funciones que dada una cadena y un caracter:

a. Inserte el caracter entre cada letra de la cadena. Ejemplo: `'separar'` y `'-'` debería devolver `'s-e-p-a-r-a-r'`.

b. Reemplace todos los espacios por el caracter. Ejemplo: `'mi archivo de texto.txt'` y `'_'` debería devolver `'mi_archivo_de_texto.txt'`.

c. Reemplace todos los dígitos de la cadena por el caracter. Ejemplo: `'su clave es: 1540'` y `'*'` debería devolver `'su clave es: ****'`.

d. Inserte el caracter cada 3 dígitos en la cadena. Ejemplo: `'2552552550'` y `'.'` debería devolver `'255.255.255.0'`

e. Modificar todas las anteriores para que, adicionalmente, reciba un parámetro que indique la cantidad máxima de reemplazos o inserciones a realizar. Ejemplo: `'su clave es: 1540'`, `'*'` y `3` debería devolver `'su clave es: ***0'`.

2. Escribir una función que reciba una cadena que contiene un largo número entero y devuelva una cadena con el número y las separaciones de miles. Por ejemplo, si recibe `1234567890`, debe devolver `1.234.567.890`. Cuidado: no es lo mismo `123.456.789.0` que `1.234.567.890`. Tienen que ser separaciones de miles y quedar un número válido.

3. Escribir funciones que dada una cadena de caracteres:

   a. Devuelva la primera letra de cada palabra. Ejemplo: si se recibe `Ciclo Básico Común` se debe devolver `CBC`.
   b. Indique si se trata de un palíndromo. Por ejemplo, `anita lava la tina` es un palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda).

4. Escribir funciones que dadas dos cadenas de caracteres:

   a. Indique si la segunda cadena es subcadena de la primera. Por ejemplo, `'compu'` es subcadena de `'computacional'`.
   b. Devuelva la que sea anterior en orden alfábetico. Por ejemplo, si recibe `'kde'` y `'gnome'` debe devolver `'gnome'`.

5. Escribir una función que, dada una cadena de caracteres, devuelva una lista con cada uno de los caracteres que la componen en mayúscula. Ejemplo: `'Hola'` debe devolver `['H', 'O', 'L', 'A']`. Restricción: no se permite el uso de ciclos for/while. Pista: Buscá en el apunte cómo usar `map`.

6. Escribir una función que, dada una cadena de caracteres, devuelva una tupla con cada uno de los caracteres que no es una vocal. Ejemplo: `'Algoritmos'` debe devolver `('l', 'g', 'r', 't', 'm', 's')`. Restricción: no se permite el uso de ciclos for/while.

7. Escribir una función que, dada una cadena de caracteres, devuelva el número de índice de posición del último caracter. Por ejemplo, para la cadena `'Hola'` debe devolver `3`. Restricción: no se permite el uso de ciclos for/while.

8. **Desafío** (obligatorio):

    a. Se quiere implementar un buscador dentro de un editor de texto, que permita encontrar todas las ocurrencias de una palabra en un texto. Para ello, se debe implementar una función que reciba como parámetro una palabra y un texto, y que devuelva la primer aparición de la palabra en el texto. Pista: `index` arrojará un error si la subcadena no se encuentra. ¿Qué otro método tenemos disponible para buscar subcadenas?

    b. Modificar la función anterior para que devuelva una lista con las posiciones de inicio de cada ocurrencia de la palabra dentro del texto. Ejemplo: si se busca `'al'` en `'calcule el precio al valor actual'`, debe devolver `[1, 18, 22, 31]`. Pista: del método usado en el punto anterior, ¿conocemos algún parámetro adicional que le podamos pasar?

    c. Modificar la función anterior para que devuelva la cantidad de ocurrencias encontradas. Ejemplo: si se busca `'al'` en `'calcule el precio al valor actual'`, debe devolver `4`. Restricción: No se puede usar el método `len`.

9. **Desafío** (no obligatorio): Escribir una función que reciba dos cadenas de caracteres y devuelva una lista con todos los caracteres que no tienen en común. Ejemplo: `'Python'` y `'Hola'` debería devolver el conjunto de letras `['P', 'y', 't', 'l', 'a', 'n']`, indiferentemente del orden y de si está en mayúscula o minúscula. Nota: para que un caracter esté en la lista, no es necesario que esté en la misma posición.

2. Rangos, Tuplas y Listas

1. Usar un rango para:

    a. Imprimir los números del 10 al 50 inclusive, saltando de 5 en 5.
    b. Imprimir los números del 40 al 20 en orden decreciente, saltando de 2 en 2.
    c. Crear una lista con los números del 4 al 10. Luego, acceder con el *índice* a los elementos que contienen a los números 4, 6 y 9 e impimirlos por pantalla. Pista: recordar que los índices comienzan en 0.

2. Escribir una función que reciba:

    a. Una lista y devuelva `True` si su longitud es par y `False` si su longitud es impar.

    b. Una lista de números cualesquiera y devuelva el elemento máximo y el mínimo.

c. Una lista de números y devuelva otra lista con los mismos números ordenados de menor a mayor. Por ejemplo, si recibe `[5, 10, 7, 3]` debe devolver `[3, 5, 7, 10]`.

3. a. Escribir una función que reciba una lista de nombres y un número, que representa el cupo. La función debe devolver en una lista a los nombres que no pudieron entrar al curso por falta de cupo. Ejemplo: `chequear_cupo(['Agustina', 'Iara', 'Priscila', 'Sol', 'Lucía'], 3)` debe devolver `['Sol', 'Lucía']`.

   b. Modificar la función anterior para que devuelva únicamente a la última persona de la lista de la gente que pudo entrar. Ejemplo: `chequear_cupo(['Agustina', 'Iara', 'Priscila', 'Sol', 'Lucía'], 3)` debe devolver `'Priscila'`, porque es la última que tuvo cupo.

4. Dada la lista de tuplas `[("Argentina", 3), ("España",1), ("Uruguay", 2), ("Francia",2)]`, donde cada tupla contiene un país y la cantidad de mundiales que ganaron:

   a. Hacer una función que reciba la lista por parámetro e imprima la información de cada país con el siguiente formato:

   `País: <nombre> - Copas: <cantidad>`

   Si y sólo si el país es "Argentina", se debe imprimir el nombre con 3 estrellas: `"Argentina "`. Usar el operador abreviado `+=`.

   b. Hacer una función que reciba la lista por parámetro y devuelva la cantidad de mundiales que ganaron entre todos los países. Ejemplo: `contar_mundiales([("Argentina", 3), ("España",1), ("Uruguay", 2), ("Francia",2)])` debe devolver `8`.

   c. Hacer una función que reciba la lista por parámetro y la devuelva, ordenada por cantidad de copas ganadas.

   d. Hacer una función que reciba la lista por parámetro y devuelva en una tupla: una lista con los países que tienen más de una copa ganada, y otra lista con valores booleanos que nos diga si la cantidad de copas es par o impar. Pista: ¿Cómo podemos usar `filter`? ¿Y `map`?
   Ejemplo: `[("Argentina", 3), ("España",1), ("Uruguay", 2), ("Francia",2)]` devuelve:
   `([("Argentina", 3), ("Uruguay", 2), ("Francia",2)] , [False, True, True])`

5. Escribir una función que reciba dos fichas de dominó y determine si *encajan* o no entre sí.

   a. Resolver teniendo en cuenta que las fichas se reciben con formato de tuplas. Ejemplo: `(3,4)` y `(5,4)`.

b. Resolver teniendo en cuenta que las fichas se reciben con formato de string. Ejemplo: `'3-4'` y `'5-4'`.

6. Escribir una función que reciba dos vectores y devuelva su prod_escalar. El prod_escalar se calcula como: Siendo $v1 = (v1_1, v1_2, ..., v1_n)$ y $v2 = (v2_1, v2_2, ..., v2_n)$, entonces

$$v1 \cdot v2 = (v1_1 \cdot v2_1) + (v1_2 \cdot v2_2) + ... + (v1_n \cdot v2_n)$$

Si los vectores no tienen las mismas dimensiones, la función debe devolver `None`.

7. a. Escribir una función que reciba una tupla, un índice, y un nuevo valor. La función debe modificar la tupla, cambiando el valor en la posición dada por el índice, por el nuevo valor pasado como parámetro. Devolver la tupla modificada.

   b. Repetir el ejercicio anterior, pero con una lista.

   c. Repetir ambos si ahora, en vez de recibir un índice, se recibe el valor a eliminar. Si no se contiene al valor, se devuelve la estructura tal cual se recibió.

8. Escribir una función que reciba una lista y un número n. Para dicho número n, debe imprimir los últimos n elementos de la lista en orden inverso, y luego devolver la lista sin ellos. Ejemplo: Si se recibe `[1, 2, 3, 4, 5]` y `n = 2`, debe imprimir `5`, `4` y devolver `[1, 2, 3]`.

9. Escribir una función que reciba una lista de números y devuelva la misma lista en orden inverso.

10. Escribir una función que dado un valor n, devuelva una lista con los números del 1 a n.

11. Escribir una función que reciba una matriz y una tupla (fila, columna), y devuelva el valor ubicado en esa posición de la matriz. Ejemplo: si se recibe la matriz `[[1, 2], [3, 4]]` y la tupla `(0, 1)`, debe devolver `2`.

12. Se tiene una lista de supermercado escrita como string con productos separados por coma: `"pan, arroz, pescado, jugo, fideos,..."`.

    a. Escribir una función que reciba la cadena de caracteres de los productos de supermercado y devuelva una lista con cada uno de los productos por separado: `['pan', 'arroz', 'pescado', 'jugo', 'fideos', ...]`.

    b. Se tiene además otra cadena de caracteres con los precios de cada producto: `"100, 50, 200, 80, 30,..."`. Escribir una función que reciba ambas cadenas y devuelva una lista con tuplas de (producto, precio): `[('pan', 100), ('arroz', 50), ('pescado', 200), ('jugo', 80), ('fideos', 30), ...]`.

    c. Para la función del punto anterior, escribir otra función que reciba la lista de tuplas y devuelva el precio total de la lista de compras.

13. Se quiere crear una lista de supermercado, solicitándole al usuario productos hasta que ingrese el valor 'X'. La función debe devolver los productos en un string, separados por comas. Ejemplo: si se ingresa 'pan', 'arroz', 'pescado', 'X', debe devolver `"pan, arroz, pescado"`.

14. Hacer una función que reciba una lista de palabras, las ordene en orden alfabético y luego las una en un string separadas por espacios. Ejemplo: si recibe `['hola', 'como', 'estas']`, debe devolver `"como estas hola"`.

15. **Desafio** (obligatorio): Escribir una función que reciba un tamaño y devuelva una matriz con 1 en la diagonal principal y 0 en el resto. Ejemplo: si recibe 4, debe devolver la matriz identidad de tamaño 4x4.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

16. **Desafio** (obligatorio): Escribir una función que reciba una matriz y devuelva su transpuesta. Ejemplo: si recibe la matriz `[[1, 2, 3], [4, 5, 6]]`, debe devolver `[[1, 4], [2, 5], [3, 6]]`.

    Si se recibe:

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

    Se debe devolver:

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

17. **Desafio** (no obligatorio): Agenda Simplificada
    Escribir una función que reciba una cadena a buscar y una lista de tuplas `(nombre_completo, telefono)`, y busque dentro de la lista todas las entradas que contengan en el nombre completo la cadena recibida (puede ser el nombre, el apellido o sólo una parte de cualquiera de ellos). Debe devolver una lista con todas las tuplas encontradas.

18. **Desafio** (no obligatorio): Sistema de facturación simplificado.
    Se cuenta con una lista ordenada de productos con tuplas de (identificador, descripción, precio), y una lista de los productos a facturar, con tuplas de (identificador, cantidad).

    Se desea generar una factura que incluya la cantidad, la descripción, el precio unitario y el precio total de cada prod_comprado, y al final imprima el total general.

    Escribir una función que reciba ambas listas e imprima por pantalla la factura solicitada.

19. **Super Desafio** (no obligatorio): Batalla Naval

    Se tiene una matriz de 10x10 que representa un tablero. Cada celda contiene un 0 si está vacía, o un 1 si hay un barco (consideramos que en este caso, sólo hay barcos unitarios que ocupan un espacio).

    La posición de los barcos se representa con tuplas de (fila, columna). Por ejemplo, si se tiene un barco en la fila 1, columna 3, se representa con la tupla (1, 3).

    Escribir una función que cree un tablero con 10 barcos ubicados aleatoriamente (usar la biblioteca `random`), y que permita al usuario intentar adivinar dónde están.

    El usuario luego ingresa una posición, y la máquina indica si había un barco en esa posición (mostrando un mensaje por pantalla "¡Hundido!") o no ("¡Agua!").

    El usuario gana cuando hunde todos los barcos del tablero. Si se equivoca más de 5 veces, pierde.

    > ! Batalla Naval: Modo Supervivencia
    >
    > ¿Te animás a que el juego sea un ida y vuelta? Es decir, que el usuario también pueda poner barcos y la máquina intente adivinar dónde están. Una posibilidad es que el usuario tenga su propio tablero en un papel, y una vez cada uno, la máquina y el usuario elijan una posición para atacar.
> Te dejamos unos tips:
>
> - Las posiciones son limitadas por el tablero 10x10
> - Las posiciones no deberían repetirse
>
> ¿Se te ocurre una forma fácil de generar y guardar todas las posiciones posibles del tablero, e ir sacando de a una para que no se repitan? ¿Quién pensás que ganaría, la máquina o el usuario? En este caso, el usuario y la máquina tienen intentos ilimitados intercalados hasta que alguno de los dos gane.

20. Se tiene una base de datos con nombres de libros de la siguiente forma `["La Noche de la Usina", "La Pregunta de sus Ojos", "Ser Feliz era Esto",...]`, y se quiere saber cuántos libros repetidos tienen. Escribir una función que reciba la base de datos y devuelva, para cada uno de los títulos, cuántos ejemplares hay. La lista no tiene un tamaño fijo, y puede contener muchos títulos repetidos.
    Pista: tenés que usar un diccionario.

3. Diccionarios

1. Escribir una función que reciba una lista de tuplas, y que devuelva un diccionario en donde las claves sean los primeros elementos de las tuplas, y los valores una lista con los segundos. Por ejemplo:

```
l = [('Hola', 'don Pepito'), ('Hola', 'don Jose'), ('Buenos', 'días')]
print(tuplas_a_diccionario(l))
```

```
{'Hola': ['don Pepito', 'don Jose'], 'Buenos': ['días']}
```

2. Escriba una función que reciba una cadena y devuelva:

   a. Un diccionario con la cantidad de apariciones de cada palabra en la cadena. Por ejemplo, si recibe `"Que lindo dia que hace hoy"` debe devolver: `{'que': 2, 'lindo': 1, 'dia': 1, 'hace': 1, 'hoy': 1}`.
   b. Un diccionario con la cantidad de apariciones de cada caracter en la cadena.

3. Escribir una función que reciba una cantidad de iteraciones N.

   a. Se deberá simular una persona que tira un dado N veces, y se deberá devolver un diccionario con la cantidad de apariciones de cada valor en el dado. Nota: para simular una tirada, usar `import random`y `random.randint(1, 6)`.

   b. Repetir el punto anterior, si ahora en vez de tirar 1 dado, tira 2. Se debe devolver un diccionario con la cantidad de apariciones de cada valor de la suma de ambos dados.

4. Se tiene una agenda implementada como diccionario, que guarda nombres de personas y sus números de teléfono. Escribir un programa que le pida al usuario que ingrese nombres.

   a. Si el nombre se encuentra en la agenda, debe mostrar el teléfono.
   b. Si el nombre no se encuentra, debe permitir ingresar el teléfono correspondiente.

   En ambos casos, El usuario puede utilizar la palabra “EXIT” para dejar de ingresar nombres.

5. Escribir una función que reciba un texto y para cada caracter presente en el texto, devuelva la palabra más larga en la que se encuentra ese caracter.

6. Nos contratan para hacer un nuevo sistema de FIUBA para almacenar información de sus estudiantes:

nombre	apellido	dni	carrera
Violeta	Perez	42000000	Informática
Carla	Guanca	42001001	Mecánica
Manuela	Gomez	42002002	Química

a. Crear un diccionario que sirva para representar a cada persona. Debe contener las claves `nombre`, `apellido`, `dni` y `carrera`. Los diccionarios se deben guardan en una lista llamada `estudiantes`.

b. Agregar al diccionario creado un nuevo elemento, que debe ser otro diccionario y represente las notas obtenidas en la carrera. La clave debe ser el `codigo` y el valor la `nota` (del 1 al 10) obtenida.

c. Crear código que agregue para la estudiante Violeta Perez la nota 7 en la materia Algoritmos y Programación III (7507), y la nota 4 en la materia Análisis Matemático II (6103).

d. Teniendo la lista de estudiantes, buscar en la lista la persona con mayor cantidad de notas e imprimirla por pantalla.

7. En un vivero se guardan las plantas en una lista de diccionarios con la siguiente información: especie, luz directa (si/no), precio. Se necesita un sistema que guarde las plantas a medida que van llegando. Hacer una función que reciba la lista de diccionarios de plantas, y los datos de la planta nueva, y agregue esa planta a la lista de diccionarios.

8. Escribir una función que reciba una lista de diccionarios y una clave, y devuelva una lista con los valores correspondientes a esa clave.

9. Se tiene un ticket de supermercado en forma de diccionario con los siguientes datos:

   - Nombre del Producto
   - Precio por Unidad
   - Cantidad

   Se pide hacer una función que reciba el ticket y devuelva el monto a pagar total.

10. Rosita tiene una lista de diccionarios donde guarda todas las películas que vió. La información para cada una es: el nombre de la película, año en que salió, y la puntuación que le puso del 1 al 10. Hacer una función que reciba el diccionario y devuelva una nueva lista de diccionarios donde sólo estén las películas que tienen puntaje mayor a 7.

    a. Resolver sin usar `filter`
    b. Resolver usando `filter`.

11. La profesora Llamell guarda las notas del parcial de Pensamiento Computacional en una lista de diccionarios. Cada diccionario tiene la siguiente información: nombre, apellido, intento, nota.

    Los intentos pueden ser 1 (si es la primera vez que rinde el parcial) o 2 (si está en el recuperatorio).

    a. Se pide hacer una función que dada esta lista de diccionarios, se devuelva el promedio de las notas en la primera oportunidad de los alumnos.

    b. Generalizar la función anterior, para que también reciba el número de intento y se pueda devolver el promedio de cualquiera de los dos intentos.

12. En una fábrica se tiene una base de datos donde se guardan todos los códigos de los productos que se fabrican como claves de un diccionario. Los valores de cada clave son nuevos diccionarios, con la siguiente información: fecha de vencimiento (mes,año), si pasó el chequeo de calidad o no.

    Se pude hacer una función que reciba esta lista de diccinoarios, y elimine a todos los productos que no pasaron el chequeo de calidad. Devolver en una tuple todos los productos eliminados en formato {codigo: diccionario del producto}.

13. Se quiere guardar información de un grupo de maratonistas. Se necesita guardar su nombre, DNI y todas las maratones que corrió. Para esto último, se guardan: nombre de cada una, año, puesto y el tiempo que tardaron en correrlas (en minutos).

    a. Crear un diccionario de ejemplo que represente esta situación.
    b. Teniendo esta lista de diccionarios, ordenarlos alfabéticamente por el nombre de los maratonistas.
    c. Teniendo esta lista de diccionarios, ordenar las maratones en tiempo ascendente según el tiempo que tardaron en correrlas.

14. **Desafío** (obligatorio): Laura tiene una lista de diccionarios donde guarda el valor de todas las reviews laborales anuales que le hicieron. La información de cada una es año, seniority en ese momento (trainee, junior, semisenior, senior), el sueldo en ese momento y el valor del bono de performance que le dieron. La semana pasada le avisaron que por políticas de la empresa, los bonos ahora deben calcularse como un porcentaje de su sueldo.

    Laura quiere entonces actualizar sus diccionarios, para que en vez de guardar el monto exacto del bono, guarde el porcentaje que le corresponde. Ejemplo: si en el 2019 su sueldo era de $1.000.000 y el bono que le dieron era de $40.000, el bono fue del 4% del sueldo.

    a. Hacer una función que reciba la lista de diccionarios, y para cada una de las reviews, modifique el valor del bono por el porcentaje correspondiente.

b. Hacer una función que reciba la lista de diccionarios ya modificada y devuelva los años en los que Laura tuvo un bono mayor al 50% de su sueldo. Restricción: usar `filter` y `map`.

15. **Desafío** (obligatorio): Los estudiantes de la materia Pensamiento Computacional quieren crear un programa que les facilite la cursada. Uno de los problemas es que no se tiene fácil acceso a los enunciados de los ejercicios, porque la guía es larga y hay que scrollear mucho. En el programa, la guía de ejercicios se guarda en un diccionario, donde cada clave es el número de guía y cada valor una lista con los enunciados de los ejercicios, cada uno en su posición correspondiente (la posición 0 de la lista sólo guarda None). Se quiere que el usuario que está usando el programa pueda acceder por pantalla a un enunciado puntual de una guía con sólo pedirlo (si existe). Para el problema mencionado, hacer una función o más que lo resuelva. Usar los temas visto en la materia hasta el momento, en la forma que se considere mejor y siguiendo las buenas prácticas y convenciones enseñadas.

16. **Desafío** (no obligatorio): **Donarg** (https://www.donarg.com.ar/) es un proyecto que nació con estudiantes de FIUBA con el fin de optimizar procesos tanto para donantes de sangre como para hospitales y servicios de hemoterapia. Formado por estudiantes y graduados universitarios comprometidos, fue galardonado con el primer puesto en la FIUBATON 2020 "Desafío Cuarentena" del FIUBA Consulting Club, destacándose entre más de 100 proyectos.

    Donarg necesita un sistema que permita filtrar una base de datos de posibles donantes de sangre, quedándose con los que cumplen los requisitos.

    La base contiene los siguientes datos de cada posible donante:

    - Nombre
    - Apellido
    - Edad
    - Peso
    - Fecha de la última donación. Puede ser 'None' si nunca donó. Formato: (dia,mes,año)
    - Fecha del último tatuaje. Puede ser 'None' si no tiene tatuajes. Formato: (dia,mes,año)
    - Tipo de sangre. Puede ser '0+', '0-', 'A+', 'A-', 'B+', 'B-', 'AB+', 'AB-'

    Los requisitos son:

    - Tener entre 16 y 65 años
    - Pesar más de 50 kilos
    - Que hayan pasado 2 meses desde la última donación
    - Que hayan pasado 6 meses desde el último tatuaje

a. Se pide hacer una función que reciba una lista de diccionarios con la información de cada posible donante, y devuelva una lista con los que cumplen los requisitos.

b. Se pide hacer una función que priorice a los donantes que tienen sangre tipo 0 (positivo y negativo) por sobre todos los A, B y AB (positivos y negativos); ya que son los que más se necesitan. La función debe recibir la lista de diccionarios con la información de cada posible donante ya filtrada por requisitos, y devolver una nueva lista ordenados de mayor a menor prioridad.

c. Se pide hacer una función que reciba la lista de diccionarios con la información de cada posible donante ya filtrada por requisitos y ordenada por prioridad, que se quede con los que son 0+ y 0-, y los ordene por órden alfabético de apellido.

Si querés saber más sobre el proyecto, podés visitar su página web:
https://www.donarg.com.ar/
o sacar turno para donar sangre en
https://www.donarg.com.ar/dondedono.

Guía 5: Entrada y Salida

1. Archivos

1. Escribir una función llamada `contar` (o `count`) que dado un archivo retorne la cantidad de filas que tiene.
2. Escribir una funcion llamada `imprimir`(o `cat`) que dado un archivo imprima por pantalla todo el contenido.

    a. Agregar un parametro a la funcion llamado `ignorar_vacias` (o `blank`) que en caso de ser `True`, no se impriman las lineas en blanco.

3. Escribir una función llamada `ver_encabezado` (o `head`) que dado un archivo y un número N retorne en una lista las primeras N lineas del archivo.
4. Escribir una función llamada `ver_final` (o `tail`) que dado un archivo y un número N retorne en una lista las últimas N lineas del archivo.
5. Escribir una función llamada `crear` (o `touch`) que dado el nombre de un archivo lo cree. Si el archivo exixte borra todo el contenido.
6. Escribir una función, llamada `wc`, que dado un archivo de texto, lo procese e imprima por pantalla cuántas líneas, cuantas palabras y cuántos caracteres contiene el archivo.
7. Escribir una función, llamada grep, que reciba una cadena y un archivo de texto, e imprima las líneas del archivo que contienen la cadena recibida.

8. Se tiene un archivo con una pregunta, llamado `pregunta.txt`. Se pide leerlo y mostrar el contenido en consola. Luego, pedirle al usuario una respuesta, y guardarla en un archivo llamado `respuesta.txt`.

9. Sea un diccionario cuyas claves y valores son cadenas.

   a. Escribir una función `guardar_diccionario` que reciba el diccionario y un nombre de archivo, y guarde el contenido del diccionario en el archivo, en formato CSV, con un par clave-valor por línea.
   b. Escribir una función `cargar_diccionario` que reciba el nombre de un archivo con el formato mencionado en el punto anterior, y devuelva el diccionario original.

10. **Desafio** (obligatorio): Distribución de Carga
    Se tiene una base de datos en formato CSV con resultados de partidos de futbol. Como el archivo es muy grande, se quiere distribuir la base en varios archivos más pequeños. El formato del archivo es el siguiente:

    ```
    local;visita;goles_local;goles_visita;gano;penales
    real madrid;boca juniors;1;2;visita;N
    A.C. Milan; boca juniors;1;1;visita;S
    .
    .
    .
    ```

    Utilizando las funciones de los ejercicios 1 a 7, procesar el archivo: Se desea que el archivo original se separe en archivos resultantes de no mas de 10 lineas, además no tienen que quedar filas en blanco. Por ejemplo si el archivo original tiene 100 lineas de las cuales 20 están en blanco, el resultado del procesamiento tienen que ser 8 archivos de 10 lineas cada uno. El tamaño del archivo original puede variar en cada ejecución.

11. La señora Rowling sospecha que su vecino le robo algunos manuscritos y los publicó como propios con algunas modificaciones. Se desea procesar una lista de archivos y guardar en un archivo llamado `reporte_plagio.txt` la siguiente información:

    ```
    archivo;palabras;apariciones
    archivo1.txt;765030;547
    ```

    - archivo: es el nombre del archivo que se proceso.
    - palabras: es la cantidad de palabras de ese archivo.
    - apariciones: es la cantidad de veces que aparece una palabra especificada.

    En ejemplo de uso podría ser:

    ```
    archivos = ["archivo1.txt", "archivo2.txt"]
    procesar_archivos(archivos, "harry")
    ```

En `procesar_archivos` se tiene que analizar los archivos y dejar el resultado en reporte_plagio.txt

12. Hacer una función que reciba un archivo y dos palabras: una que se quiere reemplazar, y otra por la que se quiere reemplazar. La función debe modificar el archivo reemplazando todas las apariciones de la primera palabra por la segunda.

13. Se tiene un archivo CSV que contiene información sobre el stock de una librería.

```
Un posible ejemplo de este archivo es el siguiente:
    lapiceras;34512;50;120
    cuadernos;41611;500;130
    sacapuntas;62812;30;210
```

Donde cada línea representa un prod_y contiene el nombre del producto, el código de barras, la cantidad en stock y el precio unitario. Hacer una función que le solicite al usuario datos de un nuevo prod_y lo agregue al final del archivo. Debe seguir pidiéndole al usuario datos de productos hasta que este ingrese como prod_"X".

14. En un cine tienen dos archivos .txt, uno con salas y otro con nombres de películas. Se sabe que en la sala de una fila del archivo se va a transmitir la película de la misma fila del archivo de películas. Se pide leer los dos archivos, y crear un nuevo archivo csv que tenga el nuevo formato sala;pelicula Por ejemplo si se tienen los siguientes archivos:

(salas.txt)

```
D2
F1
E4
```

(peliculas.txt)

```
Megamente
The Menu
Shrek
```

El nuevo archivo deberá quedar:
(funciones.csv)

```
D2;Megamente
F1;The Menu
E4;Shrek
```

15. Se tiene un archivo con información de ventas de productos en un supermercado. El archivo tiene el siguiente formato:

```
producto;precio;cantidad;fecha
arroz;50;100;2021-01-01
fideos;40;200;2021-01-01
arroz;50;100;2021-01-02
fideos;40;200;2021-01-02
arroz;50;100;2021-01-03
fideos;40;200;2021-01-03
```

Se pide hacer una función que reciba el nombre del archivo y un producto, y devuelva el precio promedio de ese prod_en el archivo.

16. Se tiene una lista de archivos y se quiere imprimir las primeras 5 líneas de cada uno. Si el archivo no existe, se lo debe crear e imprimir el mensaje: 'el archivo {nombre} no existe. Se crea y deja vacío.'.

17. **Desafío** (obligatorio): Los docentes de Pensamiento Computacional saben que los ejercicios de las primeras guías son cortitos de resolver, entonces les dicen a los estudiantes que "si el ejercicio tiene más de 5 líneas, probablemente haya que revisarlo". Los estudiantes se lo toman muy literal, y quieren buscar la forma de chequear si un ejercicio que resolvieron tiene más de 5 líneas, pero sin contar a la firma de la función.
El ejercicio puede estar guardado en un archivo o en un string todo junto, lo que resulte más cómodo para su resolución. Además, quieren la posibilidad de poder pasar varios ejercicios juntos para ahorrar tiempo, y saber en cuáles se pasaron de las 5 líneas o si algun archivo (en caso de elegir esa forma de almacenamiento) no existe. Para el problema mencionado, hacer una función o más que lo resuelva. Usar los temas visto en la materia hasta el momento, en la forma que se considere mejor y siguiendo las buenas prácticas y convenciones enseñadas.

18. **Desafío** (obligatorio): Los estudiantes del CBC pensaron que podía ser buena idea guardar los apuntes de todas las materias en un mismo archivo, para no marearse con tantos documentos diferentes. Pero terminó siendo un dolor de cabeza y es muy dificil encontrar las notas de cada materia. Para poder acceder facil a esa informacion, decidieron extraer del archivo original la informacion de las notas. Sabemos que a cada materia se le puso de título un hashtag, por ejemplo "#analisisMatematico", y que la fila que contiene un título no contiene nada más. Lo que se quiere hacer es obtener un nuevo archivo CSV con el nombre de la sección/materia, la posición (número de línea) donde comienza la misma, y la posición donde termina. Para el problema mencionado, hacer una función o más que lo resuelva. Usar los temas visto en la materia hasta el momento, en la forma que se considere mejor y siguiendo las buenas prácticas y convenciones enseñadas.

2. Manejo de Errores

1. Crear una función que asegure la apertura de un archivo. Debe recibir un nombre y si el archivo existe, retorna el contenido archivo. Si no existe, imprime un mensaje descriptivo.

2. Escribir una función que le pida al usuario que ingrese 5 números y le muestre la suma total de todos ellos. Si el valor ingresado no es un número, debe mostrar un mensaje de error y seguir pidiendo números.

3. Escribir un programa que le pida números al usuario e imprima si son pares o impares. Si el usuario ingresa un valor que no es un número, el programa debe imprimir un mensaje de error y seguir pidiendo números. El programa termina cuando el usuario ingresa `"*"`.

4. Un climatólogo tiene archivos donde guarda las temperaturas promedio de la Ciudad de Buenos Aires de cada mes. Cada línea es un número que representa la temperatura promedio del día. Por ejemplo:

   ```
   25.5
   26.0
   24.5
   # etc
   ```

   Como el climatólogo es humano, a veces se equivoca y tipea cosas que no son números. Por ejemplo: `25p3` o `25.5.5`. Se pide hacer una función que reciba el nombre del archivo y devuelva la temperatura promedio del archivo. Si en algún momento se encuentra con un valor que no es un número, debe ignorarlo y seguir con el resto de los valores.

   Al final de la ejecución, debe devolverse el valor promedio y el porcentaje de valores que no pudieron ser procesados respecto del total.

5. Crea un programa que pueda procesar un archivo: por cada fila, se debe ejecutar la operacion de división `dividendo/divisor` y almacenar el resultado en un archivo llamado "resultados.csv". Tener en cuenta que en el archivo se pueden encontar filas con errores de carga y que el divisor sea 0 o que no sean números, en tal caso en lugar de escribir el resultado escribir "Error en la fila X: {descripción específica del error}" donde X es la fila que tiene errores.

   ```
   dividendo;divisor
   83848;389
   8762;78
   .
   .
   .
   ```

6. **Desafio**: Para la venta de entradas de un teatro se cuenta con un diccionario que contiene las filas: "A", "B", "C"… y en cada fila contiene una lista con las ubicaciones disponibles en forma de lista `["L","O","O","O","L","L","L"]` donde "L" es libre y "O" es ocupada.

   Escribir una funcion que reciba el diccionario, la fila y la ubicación en ella, y reserve el asiento en caso de que este libre. Considerar que el tamaño de la lista de ubicaciones puede variar por fila. Si la fila o la ubicación no son correctas, mostrar un mensaje descriptivo. Si el asiento ya se encuentra ocupado, mostrar el mensaje: `El asiento ya se encuentra ocupado`.

   Ejemplo

```
sala = {"A":["L","O","O","O","L","L","L"], "B": ["L","O","O","O","L"]}
reservar_butaca(sala, "A", 0)
# En este ejemplo la fila "A" deberia quedar ["O","O","O","O","L","L","L"]
```

7. **Desafio** (obligatorio): Se tiene esta función, que recibe dos números y devuelve la división entre ellos:

```
def dividir(n1, n2):
    if n2 == 0:
        raise ZeroDivisionError("No se puede dividir por cero")
    return n1/n2
```

   La función puede lanzar una excepción `ZeroDivisionError` si el segundo número es 0 (es decir, la ejecución va a terminar con un error).

   a. Probar usar la función de arriba, pasándole 5 y 0 como argumentos. ¿Qué pasa? ¿Cómo podrías evitar que el programa termine con un error?

   b. Se pide hacer un programa que le pida al usuario dos números, y muestre el resultado de la división usando la función `dividir` definida arriba. Si el usuario ingresa un 0 como segundo número, debe mostrar un mensaje de error y retornar `None`.

   c. Considerar ahora también que el usuario podría ingresar algo que no es un número. En ese caso, debe mostrar un mensaje de error y seguir pidiendo números.

8. **Desafío** (no obligatorio): El kiosko de la facultad quiere automatizar un letrero que tome datos de un programa y le cobre al estudiante.

   Se tienen dos diccionarios, uno con un código y el producto, y otro con el código y el precio de cada producto.

```
opciones = {
    1: "hamburguesas",
    2: "milanesas",
    3: "gaseosa",
```

```
    4: "alfajor",
    5: "papas fritas",
    6: "agua"
}

valores = {
    1:1000,
    2:1500,
    3:500,
    4:300,
    5:600,
    6:350
}
```

Se quiere hacer un programa que muestre la información de la siguiente forma en la pantalla:

```
1. hamburguesas - $1000
2. milanesas - $1500
3. gaseosa - $500
4. alfajor - $300
5. papas fritas - $600
6. agua - $350
```

Y le pida al usuario una opción y una cantidad. Luego, debe imprimir el total a pagar.

Se debe considerar que el usuario podría ingresar una opción que no está en el diccionario, o ingresar una opción que no sea un número. El programa debe en esos casos imprimir un mensaje de error que sea descriptivo y terminar su ejecución.

Guía 6: Bibliotecas de Python

1. Numpy

1. Crear un array de 20 números espaciados uniformemente entre 0 y 10. Luego, imprimir:

   a. La dimensión
   b. La forma
   c. El tamaño

2. Crear un array llamado "arr" con números enteros del 0 al 9. Luego:

   a. Cambiar su forma de manera tal que tenga 2 filas y 5 columnas
   b. Insertar el valor 100 en la posición 3.

3. Para el array `x = np.arange(10)`, calcular:

   a. `y = x^2`
   b. `y = 3x + 2`
   c. `y = 1/(x+1)`
   d. `y = log10(x)`

4. Usando el array `y = 1/(x+1)` del ejercicio anterior:

   a. Calcular la media de `y`.
   b. Calcular la diferencia entre `y` y su media

5. Repetir el ejercicio 4.2.2 de la Guía 4:

   Escribir una función que reciba:

   a. Una lista y devuelva True si su longitud es par y False si su longitud es impar.

   b. Una lista de números cualesquiera y devuelva el elemento máximo y el mínimo.

c. Una lista de números y devuelva otra lista con los mismos números ordenados de menor a mayor. Por ejemplo, si recibe [5, 10, 7, 3] debe devolver [3, 5, 7, 10].

6. Repetir el ejercicio 4.2.6. de la Guía 4:

   Escribir una función que reciba dos vectores y devuelva su prod_escalar. El prod_escalar se calcula como: Siendo $v1 = (v1_1, v1_2, ..., v1_n)$ y $v2 = (v2_1, v2_2, ..., v2_n)$, entonces

   $$v1 \cdot v2 = (v1_1 \cdot v2_1) + (v1_2 \cdot v2_2) + ... + (v1_n \cdot v2_n)$$

   Si los vectores no tienen las mismas dimensiones, la función debe devolver `None`.

7. Repetir el ejercicio 4.2.11 de la Guía 4:

   Escribir una función que reciba una matriz y una tupla (fila, columna), y devuelva el valor ubicado en esa posición de la matriz. Ejemplo: si se recibe la matriz [[1, 2], [3, 4]] y la tupla (0, 1), debe devolver 2.

8. Repetir el ejercicio 4.2.15 de la Guía 4:

   Desafio (obligatorio): Escribir una función que reciba un tamaño y devuelva una matriz con 1 en la diagonal principal y 0 en el resto. Ejemplo: si recibe 4, debe devolver la matriz identidad de tamaño 4x4.

   $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

9. Repetir el ejercicio 4.2.16 de la Guía 4:

   Desafio (obligatorio): Escribir una función que reciba una matriz y devuelva su transpuesta. Ejemplo: si recibe la matriz `[[1, 2, 3], [4, 5, 6]]`, debe devolver `[[1, 4], [2, 5], [3, 6]]`. Si se recibe:

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

   Se debe devolver:

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

10. Una persona quiere cercar el frente de su casa con alambre. Los postes se deben poner a 1 metro de distancia como mínimo entre sí. Hacer una función que reciba un número entero que sea la cantidad de metros a cubrir, y devuelva un array con la posición de los postes necesarios. Por ejemplo, si se reciben `5` metros, debe devolver: `[0.  1.25  2.5  3.75  5.]`

11. Se va a realizar un concierto gratis en el Jardín Japonés, para lo cual se habilitó la reserva de asientos por internet. Por un error, las reservas se hicieron en una lista de 50 asientos, guardando el número de reserva si está ocupado o 0 sino: `[1,2,3,0,7,6,4,0,0,8,9,...]`.
    Los asientos en el Jardín Japonés se van a organizar en forma de matriz, teniendo 5 filas de 10 asientos cada uno. Se quiere transformar la lista de asientos en una matriz. Hacer una función que reciba la lista de asientos y devuelva la matriz de asientos organizada por filas.

    Ejemplo: Si se recibe `[1,2,3,0,7,6,4,0,0,8,9,10,0,13,11,12]`, se debe devolver: `[[1,2,3,0,7,6,4,0,0,8],[9,10,0,13,11,12]]`.

12. **Desafio** (no obligatorio): Se tiene una matriz de 10x10 que representa un tablero de Batalla Naval. Cada celda contiene un 0 si está vacía, o un 1 si no. Se pide hacer una función que cree un tablero con 10 barcos ubicados aleatoriamente (usar la biblioteca `random`).

    b. Luego, se debe hacer una función que reciba la matriz y permita al usuario intentar adivinar dónde están.
    Pista: se puede extender/modificar el **Super Desafio** (no obligatorio) de Batalla Naval de la Guía 4.

2. Pandas

> **Note**
>
> Para estos ejercicios, recomendamos usar Google Colab (ver apunte) y tener una celda por ejercicio.

Nos contratan para hacer un nuevo sistema de FIUBA para almacenar información de sus estudiantes.

Usando el siguiente DataFrame:

```python
import pandas as pd
data = {
    'nombre': ['Violeta', 'Carla', 'Manuela', 'Lucia', 'Emilia', 'Mariana', 'Aldo'],
    'apellido': ['Perez', 'Guanca', 'Gomez', 'Capon', 'Duzac', 'Szischik', 'Rastrelli'],
    'dni': [42000000, 42001001, 42002002, 37010020, 40001002, 38090080, 38111222],
    'año_nac': [1997, 1998, 1998, 1993, 2003, 1993, 1994],
    'mail': ['vp@fi.uba.ar', 'cg@fi.uba.ar', None, None, None, 'ms@fi.uba.ar', 'ar@fi.uba.ar
    'carrera': ['Informática', 'Mecánica', 'Química', 'Informática', 'Informática', 'Electró
}

df = pd.DataFrame(data)
df
```

	nombre	apellido	dni	año_nac	mail	carrera
0	Violeta	Perez	42000000	1997	vp@fi.uba.ar	Informática
1	Carla	Guanca	42001001	1998	cg@fi.uba.ar	Mecánica
2	Manuela	Gomez	42002002	1998	None	Química
3	Lucia	Capon	37010020	1993	None	Informática
4	Emilia	Duzac	40001002	2003	None	Informática
5	Mariana	Szischik	38090080	1993	ms@fi.uba.ar	Electrónica
6	Aldo	Rastrelli	38111222	1994	ar@fi.uba.ar	Informática

1. Mostrar el resumen de la información del DataFrame
2. Mostrar su forma
3. Mostrar la lista de los nombres de las columnas
4. Mostrar las primeras 3 filas
5. Mostrar las últimas 3 filas
6. Agregar la nueva columna al DataFrame con la información de la edad de los estudiantes para el año actual y mostrar el DataFrame resultante
7. Mostrar a los estudiantes que tienen edad mayor a 25.
8. Para los estudiantes que tienen edad mayor a 25, mostrar el promedio de edad.
9. Mostrar a la persona que es más joven.
10. Mostrar a la persona que se encuentra en la mitad de la tabla.
11. Mostrar sólo las columnas: “carrera” y “edad”.
12. Mostrar sólo los estudiantes que estudian Informática.

13. Mostrar sólo los estudiantes que estudian informática y tienen menos de 25 años.
14. Reemplazar `Informática` por `Informática o Sistemas` en la columna "carrera".
15. Mostrar la cantidad de estudiantes por carrera.
16. Renombrar la columna "carrera" por "ingeniería".
17. Para la primer estudiante, cambiar su carrera a "Electrónica".
18. Agregar una nueva fila al DataFrame con tus datos.
19. Agregar una nueva columna 'tiene_mail', con un valor booleano que indique si la estudiante tiene mail en el sistema.
20. Agregar una nueva columna, con un valor booleano que indique si a la estudiante se le necesita pedir un mail (sólo se le tiene que pedir un mail si no tiene mail asignado). Asumir que todavía no se tiene la columna 'tiene_mail'.
21. Ordenar el dataFrame por carrera y apellido, en ese orden.
22. Ordenar el dataFrame por carrera y edad. Carrera por forma ascendente, edad por forma descendente.
23. La profesora Llamell guarda las notas del parcial de Pensamiento Computacional en un dataFrame. Cada fila tiene la siguiente información: nombre, apellido, intento, nota. Los intentos pueden ser 1 (si es la primera vez que rinde el parcial) o 2 (si está en el recuperatorio).

    a. Obtener el promedio de las notas en la primera oportunidad de los alumnos.
    Nota: Este ejercicio ya fue resuelto en la Guía 4 de Diccionarios, es el 4.3.11.

```
data = {
    'nombre': ['Violeta', 'Carla', 'Manuela'],
    'apellido': ['Perez', 'Guanca', 'Gomez'],
    'dni': [42000000, 42001001, 42002002],
    'carrera': ['Informática', 'Mecánica', 'Química'],
    'nota': [7, 4, 6],
    'intento': [1, 2, 1]
}

df_notas = pd.DataFrame(data)
df_notas
```

	nombre	apellido	dni	carrera	nota	intento
0	Violeta	Perez	42000000	Informática	7	1
1	Carla	Guanca	42001001	Mecánica	4	2
2	Manuela	Gomez	42002002	Química	6	1

24. Rosita tiene un dataFrame donde guarda todas las películas que vió. La información para cada una es: el nombre de la película, año en que salió, y la puntuación que le puso del 1 al 10. Obtener sólo las películas que tienen puntaje mayor a 7. Nota: este ejercicio ya fue resuelto en la Guía 4 de Diccionarios, es el 4.3.10.

```
data = {
    'nombre': ['Harry Potter', 'El Señor de los Anillos', 'Barbie', 'Rapido y Furioso 18'],
    'año': [2001, 2003, 1972, 2040],
    'puntuacion': [8, 9, 8, 3]
}

df_pelis = pd.DataFrame(data)
df_pelis
```

3. Matplotlib

1. Graficar las funciones obtenidas en el ejercicio 3 de Numpy, usando los números del 1 al 100 del eje x. Hacerlo primero en distintos gráficos y luego las 4 en el mismo.

    a. y = x^2
    b. y = 3x + 2
    c. y = 1/(x+1)
    d. y = log10(x)
    e. ¿Hay alguna función que no se esté viendo correctamente en el gráfico? Tratar de entender por qué y graficarla en otro gráfico. Pista: mirar los valores que toma la función en el eje `y`.

2. Usando Numpy: Siendo `x` un conjunto de 1000 valores de 0 a 2 * pi separados de forma uniforme:

    a. Graficar, con grilla, título y referencias apropiadas, la función `seno de x` (`sin(x)`).
    b. Graficar, con grilla, título y referencias apropiadas, la función `coseno de x` (`cos(x)`).
    c. Graficar, con grilla, título y referencias apropiadas, la función `(seno de x) + x/2`.

3. Se tienen los siguientes datos de ventas de un prod_en los últimos 10 años:

```
import matplotlib.pyplot as plt
import numpy as np

años = np.arange(2012, 2022)

ventas = [140, 60, 120, 250, 200, 180, 100, 300, 120, 160]
ganancias = [14000, 45000, 20400, 100000, 50000, 25000, 100000, 30000, 15000, 35000]
productos = ["prod_1", "prod_2", "prod_3", "prod_4", "prod_5", "prod_6", "prod_7", "proc
cant_ventas_productos_2024 = [180, 160, 190, 140, 100, 200, 120, 130, 150, 170]
```

   En gráficos separados:

   a. Graficar las ventas en función de los años usando gráfico de línea y gráfico de barras horizontal. ¿Cuál te parece que es mejor para este caso?
   b. Graficar las ganancias en función de los años usando gráfico de línea y gráfico de barras. ¿Cuál te parece que es el mejor gráfico para este caso?
   c. Graficar la cantidad de ventas en el año 2024 en función de los productos usando gráfico de barras y gráfico de torta. ¿Cuál te parece que es el mejor gráfico para este caso?

4. Usando el siguiente dataframe:

```
data = {
    'Jurisdicción': [
        'Ciudad Autónoma de Buenos Aires', 'Buenos Aires', 'Catamarca', 'Chaco', 'Chubut
        'Córdoba', 'Corrientes', 'Entre Ríos', 'Formosa', 'Jujuy', 'La Pampa', 'La Rioja
        'Mendoza', 'Misiones', 'Neuquén', 'Río Negro', 'Salta', 'San Juan', 'San Luis',
        'Santa Cruz', 'Santa Fe', 'Santiago del Estero', 'Tierra del Fuego, Antártida e
        'Tucumán'
    ],
    'Capital': [
        '', 'La Plata', 'San Fernando del Valle de Catamarca', 'Resistencia', 'Rawson',
        'Córdoba', 'Corrientes', 'Paraná', 'Formosa', 'San Salvador de Jujuy', 'Santa Ro
        'Mendoza', 'Posadas', 'Neuquén', 'Viedma', 'Salta', 'San Juan', 'San Luis',
        'Río Gallegos', 'Santa Fe', 'Santiago del Estero', 'Ushuaia', 'San Miguel de Tuc
    ],
    'Población (hab)': [
        3075646, 17541141, 415438, 1204541, 618994, 3760450, 1120801, 1385961,
        605193, 770881, 358428, 393531, 1990338, 1261294, 664057, 747610,
        1424397, 781217, 508328, 365698, 3536418, 978313, 173715, 1694656
    ],
    'Superficie (km2)': [
```

```
        205.9, 305907.4, 101486.1, 99763.3, 224302.3, 164707.8, 89123.3, 78383.7,
        75488.3, 53244.2, 143492.5, 91493.7, 149069.2, 29911.4, 94422, 202168.6,
        155340.5, 88296.2, 75347.1, 244457.5, 133249.1, 136934.3, 910324.4, 22592.1
    ],
    'PBI': [
        154863803.5, 292689868, 6150949.159, 9832642.672, 17747854.21, 69363739.19,
        7968012.982, 20743409.1, 3807057.419, 6484938.334, 6990262.458, 5590515.628,
        33431369.11, 9646825.835, 22564106.16, 10264584.42, 13438834.91, 8262308.568,
        11780849.36, 11663738.04, 81588690.27, 8387858.731, 7049276.383, 13856198.9
    ],
}

df = pd.DataFrame(data)
df.head()
```

a. Realizar un gráfico de barras horizontales que muestre la cantidad de habitantes por jurisdicción. Las provincias con mayor población, deben ubicarse en la parte superior.
b. Realizar un gráfico de torta que muestre el porcentaje de superficie de cada región del país.
c. Realizar un gráfico de barras verticales que muestre el PBI de cada jurisdicción. Las provincias con mayor PBI, deben ubicarse a la derecha.
d. Realizar un gráfico de puntos que muestre la relación entre la población y la superficie de cada jurisdicción. ¿Qué conclusión se puede tomar respecto al gráfico obtenido? ¿Hay alguna excepción? De ser así, analice cuál, incluyendo también un análisis de los gráficos anteriores.

datos.gob.ar

¿Sabías que en Argentina hay un portal de datos abiertos?
Podés ingresar a datos.gob.ar y obtener información de diferentes áreas, como salud, educación, justicia, entre otras. ¡Es una excelente fuente de datos para hacer análisis!
Para usarlos, podés guardar el archivo csv en google drive e importarlo de la siguiente forma como un dataframe:

```
import pandas as pd

# Acá va el link público de tu archivo de google drive + '/export?format=csv'
url = "https://drive.google.com/drive/folders/ABC123XYZ456/export?format=csv"
df = pd.read_csv(url)
```

5. Para calcular la energía potencial elástica (Ep) almacenada en un material elástico, se

utiliza la siguiente fórmula:

$$Ep = \frac{1}{2} * k * x^2$$

donde k es la constante elástica (con unidad N/m) y x es la distancia que el resorte se estira o comprime (metros). La energía Ep resultante tiene como unidad el Joule (J).

Para un resorte con valor $*k = 0.15N/m*$, graficar en línea continua la energía potencial elástica $*Ep*$ en un rango de distancia x de 0 a 10m con pasos de 0.5. Incluir título, grilla, nombres de ejes y label (leyenda) con el valor de k.

6. Para calcular la posición final de un móvil dada una posición inicial, velocidad inicial, tiempo y aceleración, se tiene la siguiente fórmula:

$$x = x_0 + v_0.t + \frac{1}{2}.a.t^2$$

Donde x_0 es la posición inicial, v_0 la velocidad inicial, t el tiempo y a la aceleración.

Se tiene la siguiente información de dos móviles:

movil 1:

- Comienza a 0m
- Arranca desde el reposo con velocidad $ 0 m/s$
- Su aceleración es 1 m/s^2

movil 2:

- Comienza a 500m
- Arranca con velocidad 4 m/s
- Su aceleración es 0.5 m/s^2

Para 5000 valores del tiempo del 0 al 100, mostrar un único gráfico con la posición respecto del tiempo de ambos móviles. Incluir grilla, títulos y label (leyenda) indicando el valor de la posicion inicial, velocidad y aceleración.

7. **Desafio** (obligatorio): Teniendo la planilla de notas del primer parcial del 1C2024 de los alumnos de la materia "Pensamiento Computacional", importar los datos como dataframe usando el siguiente código:

```
import pandas as pd

url_sheet = "https://docs.google.com/spreadsheets/d/17ei_NER5i_R-9QLnkeIjNprzTyoSqGowJ8y
df = pd.read_csv(url_sheet)
df.head()
```

Si la nota es “NaN”, significa que el o la estudiante no rindió el examen.
En gráficos diferentes:

a. Realizar un gráfico de torta que muestre el porcentaje entre la cantidad de alumnos que asistieron al parcial y los que no.
b. Realizar un gráfico de barras que muestre la cantidad de alumnos que aprobaron y desaprobaron el parcial, sin contar a los ausentes.
c. Realizar un gráfico de puntos que muestre la relación entre el curso y la nota obtenida en el parcial. ¿Diría que existe una relación?
d. Realizar un gráfico de linea que muestre la evolución de la nota promedio de los aprobados a través de los cursos.

Contacto

Por temas administrativos, podes contactar al plantel docente de la materia enviando un mail a:
`pc-cbc-docentes@googlegroups.com`

Por otros temas, te recomendamos que te comuniques por el servidor de Discord.

Discord

La materia usa Discord como plataforma adicional para la resolución de los ejercicios de las guias.
Tengan a bien leer con atención el mensaje de bienvenida y las reglas de convivencia. Pueden ingresar al servidor a través del siguiente link.

Dejanos Feedback!

Podés dejarnos feedback de las clases prácticas completando este formulario.

Ser Docente

Para ser docente de la materia es necesario haber aprobado el CBC y ser estudiante de FIUBA. Si cumplís con estos requisitos y te interesa ser docente de la práctica, podés escribirnos a nuestro mail de Contacto (arriba) para más información.
La materia se dicta los días lunes, martes, jueves y viernes, en los turnos 7-10, 10-13 y 13-16.