Pensamiento Computacional

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

2023-12-27

Table of contents

Pensamiento Computacional

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

Docentes de la Cátedra

- **Prof. Titular:** Méndez, Mariano
- Areco, Lucas
- Bulacios, Juan
- Cáceres, Fernando
- Capón, Lucía
- Corti, Bautista
- Duchen, Leonardo
- Duzac, Emilia
- Fulco, Victoria
- Ginestet, Joaquin
- 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:

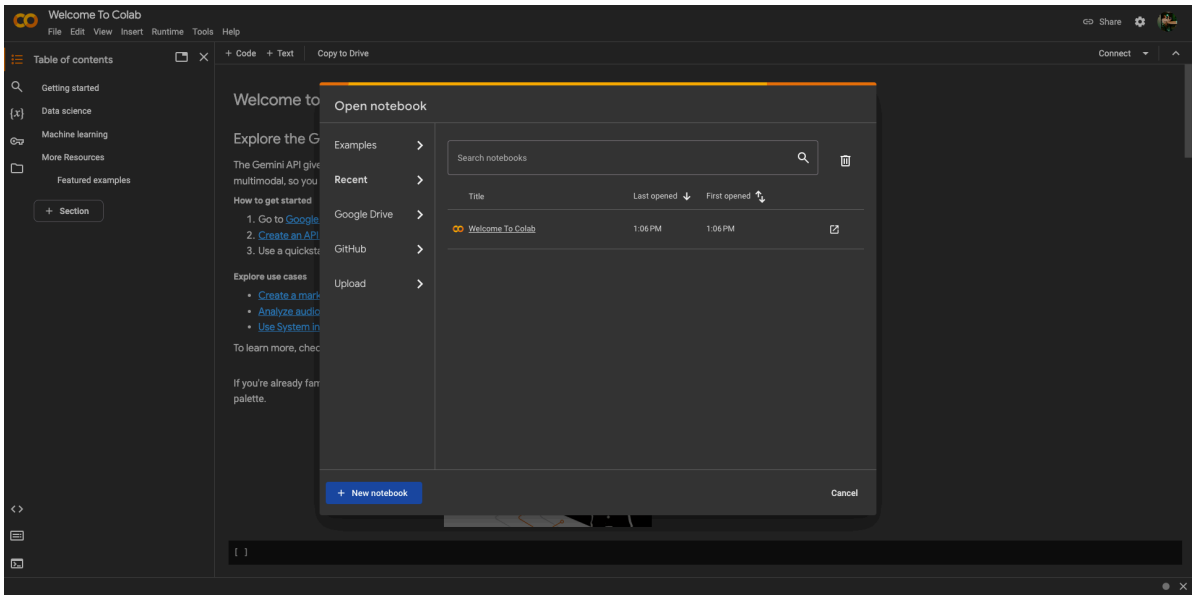


Figure 1.1: Inicio en Google Colab

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.

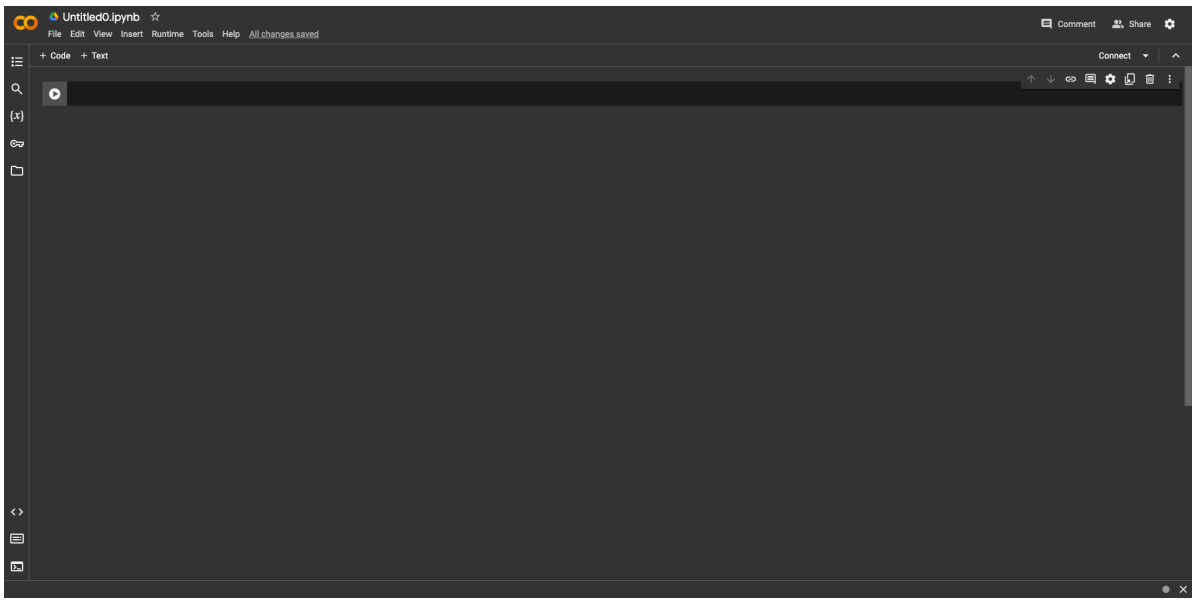


Figure 1.2: Archivo nuevo

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.

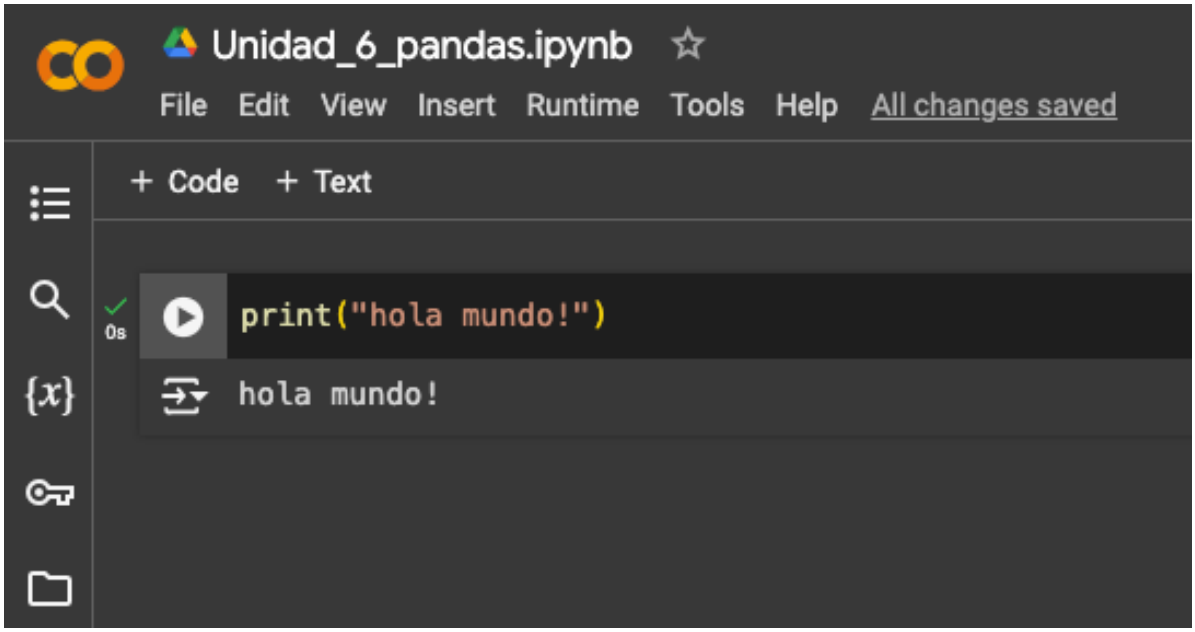


Figure 1.3: Ejecución de una celda de código

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.

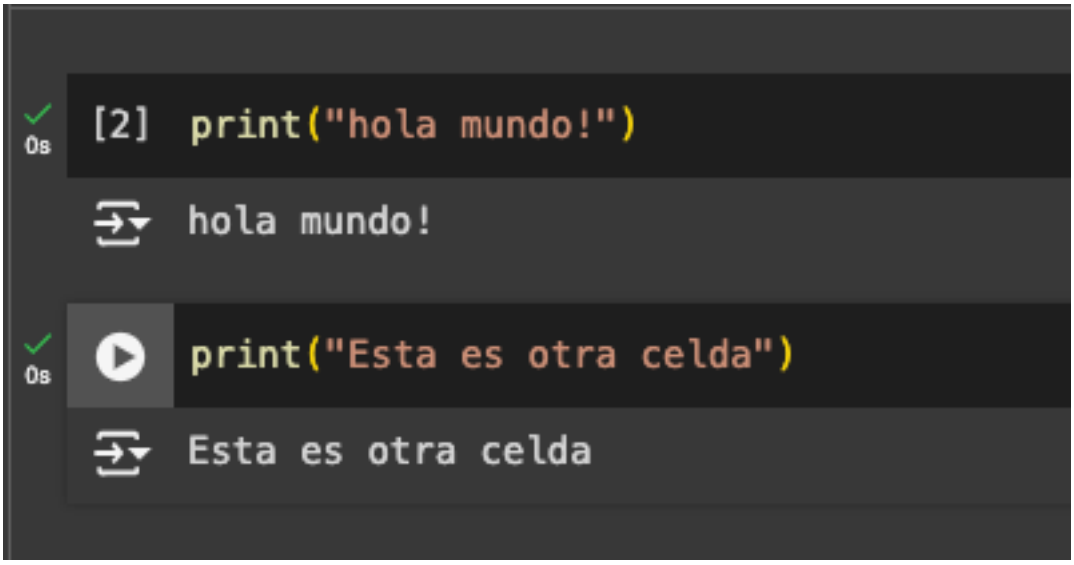
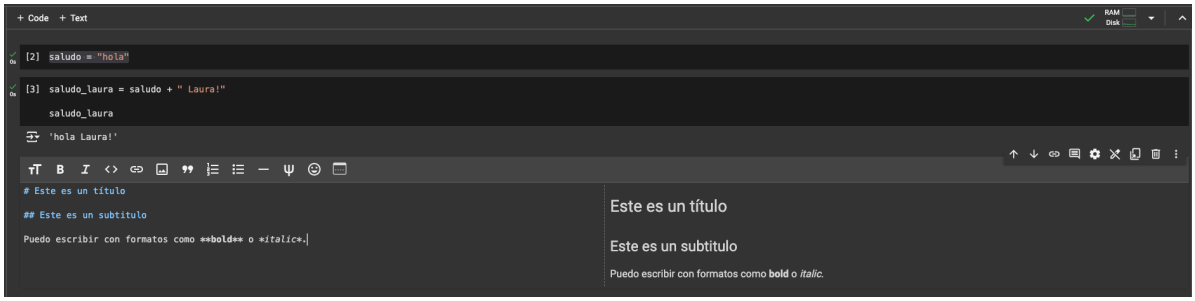


Figure 1.4: Ejecución simultánea de dos celdas de código

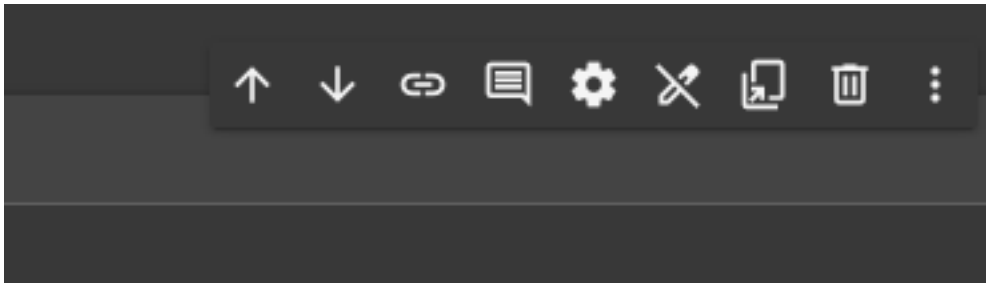
1.3.1.2 Celdas de Texto

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.

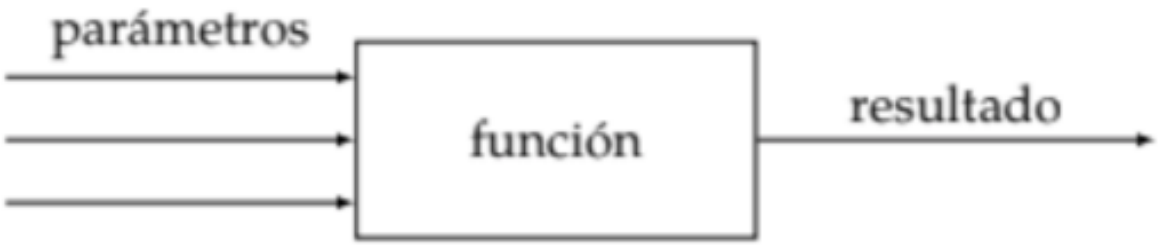


Figure 2.1: Una función recibe parámetros y devuelve un resultado

> **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!".

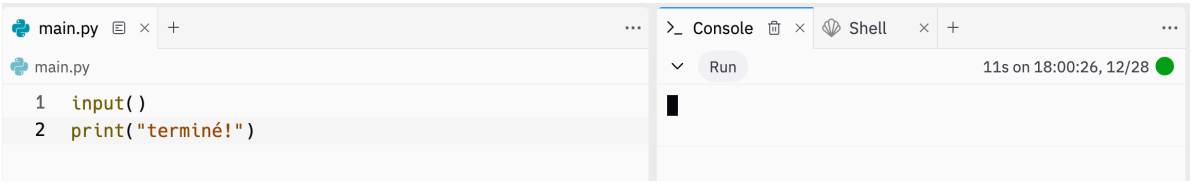


Figure 2.2: Input bloquea el flujo del programa

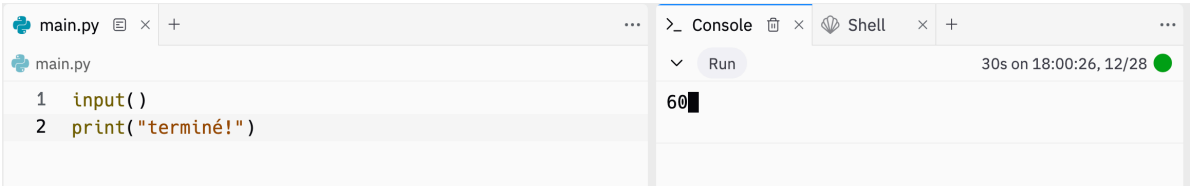


Figure 2.3: Ingresamos un valor (puede ser un número, texto, o ambos)

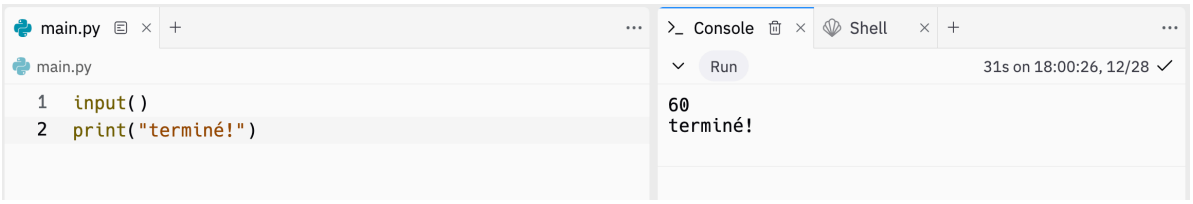


Figure 2.4: Al presionar Enter, la consola muestra el mensaje "terminé!"

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 + "!")
```

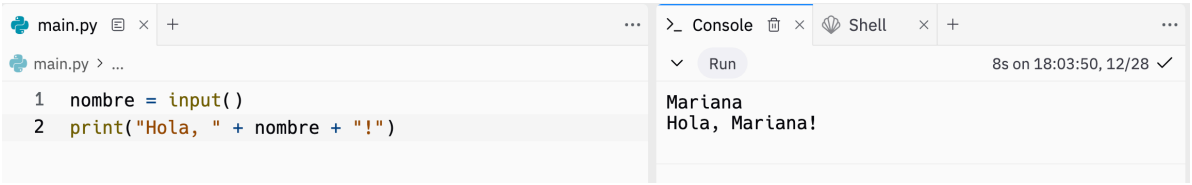


Figure 2.5: Ingresamos “Mariana” y presionamos Enter.

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 + "!")
```

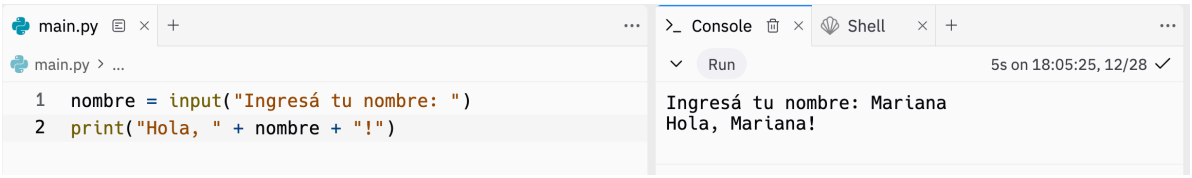


Figure 2.6: Ingresamos “Mariana” y presionamos Enter.

> ⚠ ¡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 (interpolación 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)
```

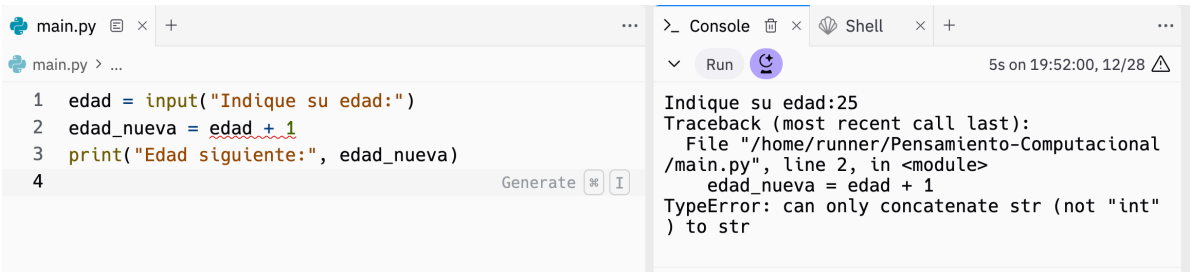


Figure 2.7: Ejecución del bloque de código

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')
```

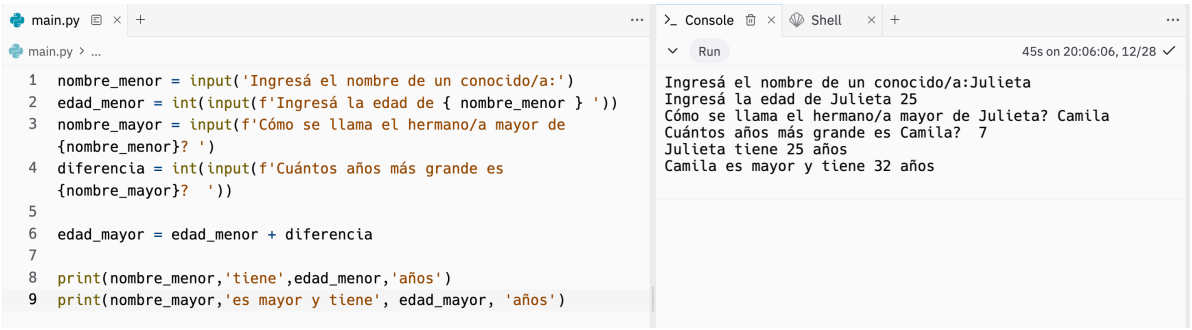


Figure 2.9: Ejecución del bloque de código

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 Operadores 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.

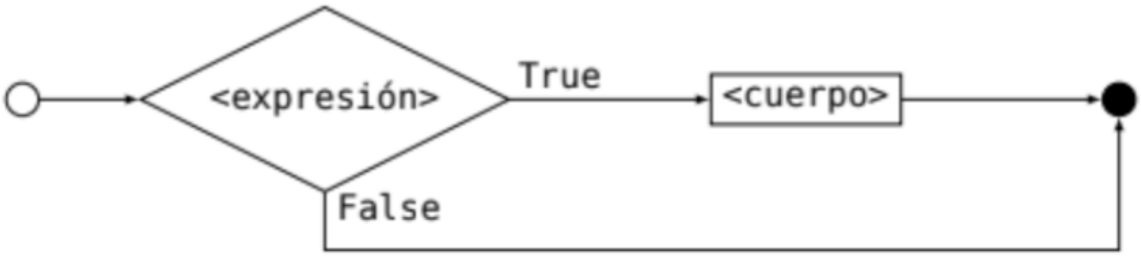


Figure 3.1: Diagrama de Flujo para la instrucción `if`

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`.

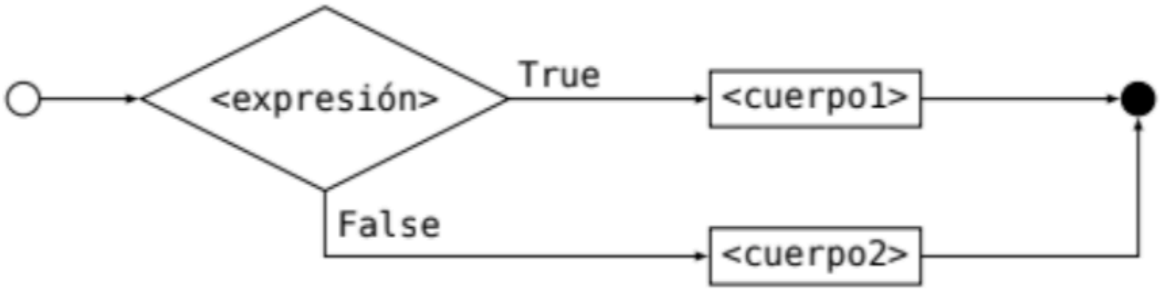


Figure 3.2: Diagrama de Flujo para la instrucción `if-else`

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:

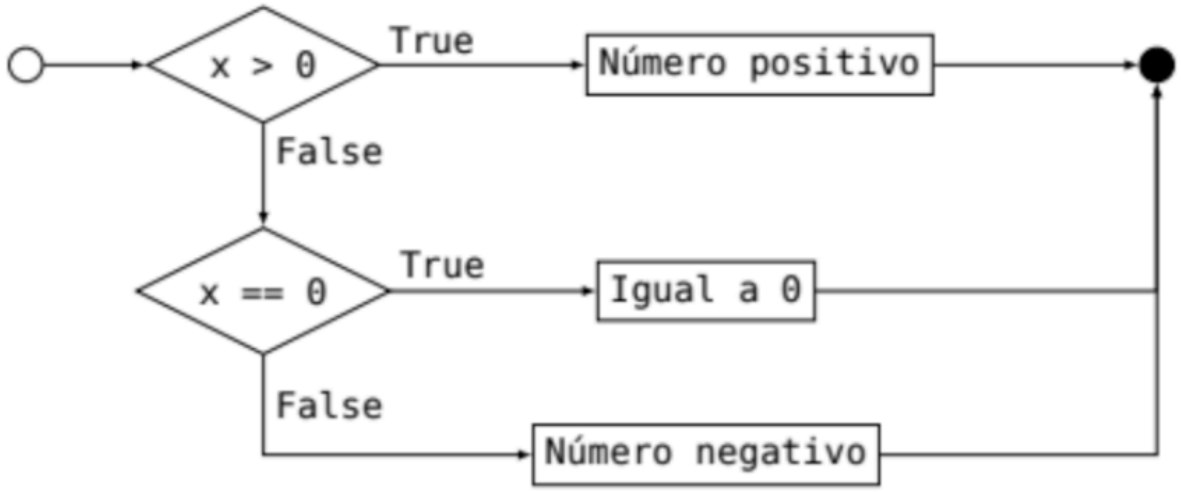


Figure 3.3: Diagrama de Flujo para la instrucción `if-elif-else` del ejemplo

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

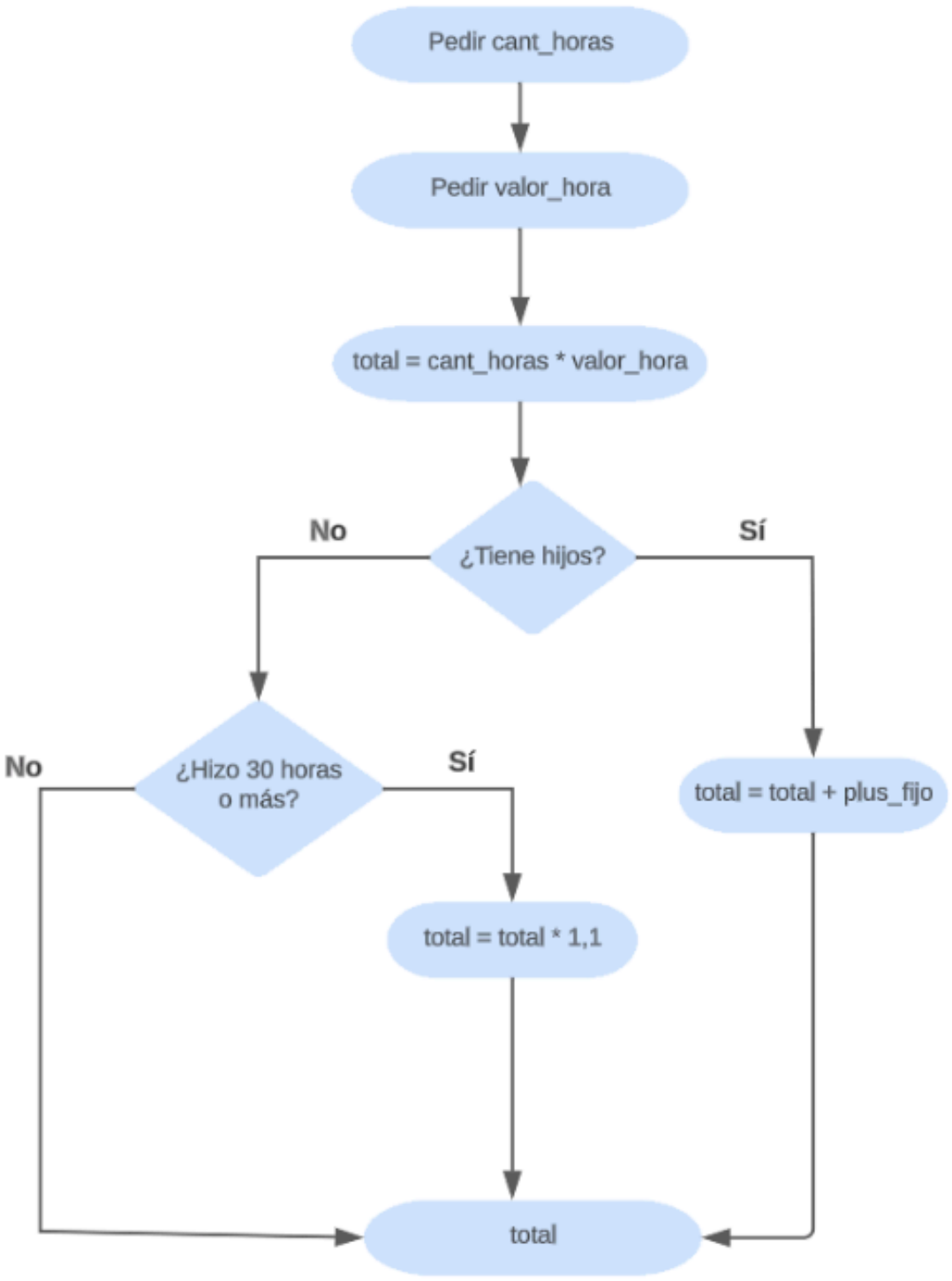


Figure 3.4: Diagrama de Flujo para el desafío

3.2 Ciclos y Rangos

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`

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

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

-
-
-

-
-
-

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

```
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)
```

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

imprimir_inverso()
```

3.2.1.1 Iterables

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

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

3.2.2 Ciclo `while`

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

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

-
-
-
-

> **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
```

3.2.3 Break, Continue y Return

> **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

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

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

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

3.2.3.2 Continue

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

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

> **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

```
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)
```

3.2.4 Consideraciones del While

3.2.4.1 No repitas

```
while <condicion>:
  <cuerpo>

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

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

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

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

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

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

print("El número es 3")
```

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

  if numero == 3:
    break
```

1.

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

2.

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

3.2.4.2 While True

3.2.4.3 Modificando la Condición

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

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

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

```
<setear condición>

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

-
-
-
-

```
i = 0 # setear condición

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

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

4.2 Rangos

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

4.3 Cadenas de Caracteres

1.

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

2.

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

3.

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

4.3.1 Métodos de Cadenas de Caracteres

4.3.1.1 Longitud de una Cadena

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

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

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

4.3.1.3 Buscando Subcadenas

unidad 3

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

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

4.3.1.4 Inmutabilidad

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

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

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

4.3.1.5 Otros Métodos de Cadenas de Caracteres

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

4.4 Tuplas

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

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

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

-
-

4.4.1 Tuplas como Secuencias

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

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

4.4.2 Tuplas como Inmutables

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

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

4.4.3 Longitud de una Tupla

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

4.4.4 Empaquetado y desempaquetado de tuplas

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

print(d)
```

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

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

¡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

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

lista_vacia = []
```

4.5.1 Longitud de una Lista

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

4.5.2 Listas como Secuencias

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

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

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

4.5.3 Listas como Mutables

-

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

-

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

-

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

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

-

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

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

-

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

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

-

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

4.5.4 Referencias de Listas

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

print(b)
```

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

print(b)
```

4.5.5 Búsqueda de Elementos en una Lista

-

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

-

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

4.5.6 Iterando sobre Listas

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

4.5.7 Ordenando Listas

-

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

print(lista)
print(lista_nueva)
```

-

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

print(lista)
```

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

¡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:

```python
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

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

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

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
```

```
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]]`

```
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)
```

4.6 Listas y Cadenas

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

print(lista)
```

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

print(cadena)
```

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

4.7 Operaciones de las Secuencias

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

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

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

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

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

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

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

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

print(lista)
```

> **!** 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

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

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

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

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

> **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

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

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

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

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

> **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`.

```
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)
```

```
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)
```

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

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

-
-

4.8.1 Diccionarios en Python

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

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

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

4.8.2 Accediendo a los Valores de un Diccionario

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

⚠ ¡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'
```

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

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

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

4.8.3 Iterando Elementos del Diccionario

4.8.3.1 Por Claves

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

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

4.8.3.2 Por Valores

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

4.8.3.3 Por Clave-Valor

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

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

> **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

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

> **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

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

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

4.8.7 Ordenamiento de Diccionarios

```python
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)
```

> **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

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

5.1.1 Abriendo un Archivo

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

5.1.2 Leyendo un Archivo

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

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

archivo.close()
```

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

```
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
```

5.1.2.1 Resumen

-
-

-

5.1.3 Escribiendo en un 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`

5.1.4 Tipos de acceso

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.

```
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()
```

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

for línea in líneas:
```

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

```
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

```
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.

```
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

```
# Abrimos el archivo de notas
def calcular_guardar_promedio(ruta_notas): # La función puede recibir "notas.csv"
  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
    datos = línea.strip('\n').split(";")
    nota = datos[1] # 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()
```

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

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

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

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

```
6.5
```

[1]

[2]

unidad 4

5.1.7 Tipos de archivos

[3]

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

5.1.8 Conclusiones

-
-
-

5.2 Colab y Archivos

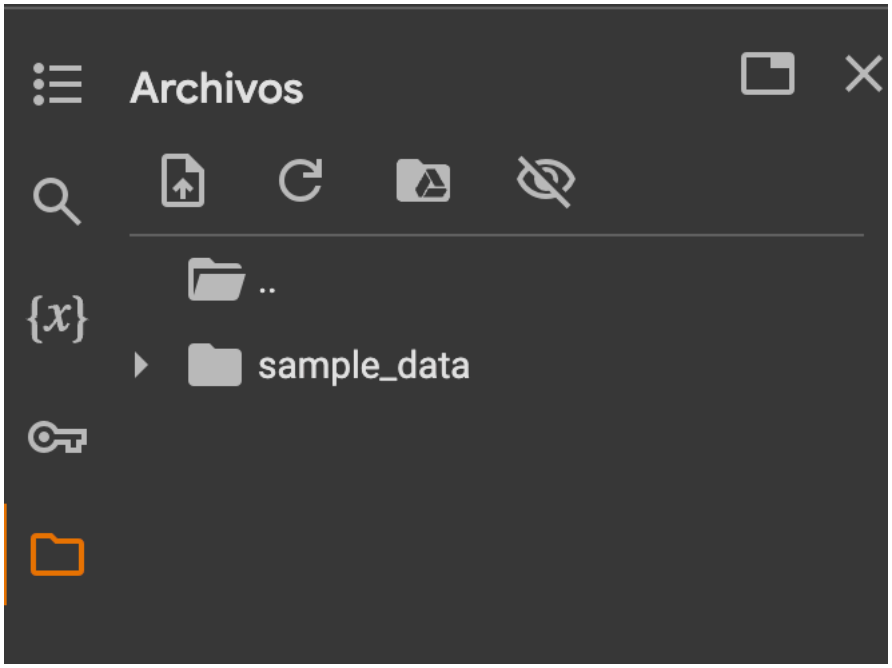


Figure 5.2: Menu Derecho

-
-
-
-

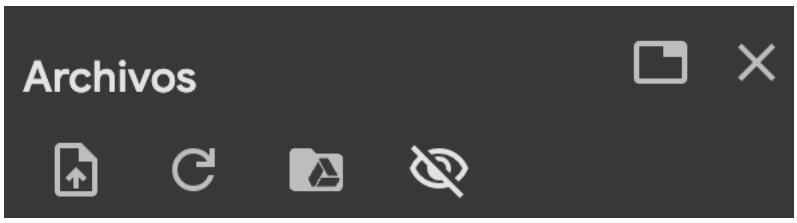


Figure 5.3: Menú Archivos

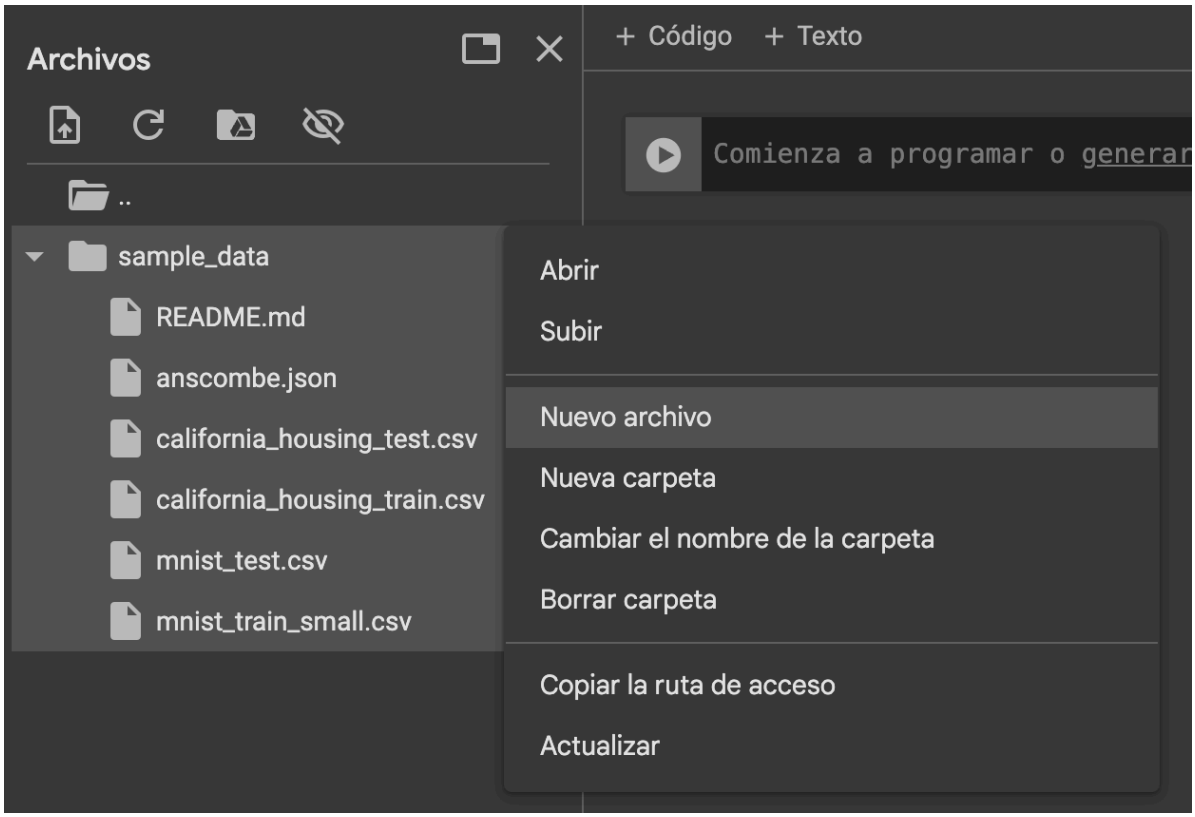


Figure 5.4: Nuevo archivo en sample data

> **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).

5.3 Manejo de errores

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

```
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
```

```
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
```

```
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'
```

-
-
-

-
-
-

-
-
-

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

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

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

```python
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")
```

> **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

5.3.3 Conclusiones

-
-
-

5.3.4 Bonus Track: Tipos de Errores

6 Bibliotecas de Python

6.1 Introducción

6.1.1 ¿Cómo se utilizan las bibliotecas?

```
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

6.3 NumPy

```
import numpy as np
```

6.3.1 Arrays

6.3.1.1 Creación de un Array

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

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

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

```
# 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)
```

```
# 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)
```

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

print(matriz)
```

6.3.1.2 Atributos de un array

6.3.1.3 Dimensión

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

6.3.1.4 Forma

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

6.3.1.5 Tamaño

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

6.3.1.6 Posiciones

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

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

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

6.3.1.7 Modificar arrays

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

print(a)
```

6.3.1.7.1 Reshape

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

print(a_reshape)
```

6.3.1.7.2 Insert

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

-
-
-
-

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

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

6.3.1.7.3 Append y Delete

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

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

6.3.1.7.4 Concatenate y Sort

```
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)
```

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

6.3.2 Operaciones aritméticas utilizando array

```
# 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)
```

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

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

6.3.2.1 Operaciones básicas:

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

-

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

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

-

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

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

-

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

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

-

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

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

-

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

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

> **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)
```

6.3.2.2 Logaritmo:

```
# 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]))
```

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

```
np.e
```

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 arcocoseno
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:

```
# 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))
```

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

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

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

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

> **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))
```

6.3.2.4 Más operaciones útiles:

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

-

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

-

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

-

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

-

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

> **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

```
import pandas as pd
```

	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

6.4.1 Serie

6.4.1.1 Creación de una Serie

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

pd.Series(lista)
```

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

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

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

6.4.1.2 Accediendo a un elemento

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

6.4.1.3 Operaciones con Series

```
a + a
```

```
a * 3
```

6.4.2 DataFrame

6.4.2.1 Creando DataFrames

```
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
```

6.4.2.2 Atributos y descripción de un Dataframe

```
data = {'nombre': ['José Martínez', 'Rosa Díaz', 'Javier Garcíaz', '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
```

> **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

-
-
-
-

```
df.info()
```

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

```
# índice
df.index
```

6.4.2.2.2 Shape y Size

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

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

6.4.2.2.3 Head y Tail

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

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

6.4.2.2.4 Describe

```
df.describe()
```

-
-
-
-

```
df.count()
```

```
df.min()
```

```
df.max()
```

6.4.2.3 Accediendo a filas de un DataFrame

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

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

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

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

6.4.2.4 Accediendo a columnas de un DataFrame

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

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

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

6.4.2.5 Modificar un Dataframe

-
-
-

6.4.2.5.1 Rename, insert y drop

```
# 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
```

```
# 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
```

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

```
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()
```

```
# 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

```
# 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
```

> ¿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

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

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

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

6.4.2.8 Filtrar un Dataframe

```
# 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 ]
```

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

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

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

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

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

6.4.2.9 Ordenando y Contando

6.4.2.9.1 Sort value:

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

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

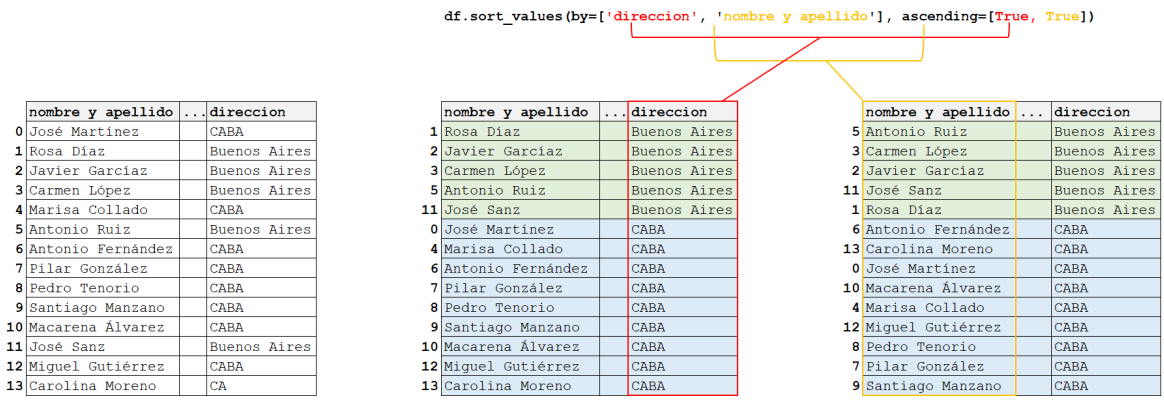


Figure 6.2: Ejemplo de sort_values()

6.4.2.9.2 Value Count:

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

6.4.3 Conclusiones

6.5 Matplotlib

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

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

6.5.1 Introducción

1.

2.

3.

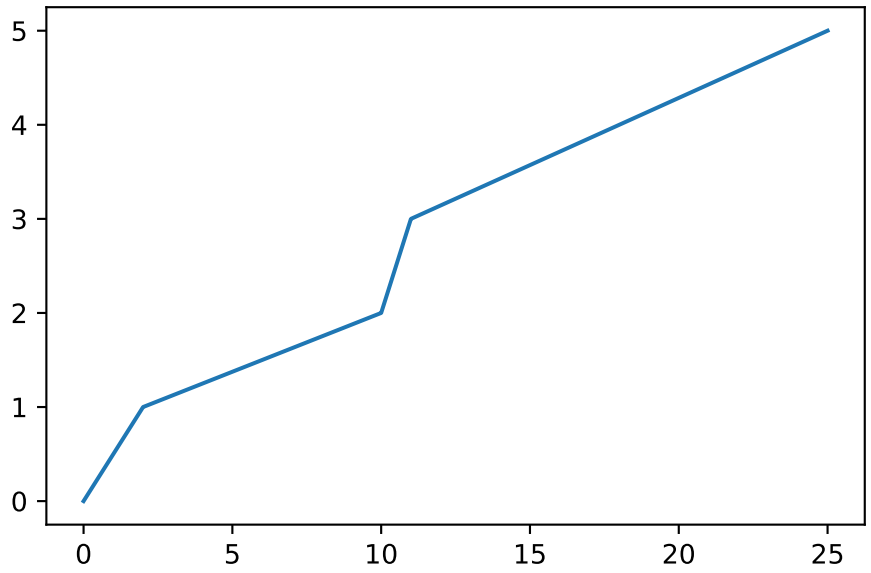
4.

```
plt.show()
```

6.5.2 Creando una figura

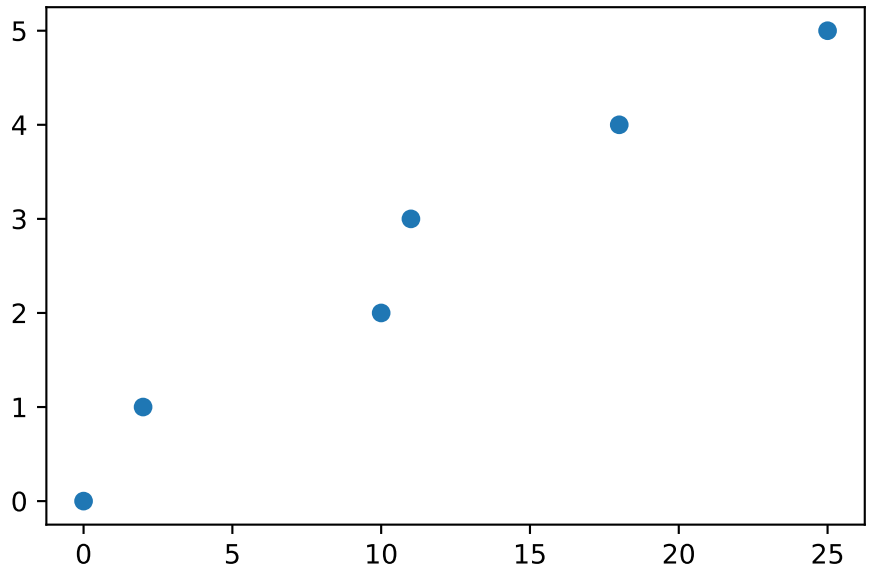
6.5.2.1 Gráfico de línea

```
# 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()
```



6.5.2.2 Gráfico de puntos

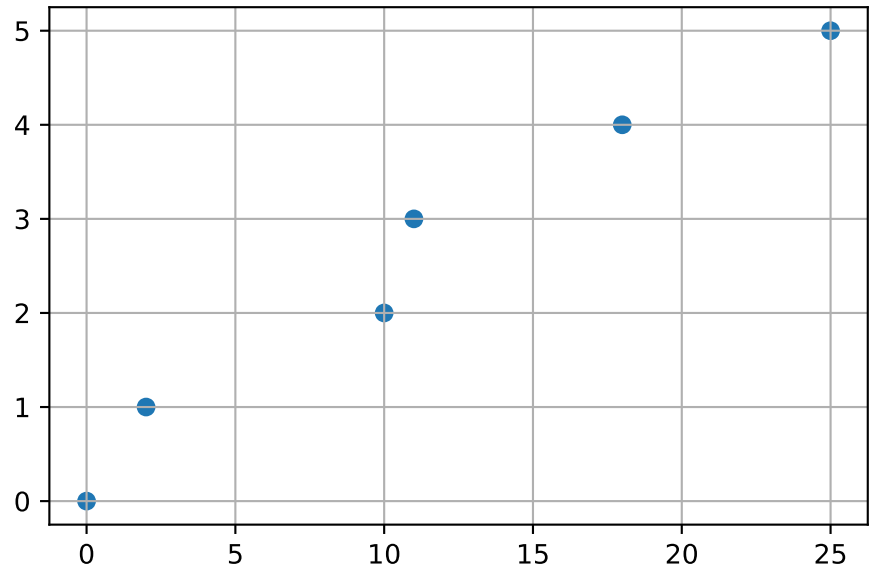
```
# Gráfico de puntos
fig2, ax2 = plt.subplots()
ax2.scatter(x, y)
plt.show()
```



! Grillas

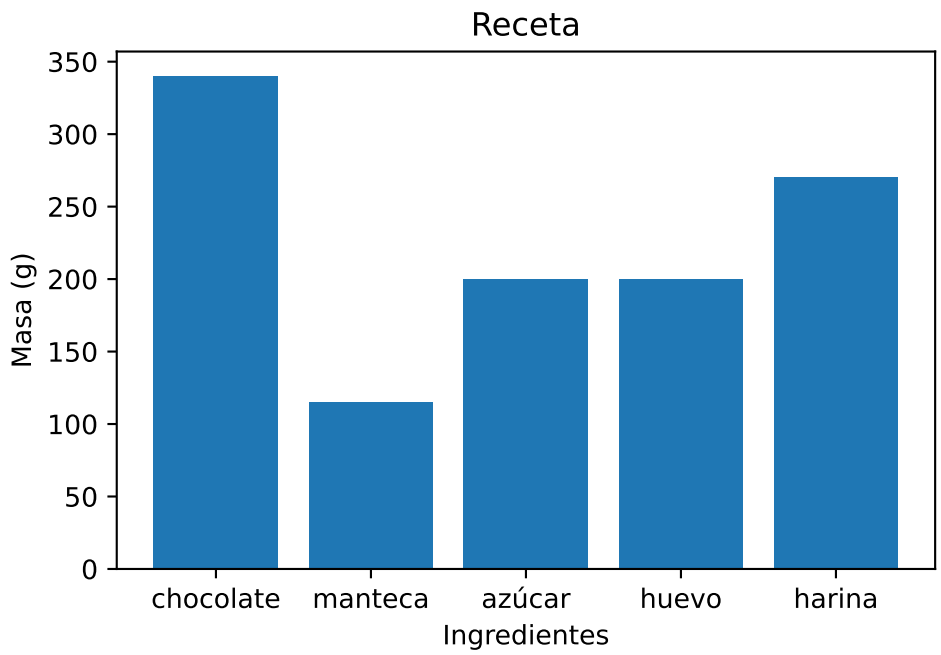
¿Ves cómo en el gráfico de puntos quizás no se entiende bien la ubicación de cada punto? Esto es porque no tenemos una guía que nos ayude. Para eso, vamos a agregar una grilla. La grilla se puede agregar con al función `grid()`, y es una buena forma de darle legibilidad a un gráfico como puede ser el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

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



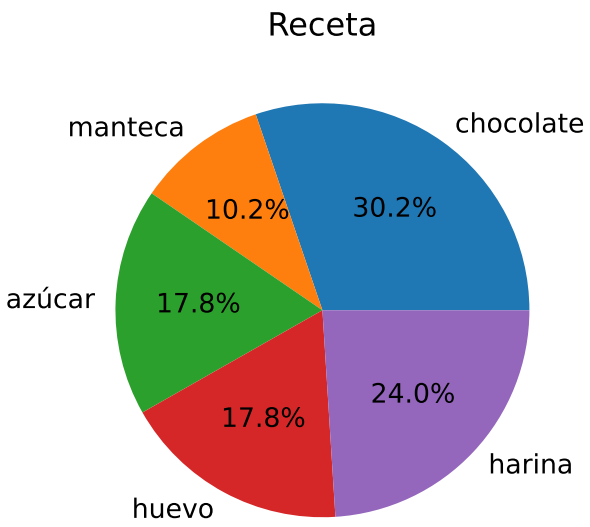
6.5.2.3 Gráficos de Barras

```
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()
```



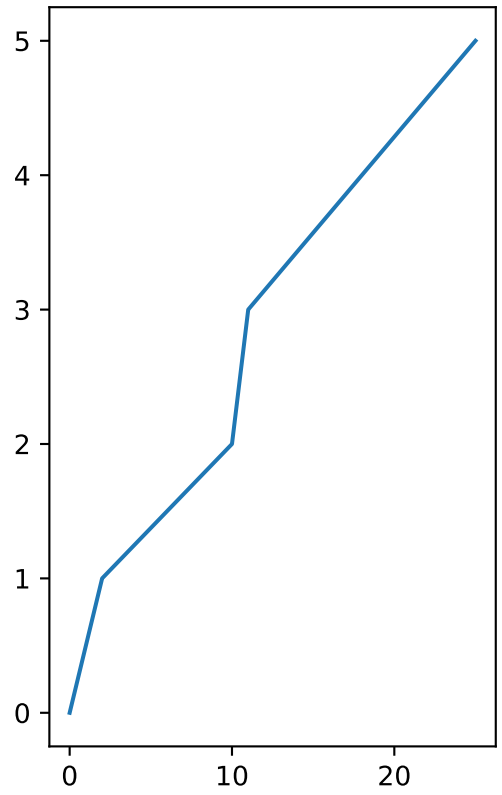
6.5.2.4 Gráfico de torta

```
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

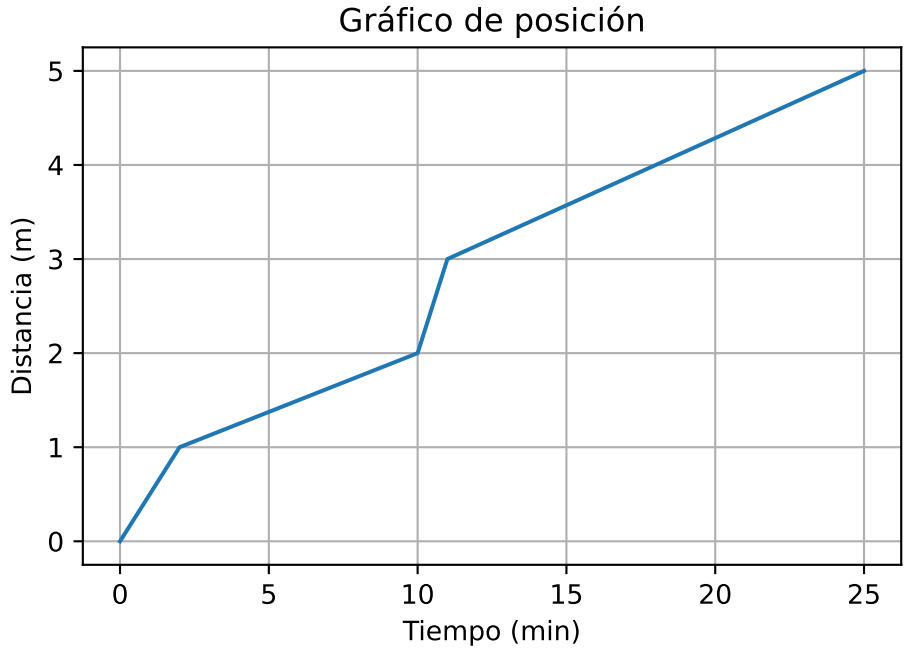
```
x = [0,2,10,11,18,25]   # Tiempo (min)
y = [0,1,2,3,4,5]       # Distancia (m)

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

ax.plot(x, y)

plt.show()
```



6.5.3 Títulos

```
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)
```

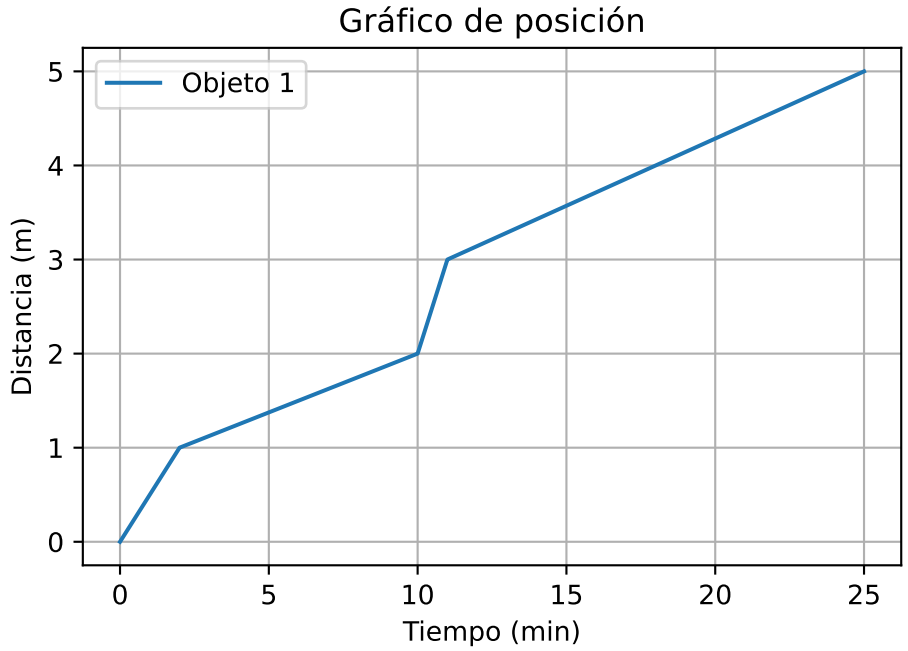
```
# Mostrar el título del gráfico
ax.set_title("Gráfico de posición")

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


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



6.5.4 Referencias

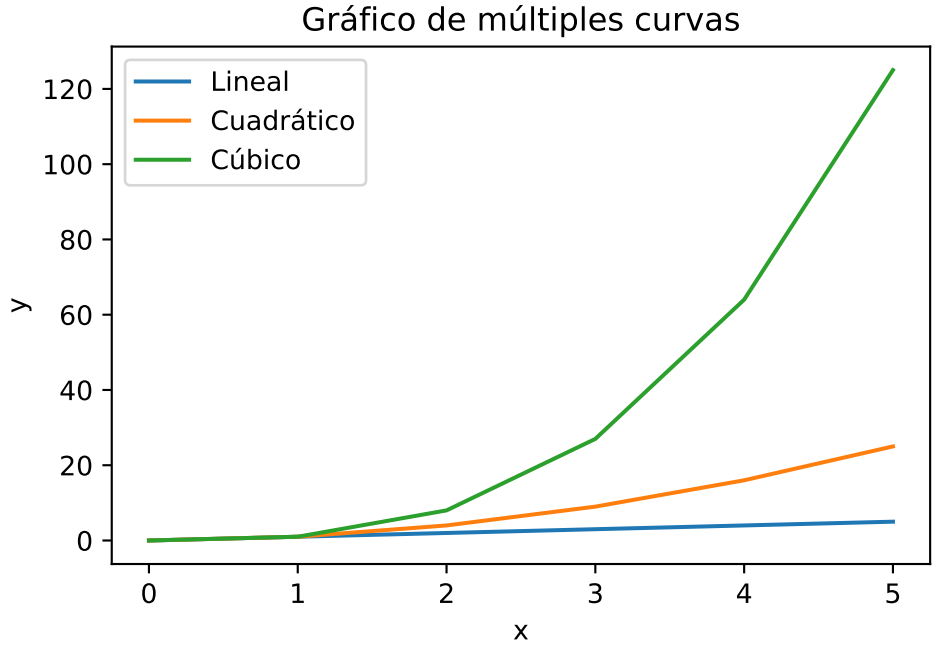
```
x = [0,2,10,11,18,25]    # Tiempo (min)
y = [0,1,2,3,4,5]        # Distancia (m)
```

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

ax.plot(x, y, 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

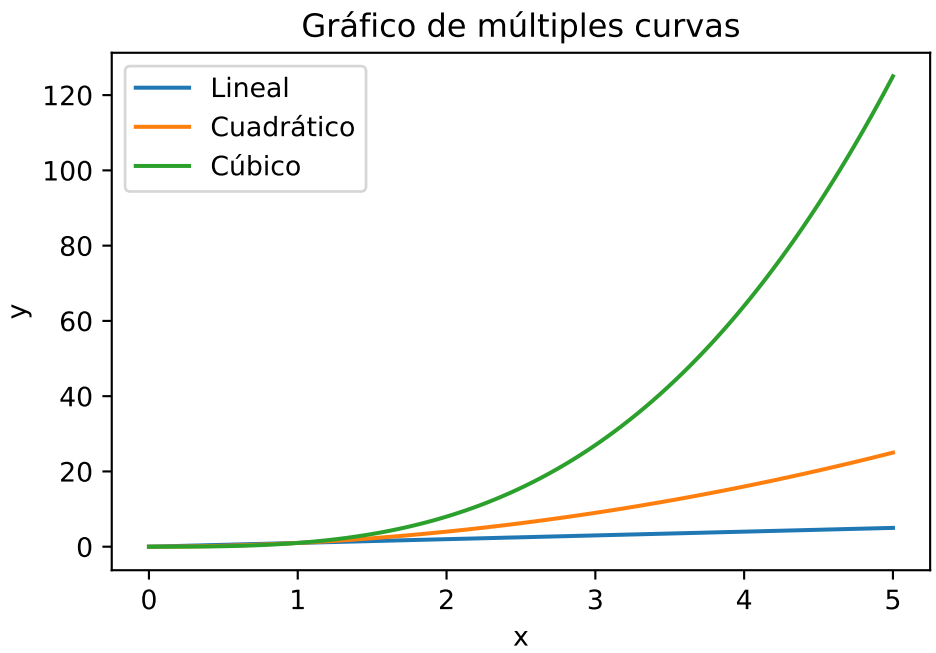
```
# Valores que se desean graficar
x = [0, 1, 2, 3, 4, 5]
y_linear = [0, 1, 2, 3, 4, 5]
y_quadratic = [0, 1, 4, 9, 16, 25]
y_cubic = [0, 1, 8, 27, 64, 125]

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()
```



6.5.6 Uniendo Bibliotecas

6.5.6.1 Matplotlib y Numpy

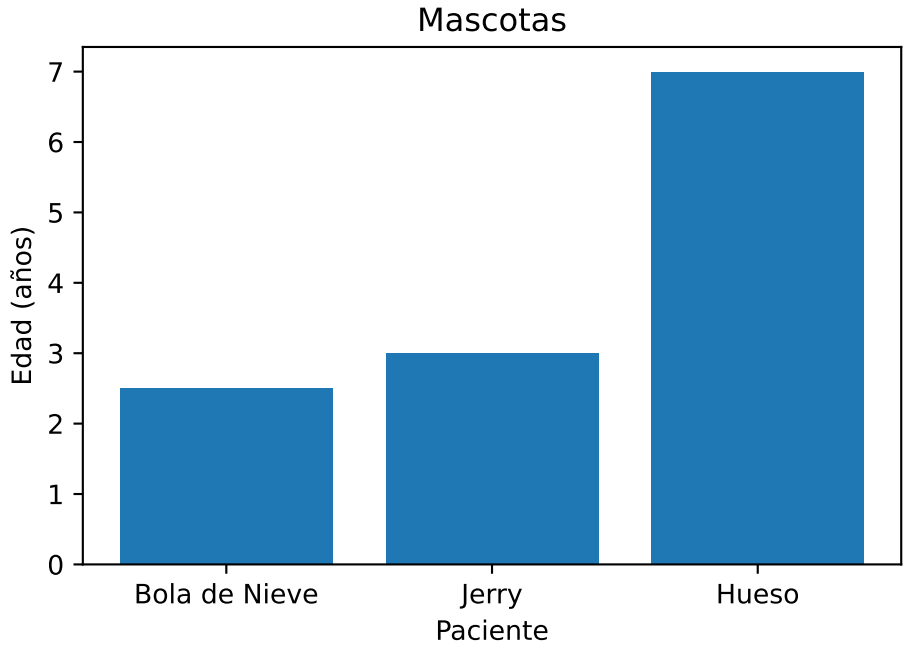
```
import numpy as np

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

```
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
```

```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.

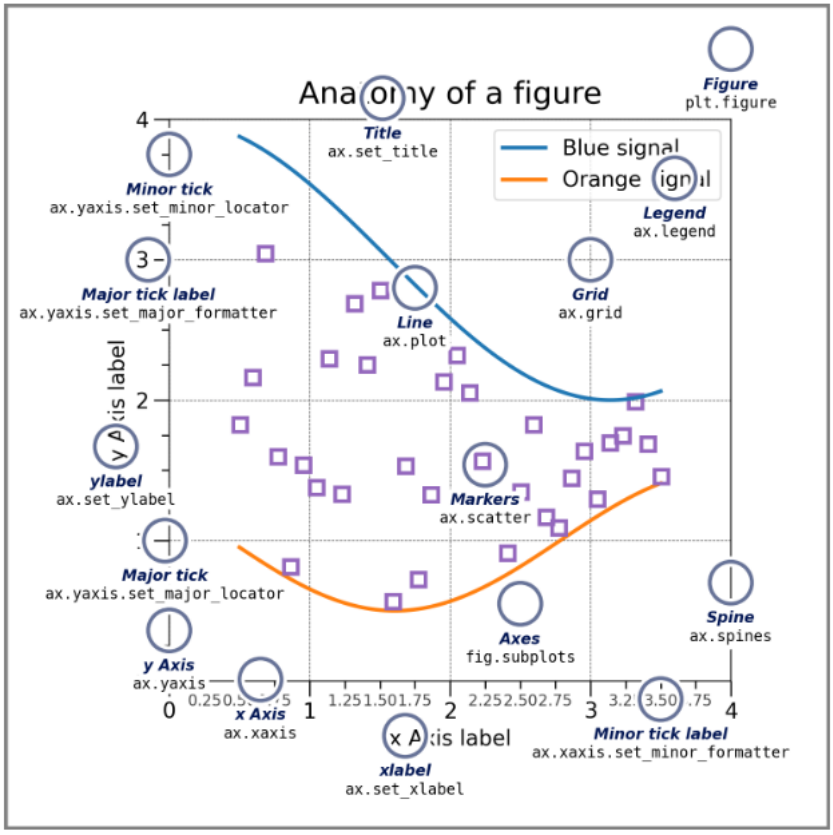


Figure 6.3: Partes de una Figura. Si querés conocer más detalle, podés ingresar a este link.

-
-
-
-

6.5.7.1 Girando los labels

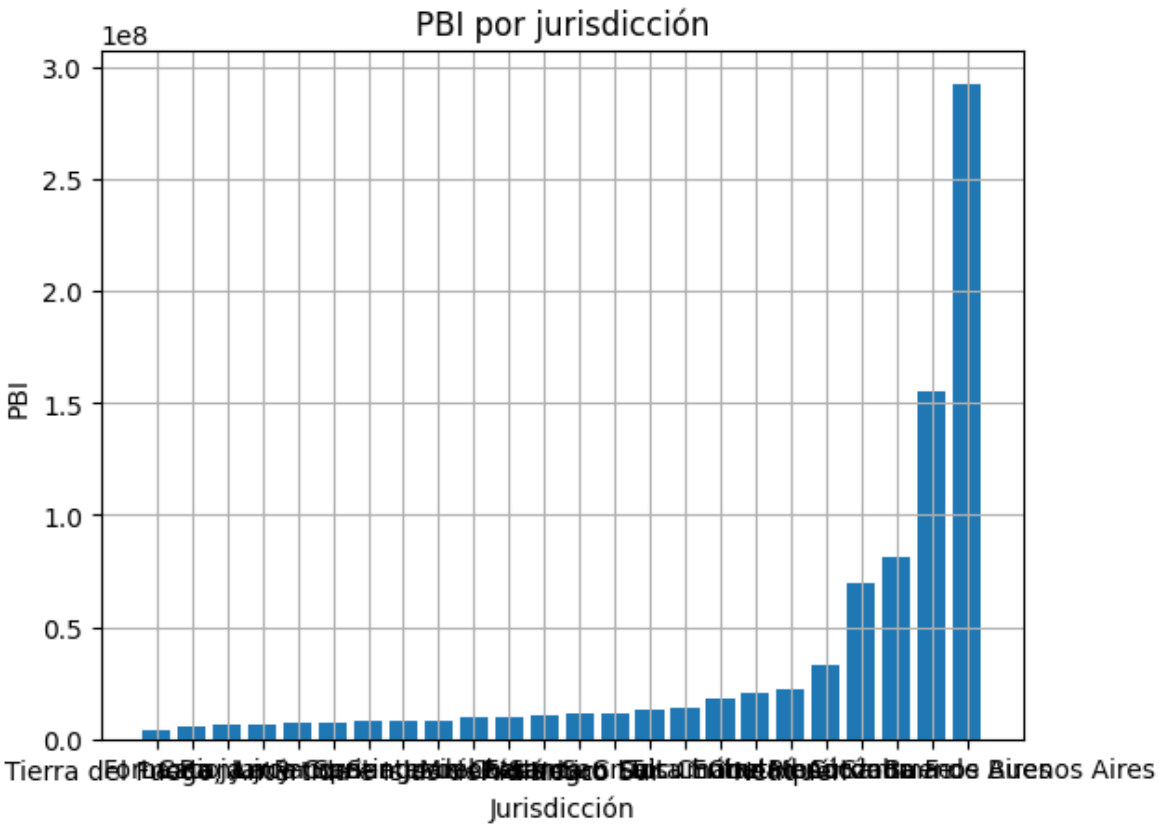


Figure 6.4: Los labels son muy grandes y no hay espacio para mostrarlos correctamente

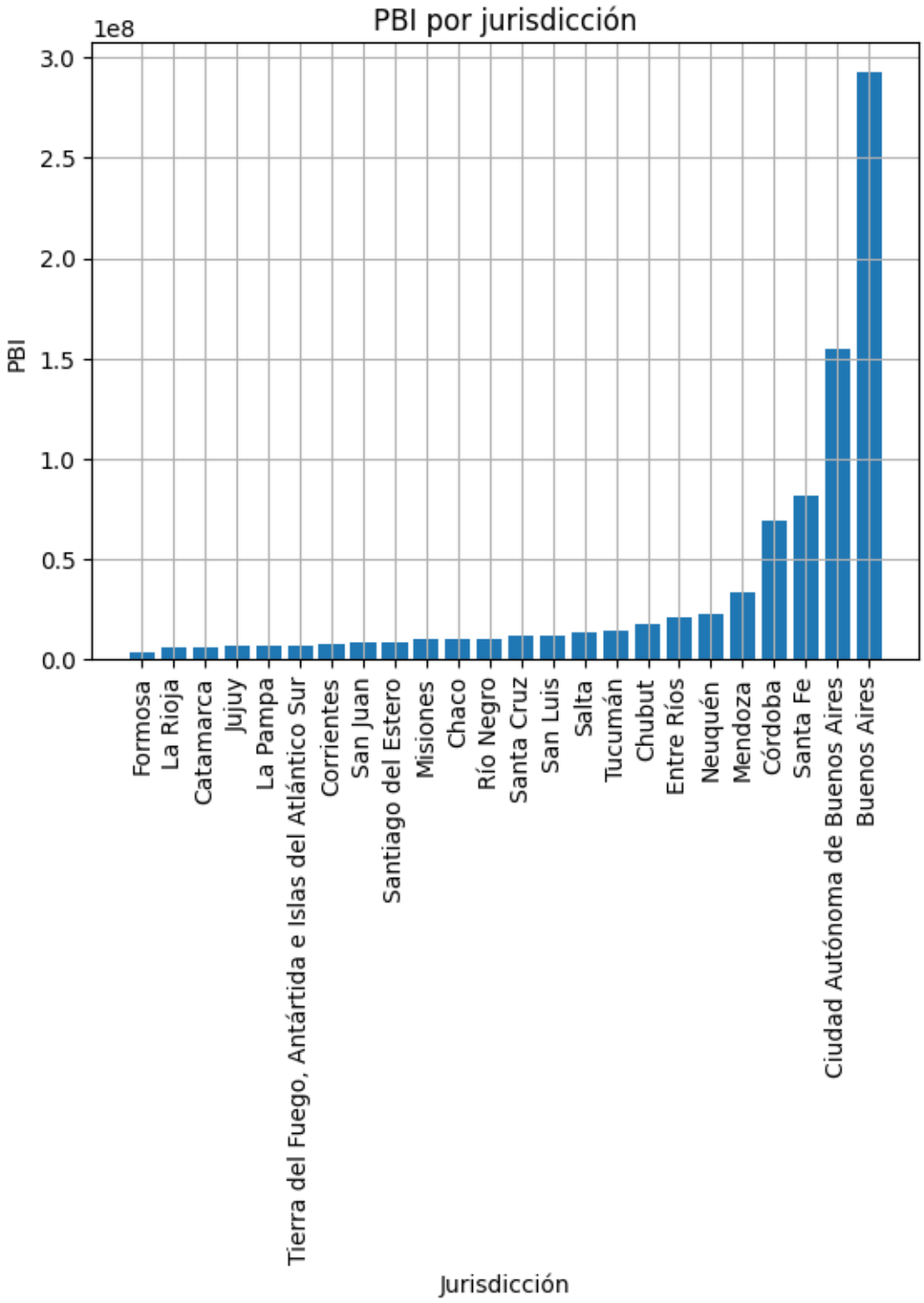
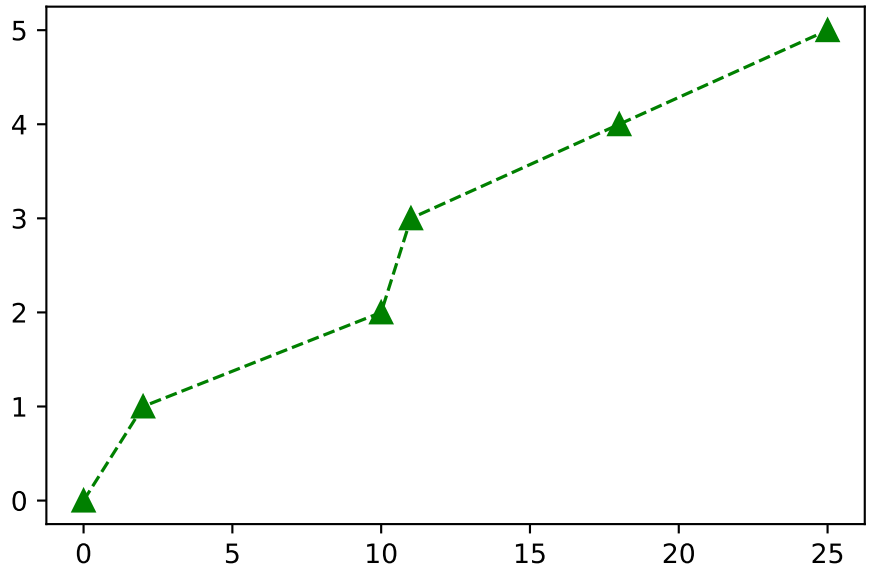


Figure 6.5: Los labels presentan un ángulo de 90 grados

6.5.7.2 Cambiando colores y estilos

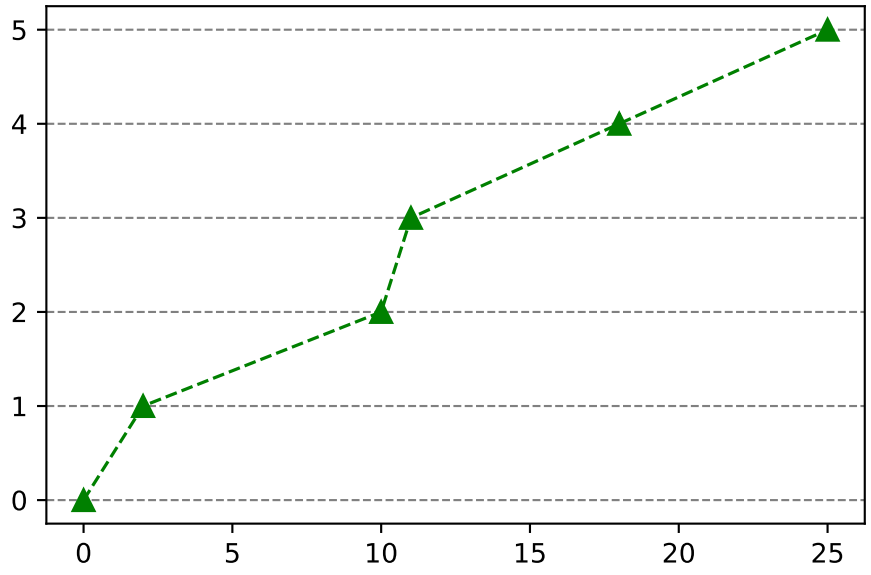
-
-
-
-

```
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()
```



6.5.7.3 Grilla personalizada:

```
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()
```



Guía de Ejercicios

Recomendaciones al realizar las guías

-
  -
  -
  -
  -
-
-
  -
  -
-
-

> **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.

2.

3.

4.

5.

6.

7.

Guía 2: Tipos de Datos, Expresiones y Funciones

1.
2.
3. a.

   b.

4.

   a.
   b.
   c.

5.

6.

7.

   a.
   b.

8.
9.

10.

11.

12.

13.

14. a.

b.

c.

15.

16.

a.
b.
c.
d.
e.
f.
g.
h.
i.
j.
k.
l.
m.
n.
o.
p.
q.
r.
s.
t.

17. a.

b.

18.

19.

20.

Guía 3: Estructuras de Control

1. Decisiones

1.
2.

3.

4.

a.
b.
c.

> 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.

   a.

   b.

   c.

6.

7.

2. Ciclos

1.

a.
b.

c.
d.
e.

2.

3.

4. a.

b.

5.

6.

a.

b.

c.

7. a.

b.

c.

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

> **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.

b.

9.

10.

11.

a.

b.

c.

d.

12.

a.

b.

Guía 4: Tipos de Estructuras de Datos

1. Cadenas de caracteres

1.

   a.

   b.

   c.

   d.

   e.

2.

3.

   a.

   b.

4.

   a.

   b.

5.

6.

7.

8.

   a.

   b.

   c.

9.

2. Rangos, Tuplas y Listas

1.

   a.
   b.

c.

2.

a.

b.

c.

3. a.

b.

4.

a.

b.

c.

d.

5.

a.

b.

6.

7. a.

b.

c.

8.

9.

10.

11.

12.

a.

b.

c.

13.

14.

15.

16.

17.

18.

19.

> **!** 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.

3. Diccionarios

1.

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

2.

    a.

    b.

3.

    a.

    b.

4.

    a.
    b.

5.

6.

a.

b.

c.

d.

7.

8.

9.

-
-
-

10.

a.
b.

11.

a.

b.

12.

13.

a.
b.

c.

14.

a.

b.

15.

16. https://www.donarg.com.ar/

-
-
-
-
-
-
-
-
-
-

a.

b.

c.

https://www.donarg.com.ar/

https://www.donarg.com.ar/dondedono

Guía 5: Entrada y Salida

1. Archivos

1.

2.

a.

3.

4.

5.

6.

7.

8.

9.

   a.

   b.

10.

```
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
.
.
.
```

11.

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

   -
   -
   -

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

12.

13.

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

14.

```
D2
F1
E4
```

```
Megamente
The Menu
Shrek
```

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

15.

```
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
```

16.

17.

18.

2. Manejo de Errores

1.

2.

3.

4.

```
25.5
26.0
24.5
# etc
```

5.

6.

```
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.

```
def dividir(n1, n2):
    return n1/n2
```

a.

b.

c.

8.

```
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
}
```

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

Guía 6: Bibliotecas de Python

1. Numpy

1.

   a.
   b.
   c.

2.

   a.
   b.

3.

   a.
   b.
   c.
   d.

4.

   a.
   b.

5.

   a.

   b.

c.

6.

7.

8.

9.

10.

11.

12.

    b.

2. Pandas

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

```
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
```

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.

13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.

a.

```
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
```

24.

```
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.

a.
b.
c.
d.
e.

2.

a.
b.

c.

3.

```
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]
```

a.

b.

c.

4.

```
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.

b.

c.

d.

> **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.

6.

-
-
-

-
-
-

7.

```
import pandas as pd

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

a.

b.

c.

Contacto

Discord

Dejanos Feedback!

Ser Docente