
INTRODUCCIÓN A LA COMPUTACIÓN

APUNTES DE TEÓRICO

Universidad de la República
Facultad de Ciencias
2025

Francisco Carballal

Prefacio

Este documento consta de material teórico para la edición 2025 del curso “Computación” de la Facultad de Ciencias de la Universidad de la República. Dentro de los dos cursos de computación dictados por el Centro de matemática, es el básico.

En el capítulo 1 se introducen conceptos de computación, incluyendo formalización matemática, hardware, software, lenguajes de programación y compiladores. Para entender el resto del documento, alcanza con la idea “por arriba”.

En el capítulo 2 se presentan los conceptos básicos de programación en python, sin asumir conocimientos ni experiencia previa. Se trata de un material bastante autocontenido, con explicaciones y ejemplos además de las definiciones.

En el capítulo 3 estudiamos la programación de funciones recursivas con enteros y listas y hacemos una pequeña introducción al estudio de eficiencia de algoritmos. Es el capítulo con más formalismo matemático.

En el capítulo 4 se introduce el concepto de modularidad y vemos varias bibliotecas de python, con especial énfasis en algunas de computación científica. Aquí el enfoque es más de aprovechar cosas provistas que programar algoritmos desde cero.

En el capítulo 5 se introducen TeX y LaTeX. Como es en parte sobre TeX, hay cosas que no se consideran buenas prácticas en LaTeX. El objetivo no es solo introducir LaTeX de modo práctico, sino también aportar a la comprensión de su funcionamiento.

Índice general

1. Introducción	4
1.1. Conceptos básicos de computación e intuiciones	4
1.1.1. Bases de la formalización matemática	5
1.2. Arquitectura de Von Neumann	5
1.2.1. Ejemplo de programa	7
1.3. Lenguajes de programación y compiladores	9
2. Programación en python	11
2.1. Instalación y ejecución	11
2.2. Tipos de datos	13
2.2.1. Booleanos	14
2.2.2. Enteros	14
2.2.3. Números de punto flotante	14
2.2.4. Texto	15
2.2.5. Listas	16
2.2.6. Transformación entre tipos	17
2.3. Variables, expresiones y asignaciones	17
2.4. Indizado y slicing	19
2.4.1. Referencias múltiples	22
2.5. Programas e instrucciones	23
2.5.1. Más sobre print	26
2.6. Condicionales	27
2.7. Iteraciones	29
2.8. Funciones	33
2.8.1. Variables locales y globales	37
2.9. Metodología de resolución de problemas	38
2.9.1. Ejemplo	39
3. Funciones recursivas	44
3.1. Inducción y recursión	45
3.2. Funciones recursivas en python	46
3.2.1. Límite de recursión	49
3.3. Diseño y estudio de funciones recursivas	50
3.3.1. Recursión con números naturales	50
3.3.2. Recursión con listas	51
3.4. Eficiencia de algoritmos	54

3.4.1.	Búsqueda	57
3.4.2.	Ordenamiento	59
3.5.	Algoritmos exponenciales	60
3.5.1.	Funciones con un parámetro. Ejemplo Fibonacci	62
3.5.2.	Funciones con dos parámetros. Ejemplo combinaciones	65
4.	Modularidad	68
4.1.	Algunos otros elementos de python	68
4.1.1.	Tuplas y diccionarios	69
4.1.2.	Funciones como objetos y expresiones lambda	70
4.1.3.	Manejo básico de archivos	72
4.2.	Módulos	73
4.3.	Bibliotecas	74
4.3.1.	Ejemplos de la biblioteca estándar	75
4.3.2.	Ejemplos de otras bibliotecas	78
4.4.	numpy	81
4.4.1.	Arreglos	82
4.4.2.	Operaciones con arreglos	85
4.4.3.	Álgebra lineal	86
4.4.4.	Mínimos cuadrados	88
4.5.	matplotlib	89
4.6.	scipy	92
4.7.	sympy	94
5.	Introducción a TeX y LaTeX	97
5.1.	Instalación y uso de latex	98
5.2.	Estructura de un archivo latex	98
5.3.	Escritura básica de texto	99
5.4.	Modos	100
5.5.	Grupos y estructuras de control	101
5.6.	Cajas y pegamento	104
5.7.	Escritura matemática	108
5.8.	Macros	117
5.9.	Funcionalidades de latex	120
5.9.1.	Secciones	120
5.9.2.	Imágenes	121
5.9.3.	Referencias	123
5.10.	Cosas que no tratamos y comentarios finales	125

Capítulo 1

Introducción

Este es un curso sobre computación con una introducción a TeX y LaTeX al final. Aparte de ese último tema, a grandes rasgos se plantean tres objetivos. El primero es que aprendan las bases de la programación. El segundo es que desarrollen la capacidad de utilizar las computadoras con fines matemáticos/científicos, como por ejemplo para determinar si un número es primo, o aproximar numéricamente el valor de una integral, o generar gráficas de distintos tipos. El tercero es que desarrollen la capacidad de entender la computación como un concepto matemático abstracto, para desarrollar programas en base a razonamiento matemático y además poder demostrar propiedades sobre estos programas. De modo sintético, se incluyen programación, computación para matemática y matemática para computación.

El objetivo principal de este capítulo es transmitir una idea básica de qué es la computación, cómo funcionan las computadoras y qué son los lenguajes de programación.

1.1. Conceptos básicos de computación e intuiciones

La computación se trata de operaciones definidas por reglas precisas que operan con objetos finitos y en una cantidad finita de pasos llegan a un resultado.

Los objetos finitos, a los que en general llamaremos **datos**, pueden ser por ejemplo números naturales, palabras o listas finitas. Cabe aclarar que al decir que los números naturales son finitos nos referimos a que cada número natural es un objeto finito. Por supuesto que el conjunto de todos los números naturales, por otra parte, es infinito.

A una operación definida por reglas precisas que opera con datos y termina en finitos pasos le llamaremos **algoritmo**. Un programa es lo mismo que un algoritmo. Un ejemplo de algoritmo es el de la suma que se aprende en la escuela. Dados dos números, aplicando una secuencia finita de pasos bien definida se llega al resultado.

Una buena analogía consta de las recetas de cocina. Sin embargo, hay una diferencia importante. En las recetas de cocina, por la naturaleza de la situación, pueden haber ciertas ambigüedades, o puntos en los que hay que tomar una decisión, de modo que si dos personas realizan la misma receta el resultado puede ser distinto, sin que ninguno de los dos lo haya hecho mal. Pueden haber frases como «un poquito de sal» que según la persona se traduzcan a distintas cantidades. En computación, un algoritmo bien ejecutado debe tener un único resultado correcto, como ocurre con la suma. Que las reglas sean precisas significa que no hay margen de variabilidad en lo que se debe hacer, ni

puntos en los que un ser con voluntad propia deba pensar y tomar una decisión. Esto es importante por dos motivos. Primero, asegura que si se realiza por una persona, hay un único resultado correcto. Segundo (y tal vez más importante), es necesario para que pueda ser realizado automáticamente por una computadora, que en lugar de un ser con pensamiento y voluntad es una cosa.

Como los algoritmos deben estar bien definidos y no puede haber ambigüedad, son en esencia conceptos matemáticos —la computación teórica es una rama de la matemática.

1.1.1. Bases de la formalización matemática

Una forma de representar algoritmos en computación teórica es mediante funciones $f : \mathbb{N} \rightarrow \mathbb{N}$, donde la idea es que si n es la entrada del programa, entonces $f(n)$ es la salida. Como se trata de un algoritmo, no puede ser cualquier función, sino una que puede calcularse en finitos pasos con reglas precisas. A primera vista puede parecer que usar solamente números naturales es una restricción, pero de hecho alcanza para hablar de todo lo que se puede programar. De hecho, dentro de la computadora todos los distintos tipos de datos se representan con secuencias de bits (*binary digits*, es decir dígitos binarios: 0 o 1) y cualquier secuencia de bits se puede interpretar también como un número escrito en binario. Por lo tanto, todos los datos finitos se pueden representar con números (en general muy grandes). Debido a esto, un programa que recibe como entrada una imagen y retorna la imagen con alguna modificación, por ejemplo, se puede ver como una función $f : \mathbb{N} \rightarrow \mathbb{N}$ que recibe el número que codifica la imagen original y retorna el número que codifica la imagen modificada.

Hay distintos formalismos que permiten definir funciones $f : \mathbb{N} \rightarrow \mathbb{N}$ computables, es decir, que corresponden a algoritmos. Algunos de ellos son las máquinas de Turing, el cálculo lambda y la teoría de las funciones recursivas. Notablemente, aunque todos son formalismos muy distintos, siempre dan lugar a exactamente el mismo conjunto de funciones computables. Este conjunto de funciones computables no es todo $\mathbb{N}^{\mathbb{N}}$. De hecho, el conjunto de las funciones computables es numerable, por lo que la mayoría de las funciones no lo son.

Por último, está la pregunta de si la definición matemática de función computable corresponde con las funciones que en el mundo físico se pueden calcular con una computadora. La Tesis de Church-Turing conjetura que efectivamente es el caso. Al ser algo del mundo físico y salirse de la matemática, no hay ninguna prueba formal, pero en general es asumido como cierto.

1.2. Arquitectura de Von Neumann

La arquitectura de Von Neumann es un modelo para computadoras físicas, a diferencia de formalismos como las máquinas de Turing, que son conceptos puramente teóricos.

Primero sería bueno decidir qué consideramos como una computadora. Digamos que es un dispositivo físico que puede automáticamente calcular funciones computables y que tiene mecanismos para interactuar con el entorno. Podemos pensar que es una caja negra que de alguna forma calcula funciones computables y además tiene algún mecanismo por el cual podemos ingresar información (por ejemplo con un teclado) y algún mecanismo por el cual puede devolver información (por ejemplo con una pantalla).

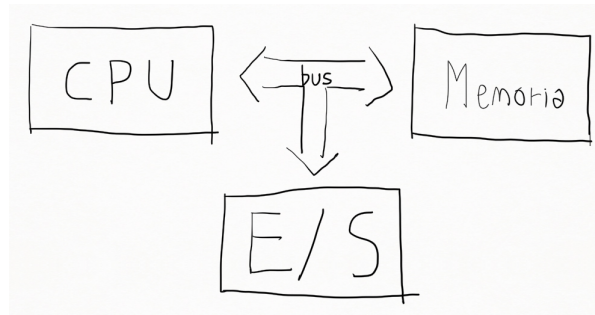


Figura 1.1: Esquema de los componentes de la arquitectura de Von Neumann.

La arquitectura de Von Neumann define que la computadora tiene tres componentes principales que interactúan entre sí: un **procesador** (CPU), una **memoria** y un subsistema de **entrada y salida** (figura 1.1). A la conexión entre las distintas componentes se le llama **bus**. El procesador es un dispositivo con la capacidad de ejecutar un conjunto finito de operaciones básicas, llamadas instrucciones. Un programa está dado por una secuencia (finita) de instrucciones —la idea es que aunque las instrucciones sean finitas, al usarlas de forma secuencial se pueda implementar cualquier función computable. Tanto la secuencia de instrucciones que conforma un programa como los datos con los que este opera se guardan en la memoria. El subsistema de entrada y salida permite que la computadora interactúe con el entorno (por ejemplo con un usuario que ingresa datos y observa resultados o con otra computadora dentro de una red). Se le llama **hardware** a los componentes físicos de la computadora y **software** a los programas.

Todas las componentes de la computadora se construyen con circuitos electrónicos y usando la abstracción del *bit*, que significa dígito binario, es decir 0 o 1. Claramente el bit es un concepto abstracto que no existe de por sí en los circuitos eléctricos. Lo que se hace es tomar una convención, como que si circula corriente significa 1 y si no significa 0, o que cierto voltaje significa 1 y cierto voltaje significa 0. Sea cual sea la convención, lo importante es que se pueden construir circuitos con entrada y salida que implementen cualquier operación finita. Por ejemplo, se puede armar un circuito con 4 bits de entrada, a_0, a_1, b_0, b_1 y tres bits de salida c_0, c_1, c_2 tal que si tomamos la entrada como la codificación en binario de dos números entre 0 y 3, la salida sea la codificación en binario de la suma. En general, para cada función $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ se puede construir un circuito con n bits de entrada y m bits de salida que implementa la función f . A estos se les llama *circuitos combinatorios*. Junto con otro tipo de circuitos llamados *secuenciales* (que funcionan por etapas y además de la entrada actual dependen de las entradas anteriores, o de un estado que puede ir variando), son un elemento base de las computadoras, tal vez el principal.

El procesador tiene tres componentes que cumplen distintas funciones: la unidad de control, la unidad aritmética lógica (ALU) y el banco de registros. La unidad aritmética lógica consta de varios circuitos combinatorios que implementan operaciones básicas, como por ejemplo operaciones lógicas como «y», «o» y «no», operaciones aritméticas como suma y multiplicación (para números codificado con una cantidad fija de bits) u operaciones básicas sobre cadenas e bits, como el *shift*, que traslada cada bit un lugar a la derecha o la izquierda. El banco de registros consta de ciertos registros (un tipo de circuito que almacena valores) en general variables que se usan para el funcionamiento de la CPU. Por ejemplo, pueden tener los argumentos de las operaciones que va a realizar

la unidad aritmética lógica. También hay un registro de particular importancia llamado *program counter*, que indica la posición de memoria de la siguiente instrucción a ejecutar (es de suma importancia, ya que la idea del CPU es ejecutar secuencias de instrucciones). Finalmente, la unidad de control es un circuito secuencial que se encarga de llevar a cabo las instrucciones, usando la unidad aritmética lógica y el banco de registros. Estos circuitos secuenciales están hechos para que automáticamente se lea e identifique la instrucción que indica el program counter, se carguen nuevos datos en los registros (si es necesario), se realice la operación indicada en la ALU y se guarde el resultado donde corresponda (si hay que hacer alguna operación), y que el program counter avance a la siguiente instrucción, para poder repetir el proceso.

Respecto a la suma, la ALU al ser un circuito combinatorio, es en realidad como una tabla que tiene el resultado para cada combinación de operandos. Las sumas que se realizan directamente por la ALU pueden ser por ejemplo de números representados por uno o dos bytes (donde un byte es por definición una secuencia de 8 bits). Como la cantidad de números representables crece exponencialmente con la cantidad de bits, hacer sumas de números más grandes por la ALU (que lo que hace es tener una tabla con todos los resultados) es extremadamente ineficiente. Lo que se hace para realizar sumas con números más grandes es descomponerlas como una secuencia de instrucciones más sencillas. Esto es como lo que se hace en los algoritmos aritméticos básicos. Pensemos por ejemplo en el producto. Está la tabla de multiplicación que uno se aprende de memoria (lo cual es análogo a lo que está en los circuitos de la ALU) pero luego a partir de eso, siguiendo un algoritmo (secuencia de instrucciones) se puede calcular la multiplicación de números de cualquier cantidad de cifras sin precisar saber nada más de memoria.

La memoria es un dispositivo electrónico capaz de almacenar muchos valores, los cuales pueden ser leídos o modificados. Tiene la estructura de un gran vector de bytes, indizado por secuencias de bits de largo fijo. Por ejemplo, si se usan dos bits para indizar, los lugares de la memoria serían el 00, el 01, el 10 y el 11, cada uno de los cuales contendría un byte de información. En las computadoras actuales la memoria suele tener del orden de 2^{32} lugares de memoria, es decir 4GB (2^{10} bytes es un KB, 2^{20} es un MB y 2^{30} es un GB). Se trata de la llamada memoria RAM. Actualmente la cantidad de bits que se usa para indizar es 64 (la llamada arquitectura de 64 bits), pero eso no implica que todas las palabras que se puedan formar correspondan a direcciones válidas (en computadoras personales no existe actualmente memoria de 2^{64} bytes, que serían como 16 mil millones de gigabytes). Conceptualmente el banco de registros del procesador es como una memoria. El motivo de que existan las dos cosas es que el banco de registros es memoria de mejor calidad y más cara que se reserva para estar dentro de la CPU, mientras que la memoria RAM es más lenta pero más económica para tener en grandes cantidades.

1.2.1. Ejemplo de programa

Hagamos un ejemplo del funcionamiento de una computadora. Supongamos que tenemos 8 registros (enumerados) de un byte cada uno y que las direcciones de memoria son indizadas también por un byte. Supongamos que tenemos las siguientes instrucciones

- **LOAD.** Instrucción con dos parámetros: M y R, que lo que hace es cargar en el registro R el contenido de la dirección de memoria M.

- **SAVE.** Instrucción con dos parámetros: M y R , que lo que hace es guardar en la dirección de memoria M el contenido del registro R .
- **ADD.** Instrucción con tres parámetros: R_1 , R_2 y R_3 que lo que hace es sumar el contenido de los registros R_1 y R_2 y guardar el resultado en el registro R_3 .

Respecto a la operación **ADD**, como los contenidos de los registros son bytes, o sea 8 bits, se considera que cada secuencia de 8 bits representa en binario un número entre 0 y $2^8 - 1$. En caso de que el resultado se salga del rango, se le sustraye 2^8 al resultado (formalmente, es la suma de \mathbb{Z}_{2^8} , es decir aritmética modular).

Por otra parte, cada instrucción está codificada por un byte. Supongamos que el código de **LOAD** es 00000001, el de **SAVE** es 00000010 y el de **ADD** es 00000011.

Asumamos también que los tres primeros registros, R_1 , R_2 y R_3 se identifican con los bytes 00000001, 00000010 y 00000011.

Supongamos que queremos un programa que lo que haga es sumar los contenidos de las direcciones de memoria 10000001 y 10000010 y guardar el resultado en la dirección 10000011. Podríamos hacerlo realizando cuatro instrucciones. Primero un **LOAD** para cargar el contenido de la primera dirección de memoria en el registro 1, luego otro **LOAD** para cargar el contenido de la segunda dirección de memoria en el registro 2, luego un **ADD** para sumar los contenidos de los registros 1 y 2 y guardar el resultado en el registro 3 y finalmente una **SAVE** para guardar el contenido del registro 3 en la tercera dirección de memoria. Primero escribamos las instrucciones de un modo un poco más abstracto (para no ir de una a los 0s y 1s). Definamos las direcciones de memoria 10000001, 10000010 y 10000011 como M_1 , M_2 y M_3 , respectivamente. El programa sería lo siguiente.

```
LOAD  M1 R1
LOAD  M2 R2
ADD   R1 R2 R3
SAVE  M3 R3
```

Esto último es lo que se considera como un programa escrito en *assembler*. Es una forma de programar en la que cada línea es una instrucción del procesador, pero se pueden usar los nombres de las instrucciones y nombres de lugares de memoria, en lugar de tener que escribir todo en bits. El mismo programa escrito ahora en código máquina, es decir en bits, exactamente como iría en la memoria, sería:

```
00000001 10000001 00000001
00000001 10000010 00000010
00000011 00000001 00000010 00000011
00000010 10000011 00000011
```

La organización en líneas del código es para que sea más fácil de leer, pero el programa de hecho es simplemente 000000011000000100000000100000001100000100000000100000000110000010000000010000000101000001100000011. Si tenemos en alguna parte de la memoria esa secuencia de bytes y el program counter indicando la dirección donde comienza, la computadora automáticamente lo ejecutará. Esto se ilustra en la figura 1.2.

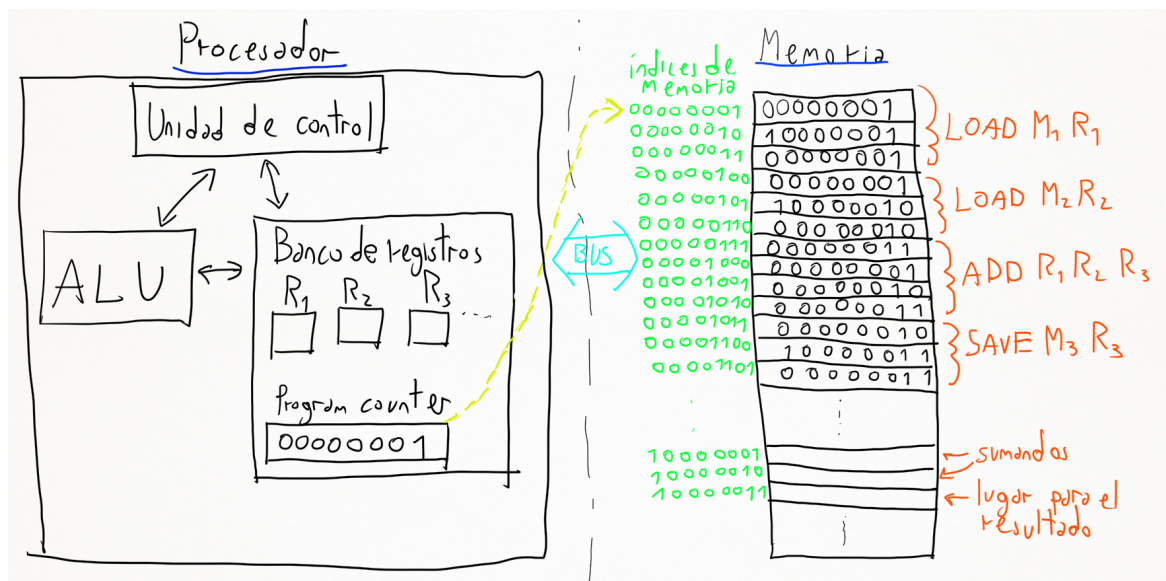


Figura 1.2: Esquema de una computadora con un programa.

Cabe resaltar que este es un ejemplo muy sencillo. Hay otros tipos de instrucciones (por ejemplo las que modifican el program counter y permiten realizar ciclos) y los programas suelen ser mucho más largos. De hecho, cuáles son exactamente las instrucciones y cómo se codifican varía de un procesador a otro.

1.3. Lenguajes de programación y compiladores

Dada una computadora que sigue la arquitectura de Von Neumann, se puede implementar cualquier función computable con algún programa escrito en código máquina. Sin embargo, hay algunas características poco prácticas.

- Es relativamente difícil escribir programas como secuencias de instrucciones de máquina. En general hay muchos más detalles que entran en juego que en el ejemplo dado y hace falta mucho esfuerzo para escribir programas que hagan operaciones complejas.
- Distintos procesadores suelen tener distintas instrucciones. Además, incluso cuando la instrucción es la misma puede estar codificada distinto. Por ejemplo, puede que en un procesador 00000001 sea el código de la suma y en otro sea el código de la operación de cargar en un registro. Por lo tanto, un programa que se escribe para un procesador probablemente no funcione en otro.

Estos factores dificultan escribir programas y hacen que solamente lo puedan hacer especialistas. Sin embargo, con el tiempo se desarrolló un concepto que mejora la situación, haciendo que sea más fácil desarrollar un programa y que la curva de aprendizaje sea más corta. Se trata de los **lenguajes de programación**.

Los lenguajes de programación introducen abstracciones que permiten escribir programas sin preocuparse por detalles técnicos de cómo se resuelve con instrucciones del

procesador. Por ejemplo, se introduce el concepto de las variables para trabajar con datos sin preocuparse por en qué direcciones de memoria están ni de cómo se utilizan los registros para operaciones. Por ejemplo, se si tenemos tres variables x , y y z , para sumar los contenidos de las primeras dos y guardarlo en la tercera, simplemente habría que escribir:

$$z = x + y$$

sin preocuparse ni por en qué dirección va cada una ni de si hay que usar registros para realizar la suma o no.

Un programa escrito en un lenguaje de programación es un archivo de texto. Se compone por líneas escritas con caracteres comunes que representan operaciones de forma clara (para una persona). Las instrucciones escritas en un lenguaje de programación no pueden ser ejecutadas directamente por un procesador (estos solo entienden ceros y unos). En este punto entran en juego los **compiladores**. Estos son programas cuya función es traducir un programa escrito en un lenguaje de programación a un programa en código máquina. Se los puede pensar como funciones cuya entrada es un programa escrito en un lenguaje de programación y su salida es una secuencia de instrucciones de procesador (lenguaje máquina) que hace lo mismo. El proceso usual al programar es el siguiente: primero se escribe el programa en un lenguaje de programación, luego se usa un compilador para traducirlo a código de máquina y finalmente se lo ejecuta.

Los lenguajes de programación junto con los compiladores solucionan las dos desventajas previamente mencionadas. Por una parte, escribir en un lenguaje de programación es más fácil que en lenguaje máquina. Por otra parte, un lenguaje de programación suele tener compiladores para distintos procesadores, de modo que un mismo programa escrito en este lenguaje puede ser compilado a código máquina de distintos procesadores y por lo tanto ser ejecutado en cada uno de ellos.

Se dice que un lenguaje de programación es de alto o bajo nivel según el nivel de abstracción que hay por sobre el lenguaje de máquina (instrucciones del procesador). Escribir directamente las instrucciones del procesador o código en assembler es de bajo nivel, mientras que usar un lenguaje de programación que incluye abstracciones como variables, estructuras de datos o iteraciones es de alto nivel. Entre los distintos lenguajes de programación hay varios niveles de abstracción.

El lenguaje `c` (así como `pascal`) es un lenguaje de relativamente bajo nivel, pues tiene abstracciones como las previamente mencionadas, pero no está *tan* lejos del lenguaje máquina. Mirando un programa escrito en `c`, para alguien con experiencia no es tan difícil darse cuenta de cómo podrían ser las instrucciones en lenguaje máquina.

Por otra parte, el lenguaje `python`, que es el que usaremos en el curso, es de muy alto nivel. Esto significa que tiene abstracciones muy complejas por sobre el lenguaje máquina, las cuales hacen que se pueda programar en base a razonamientos intuitivos sin saber cómo se haría en lenguaje de máquina. De hecho puede ocurrir que una sola línea de código en `python` corresponda a cientos de instrucciones del procesador, con lo que se resuelven automáticamente detalles técnicos y se le ahorra trabajo al programador. En general usar un lenguaje de programación de alto nivel es más fácil que usar uno de bajo nivel. La desventaja que tienen es que las abstracciones pueden hacer que el programa no sea lo más eficiente posible. En general un programa escrito en `c` es mucho más rápido que uno equivalente en `python` (pero mucho más difícil de escribir).

Capítulo 2

Programación en python

En este capítulo se presentan los conceptos básicos para programar en el lenguaje python. Si bien se trata de un lenguaje específico, los conceptos se aplican de forma muy similar en otros.

Como material complementario, se recomienda el tutorial de la documentación oficial de python: <https://docs.python.org/es/3.13/tutorial/index.html>. Ese tutorial contiene mucho más que lo que se necesita para el curso. Se recomienda ver todo el capítulo 3 y algunos conceptos del 4, pero sin preocuparse por detalles técnicos (hay explicaciones pensadas para gente que ya conoce conceptos de programación por otros lenguajes).

Cabe aclarar que lo que se ve en este curso es solo una pequeña introducción a python. Además, las explicaciones buscan transmitir ideas del lenguaje, sin necesariamente ser totalmente correcto con las definiciones formales de los conceptos o exactamente cómo funciona por dentro. Para esto se recomienda nuevamente la documentación (reiterando que excede los contenidos del curso).

2.1. Instalación y ejecución

Instalar python es principalmente instalar el compilador que traduce el código en python a lenguaje máquina, de modo que podamos ejecutar lo que programemos. Un detalle técnico es que en lugar de llamársele «compilador» se le llama «intérprete». Esto es porque en lugar de pasar todo el programa a lenguaje máquina y luego ejecutarlo, va pasando las líneas de código a lenguaje máquina una por una a la vez que se van ejecutando. Se dice que es un compilador cuando primero se traduce todo a lenguaje máquina y luego se lo ejecuta.

En el sistema operativo linux, en general python ya viene instalado por defecto. Por otra parte, en windows no es el caso (nos enfocaremos en este sistema operativo, ya que quienes utilizan linux es más probable que ya sepan ejecutar python). Una forma de instalarlo es descargando el instalador desde el sitio web de python y ejecutándolo (<https://www.python.org/downloads/>). Durante la instalación es importante marcar la opción de «Add python.exe to PATH» (figura 2.1), por motivos que veremos a continuación.

Vamos a ejecutar python desde una ventana de comandos (o terminal). Esto es una ventana a través de la cual se le escriben comandos al sistema operativo. En windows se llama Windows PowerShell o símbolo del sistema.

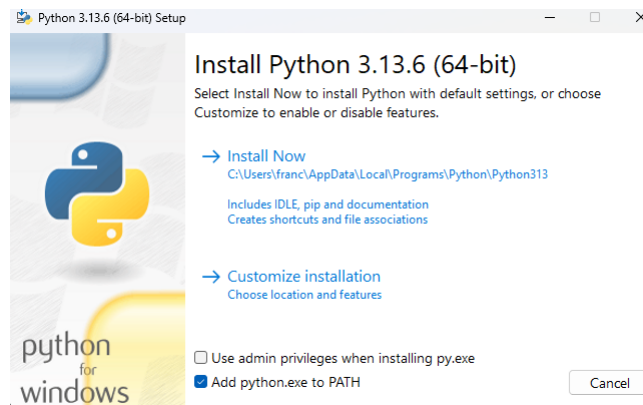


Figura 2.1: Es importante marcar la opción «Add python.exe to PATH».

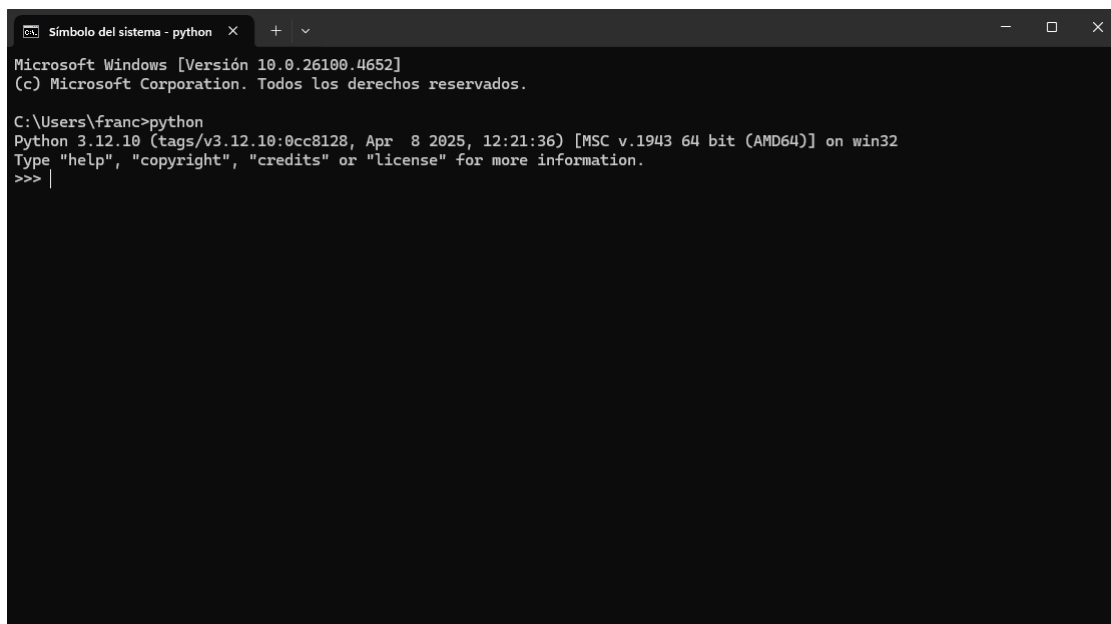


Figura 2.2: Ventana de comandos con el intérprete de python.

Si hicimos bien la instalación de python, al abrir una ventana de comandos, si se escribe python y se toca enter, comienza a ejecutarse el intérprete, el cual nos permite ir ingresando instrucciones una a una. Los tres símbolos de > significan que el intérprete está listo para recibir instrucciones. Una primera cosa que se puede hacer es usarlo como calculadora, escribiendo expresiones aritméticas y tocando enter (figura 2.2). Para salir del intérprete se puede escribir `quit()` y tocar enter.

Al escribir python, lo que se hace es ejecutar el intérprete. Para que esto funcione, el sistema operativo debe tener la dirección en la que python está instalado dentro de una cierta lista. Esto es lo que se hace al marcar la opción «Add python.exe to PATH» durante la instalación. Luego de haber ejecutado python en la ventana de comandos, el uso del intérprete ocurre dentro de esta misma ventana, sin usarse ninguna interfaz gráfica (a diferencia de la mayoría de los programas que utilizamos). En general no hay necesidad de nada más sofisticado.

Con la modalidad de uso del intérprete que vimos, se escriben y ejecutan líneas una a

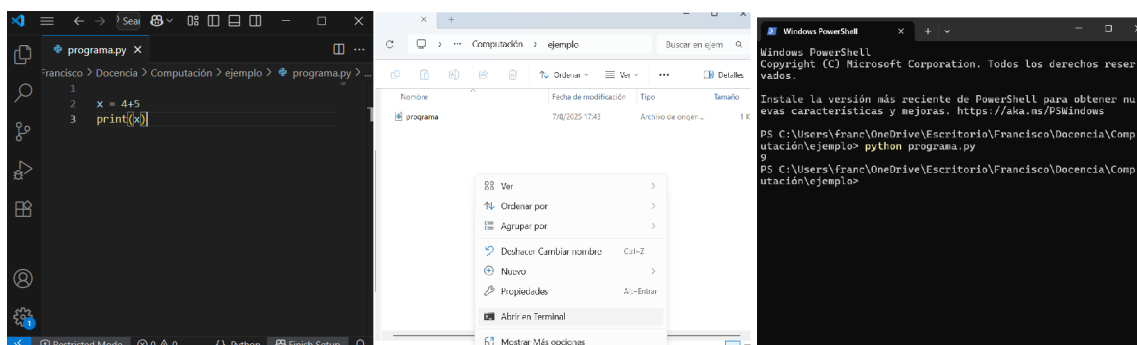


Figura 2.3: A la izquierda visual studio code con el código del archivo llamado programa.py. Al centro se ve la carpeta llamada ejemplo en la que está el archivo. Dentro de esa carpeta se abre la terminal. A la derecha se ve la terminal en la que se escribe «python programa.py». Antes del símbolo >, a la izquierda de «python», se indica la carpeta en la que está la terminal. Si al hacer click derecho en la carpeta no aparece la opción «abrir en terminal», puede aparecer al hacer click derecho manteniendo apretado mayus/shift (recordar que powershell también es una terminal). En todo caso, para problemas como este se recomienda buscar en internet, con lo cual se suele encontrar alguna solución rápidamente. También se puede abrir una terminal en cualquier sitio y cambiar la ubicación con instrucciones específicas, como cd (buscar en internet en caso de que haya interés).

una. Por otra parte, se puede escribir un archivo con varias líneas de código y pasárselo al intérprete para que lo ejecute. El código se escribe en un archivo de formato «.py». Para escribir este tipo de archivos se puede usar cualquier editor de texto, pero se recomienda utilizar un editor de código especializado. Uno muy popular actualmente es visual studio code (<https://code.visualstudio.com/>). Una vez que se escribió el archivo con el código y se lo tiene guardado en una carpeta, hay que abrir una ventana de comandos en esa carpeta (cada ventana de comandos trabaja en alguna carpeta particular) e ingresar «python» seguido del nombre del archivo con el «.py» incluido. Esto se ejemplifica en la figura 2.3.

Para todo lo que mencionamos en esta sección, buscando en internet (motores de búsqueda como google, o sitios con contenido audiovisual como youtube) se encuentran muchos tutoriales.

2.2. Tipos de datos

Una de las abstracciones que aportan los lenguajes de programación consiste en los tipos de datos. Se trata de la abstracción de distintos tipos de objetos, como enteros, texto o listas, cada uno de los cuales tiene distintos posibles valores y distintas operaciones que se pueden aplicar. Por supuesto que todos estos objetos se codifican como secuencias de bits, pero poder trabajar pensando en términos de objetos de distinto tipo hace que programar sea más intuitivo. Se puede saber el tipo de datos de un objeto escribiendo `type()` con el objeto entre paréntesis. Veamos algunos de los tipos de datos que tiene python con algunas de las operaciones disponibles.

```

>>> type(True)
<class 'bool'>
>>> not True
False
>>> not False
True
>>> True and False
False
>>> True or False
True
>>> True and (True or False)
True
>>> False or (True and False)
False
>>> False or not (True and False)
True

```

Figura 2.4: Ejemplos de operaciones con booleanos.

```

>>> 3 + 4 #suma
7
>>> 3 - 4 #resta
-1
>>> 3 * 4 #producto
12
>>> 3 ** 3 #exponencial
27
>>> 7 // 2 #cociente de división entera
3
>>> 7 % 2 #resto de división entera
1
>>> |

```

Figura 2.5: Operaciones aritméticas básicas. Lo que está escrito después de los # son comentarios. Los comentarios son ignorados por el intérprete, la función que cumplen es de aclaraciones para personas que lo leen.

2.2.1. Booleanos

Los booleanos conforman un tipo de datos que representa valores de verdad. Tiene solo dos objetos: `True` y `False`. Se tienen las operaciones lógicas básicas: `not`, `and` y `or`. Hay algunos ejemplos en la figura 2.4. Si aplicamos `type()` a un booleano, retorna `class bool`. En general los booleanos por sí solos no suelen ser de mucha utilidad. Su principal aplicación es en condicionales e iteraciones, que estudiaremos en lo siguiente.

2.2.2. Enteros

Se trata de un tipo de datos para representar enteros naturales. Los objetos son 0, 1 -1, 2, -2, etc. Si aplicamos `type()` a un entero, retorna `class int` por *integer*. Tienen las operaciones aritméticas elementales, que se muestran en la figura 2.5. Notar que la división entera es con dos barras. Una barra sola es otra operación que veremos en la brevedad. Por otra parte se tienen las operaciones de comparación, que reciben dos enteros y retornan un booleano. Se muestran en la figura 2.6. En el caso de la igualdad, si se escribe un solo símbolo «=» es una asignación, lo cual veremos más adelante.

2.2.3. Números de punto flotante

Se trata de una representación para operar con números reales usando información finita. De hecho lo que representan es un conjunto finito de números racionales, pero

```

>>> 3 == 3 #igualdad. Un "=" solo es otra cosa
True
>>> 3 == 4
False
>>> 3 != 3 #no igualdad
False
>>> 3 < 4 #desigualdad estricta
True
>>> 3 <= 3 #desigualdad amplia (menor o igual)
True
>>> 4 > 3
True
>>> 3 >= 3
True

```

Figura 2.6: Comparaciones entre enteros.

a menos de cierta aproximación se opera con ellos igual que con números reales. Estos números se escriben con la notación usual para números con parte fraccionaria o con notación científica, en ambos casos separando la parte fraccionaria con un punto. Están las mismas operaciones aritméticas y comparaciones, salvo por la división. En punto flotante tenemos el operador `/` para la división de números reales. Un detalle importante es que si aplicamos la operación `/` a enteros, el resultado es de punto flotante, independientemente de que la división de exacta o no. Se presentan ejemplos en la figura 2.7.

Los números de punto flotante tienen limitaciones debido al hecho de que se pueden representar solo finitos números. Por ejemplo, si a `1.0` le sumamos un número muy chico, se redondea a `1.0`, lo cual puede ser problemático en algunas situaciones. Ignoraremos este problema, pero es un tema relevante dentro del área de análisis numérico.

Hay más operaciones sobre punto flotante que se pueden realizar si importamos la biblioteca `math`. Más adelante en el curso nos centraremos más en las bibliotecas, pero por ahora alcanza con saber que es algo que permite extender el lenguaje con más operaciones. Para incluir el contenido de la biblioteca `math` hay que escribir la instrucción `import math`. Después de eso se pueden utilizar las operaciones incluidas en esta biblioteca. Esto se hace escribiendo el nombre de la biblioteca, un punto y el nombre de la operación. Por ejemplo, tiene una operación de raíz cuadrada que se utiliza escribiendo `math.sqrt()`, donde al argumento lo ponemos entre los paréntesis. Por otra parte, está la operación `math.floor()` que retorna el entero más cercano a la izquierda. Esta última función retorna efectivamente un objeto de tipo entero y no un objeto de tipo punto flotante con valor entero. Se presentan ejemplos en la figura 2.8.

En general se pueden realizar operaciones aritméticas entre enteros y números de punto flotante. En ese caso, el resultado es de punto flotante. Lo que ocurre en estos casos es que (al menos conceptualmente) se convierten automáticamente enteros a punto flotante. Al aplicar `/` a enteros ocurre lo mismo (al menos conceptualmente): primero los enteros se transforman a punto flotante y luego se aplica la operación. Por otra parte, en general no hay conversión automática de punto flotante a entero, aunque el número no tenga parte fraccionaria.

2.2.4. Texto

En python se puede representar texto como secuencias de caracteres, denominadas *strings*. Los strings se delimitan con comillas simples o dobles. Si aplicamos `type()` a un string, retorna `class str`. Los caracteres pueden ser por ejemplo letras, dígitos, signos de

```

>>> 1.0
1.0
>>> -4.325
-4.325
>>> 1.5 * -4.76
-7.14
>>> 1.45e3
1450.0
>>> -4.876e-3
-0.004876
>>> 1.001 < 1.001001
True
>>> 1.45**2.54
2.5696544467216724
>>> 4.5/1.5
3.0
>>> 5 / 5 #Notar que si ponemos enteros con esta operación el resultado es igual punto flotante
1.0
>>> 1.234e2 / -3.5452e-1
-348.0762721426154

```

Figura 2.7: Ejemplos de números de punto flotante.

```

>>> import math
>>> math.sqrt(123.0)
11.090536506409418
>>> math.floor(124.54) #Esta operación retorna el entero por debajo
124
>>> math.sqrt(123.0) + math.floor(124.54)
135.09053650640942

```

Figura 2.8: Ejemplos de operaciones de la biblioteca math.

puntuación, espacios o caracteres especiales para representar cosas como el salto de línea (pasar a la siguiente línea). Algunos ejemplos de strings son “Hola.”, “345”, “ ” (espacio) y “” (string vacío). No hay que confundir, por ejemplo, al string “345” con el entero 345, pues son objetos de distintos tipos (se lo puede verificar con `type()`). Con el símbolo `+` se realiza la operación de concatenación, es decir “Hola.” + “345” da “Hola.325”. Notar que el símbolo `+` tiene varios usos. Cual de todos se usa depende del tipo de los argumentos. Por ejemplo, `12 + 12` da 24 (los argumentos son enteros, por lo que se los suma), mientras que `“12” + “12”` da “1212” (los argumentos son strings, por lo que se los concatena). Si se ejecuta `12 + “12”` da un error por los tipos.

Por otra parte, se pueden realizar con strings las mismas comparaciones que con números. La igualdad `==` y la desigualdad `!=` determinan si se trata de exactamente el mismo string o no, mientras que las desigualdades usan orden lexicográfico.

2.2.5. Listas

Las listas conforman un tipo de datos compuesto. Cada lista es, como su nombre sugiere, una lista de objetos. Una lista se comienza y se termina con `«[»` y `«]»`. Los distintos elementos se separan con comas. Por ejemplo: `[0,1,4]`, `[“aa”,“a”,“”]`, `[3.5, “a”]` y `[]`. Las listas pueden tener cualquier cantidad de elementos (a partir de cero) y estos elementos pueden ser de cualquier tipo, incluso otras listas. Por ejemplo `[[],[]]` es una lista con dos elementos, que los dos son la lista vacía. Con `+` se puede concatenar listas, de modo que `[[],[]] + [1]` da `[[],[],1]`.

Hay una operación de pertenencia, para la que se usa la palabra `in`. La forma de aplicarlo es `«x in l»`, siendo `x` un objeto y `l` una lista. Esta operación retorna un booleano, el cual es `True` si `x` está en la lista `l` y `False` en otro caso. Por ejemplo, `3 in [1,2,3]`

da `True` mientras que `4 in [1,2,3]` da `False`. Para esta operación se piensa a la lista como un conjunto, en el sentido de que el resultado no se ve afectado por el orden de los elementos ni por si aparecen repetidos o no. A modo de ejemplo, `3 in [1,2,3]` da lo mismo que `3 in [3,2,1]` y que `3 in [3,3,2,1,1,3]`.

2.2.6. Transformación entre tipos

En python hay ciertas operaciones que convierten un objeto a cierto tipo. Por ejemplo tenemos `int()`, `float()` y `str()`, que permiten cambiar un objeto a entero, punto flotante o string, respectivamente.

La operación `int()` se puede aplicar a un número en punto flotante o a un string. En caso de aplicarse a un número en punto flotante, se lo trunca. En caso de aplicarse a un string, si este string representa un entero retorna el entero y en otro caso da un error.

La operación `float()` se puede aplicar a un número entero o a un string. En caso de aplicarse a un string, si este string representa un número en punto flotante, retorna este número y en otro caso da un error.

La operación `str()` convierte el objeto en cuestión a un string y en general se puede aplicar a cualquier tipo. Los siguientes son algunos ejemplos.

```
>>> int(1.0)
1
>>> int(1.1)
1
>>> int(0.9)
0
>>> int("34")
34
>>> int("-34")
-34
>>> int("-34d") #da error
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '-34d'
>>> float(4)
4.0
>>> float("-4.634")
-4.634
>>> str(6)
'6'
>>> str(6.845)
'6.845'
```

2.3. Variables, expresiones y asignaciones

Las **variables** son objetos sintácticos (es decir elementos del lenguaje) que consisten en un nombre, el cual se puede usar para referenciar distintos objetos de cualquier tipo

de datos. Diremos que el objeto referenciado es el valor de la variable. Es una abstracción para usar la memoria de modo intuitivo, en base a un nombre. Durante la ejecución de un programa, el objeto referenciado por una variable puede ir variando. Nombres válidos de variable son secuencias de caracteres alfanuméricos y «_» en los que el primer carácter no es un número, como por ejemplo `x`, `x1`, `total` y `altura_caja`. Hay ciertas palabras reservadas para usos especiales, las cuales no pueden ser usadas como variables, como por ejemplo `if`, `for` y `while`, cuyos usos veremos más adelante en este capítulo.

Las **expresiones** son objetos sintácticos que se pueden calcular, resultando en un objeto de algún tipo de datos (que puede ser cualquiera). Están definidas en base a los siguientes casos.

1. Los valores constantes (llamados *literales* en python) de cualquier tipo son expresiones válidas. Por ejemplo, `False`, `10`, `‘Hola’` y `[1,2,3]`. Estas expresiones al calcularse retornan el valor constante en cuestión.
2. Una variable es una expresión válida. Al calcularse retorna el valor actual de la variable.
3. Dada una operación, al aplicarla a ciertas expresiones, da lugar a una nueva expresión más compleja. Por ejemplo, `«3+7»`, `«3**(5-2)»`, `«x + 10»`, `«x*y + z/2»` y `«flag and (x<y)»`. Como en varios de estos ejemplos, se puede aplicar una operación a expresiones que ya tienen operaciones. Al calcular la expresión, se realiza la operación correspondiente. Si `x`, `y`, `z` tiene los valores 1, 2, 3 respectivamente y `flag` tiene el valor `True`, entonces las expresiones calculan 10, 27, 11, 3.5 y `True`, respectivamente.

Formalmente, dada una expresión definida por una operación entre otras dos expresiones `exp1 op exp2`, para calcularla primero se calculan `exp1` y `exp2` en objetos x_1 y x_2 y luego se realiza la operación `op` entre x_1 y x_2 .

4. Una función (elemento que se presenta la sección 2.8) aplicada a ciertos argumentos (que pueden ser otras expresiones) da lugar a una expresión más compleja. Para calcular la expresión se evalúa la función. Por ejemplo, si tenemos definida una función `f` con dos parámetros, entonces `f(4,9)`, `f(x,2*(x+y))` y `f(f(x,2),f(1,y))` son ejemplos de expresiones válidas.

Formalmente, dada una expresión del tipo `f(exp1, exp2, ..., expn)`, para calcularla primero se calculan las n expresiones de los argumentos, dando n objetos, luego se evalúa `f` con estos objetos y lo que retorna `f` es el resultado.

La **asignación** es la instrucción básica de python para dar un valor a una variable. La sintaxis es:

$$\text{var} = \text{expr}$$

donde `var` es el nombre de la variable a la que queremos asignar un objeto (es decir, hacer que la variable referencia al objeto, o que el objeto sea el valor de la variable) y `expr` es una expresión que calcula el objeto a asignar. Es distinto al uso que se le da al símbolo «=» en matemáticas (en particular, los lados tienen roles distintos), asemejándose más al de «:=».

Al ejecutarse una asignación se hacen las siguientes dos cosas.

1. Se calcula la expresión de la derecha.
2. Se asocia ese objeto a la variable de la izquierda

Si en la expresión hay alguna variable que no se ha definido antes o una operación aplicada a objetos de tipo incorrecto (por ejemplo “a” / [1,2]), se produce un error.

La expresión puede contener la misma variable **var**. En ese caso para evaluarla se usa el valor actual de la variable y luego se le asigna el resultado.

Los siguientes son algunos ejemplos. Lo que se escribe después del # es un **comentario**: está solo para aclarar algo a la persona que lee; python lo ignora.

```
>>> x = 10 #asignamos el valor 10 a la variable x
>>> x #Mostrar el valor actual de la variable x
10
>>> x*2
20
>>> x1 = x**2
>>> x1
100
>>> x = x + 1
>>> x
11
>>> z = 20
>>> z = z + x2 #Usamos una variable x2 que no definimos antes. Da error
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'x2' is not defined. Did you mean: 'x'?
>>> flag = True
>>> y = flag and x < 20
>>> y
True
>>> lista1 = ["a","b","c"]
>>> lista2 = [1,2,3]
>>> lista3 = lista1 + lista2
>>> lista3
['a', 'b', 'c', 1, 2, 3]
```

En python las variables no guardan un valor dentro de ellas, sino que en realidad referencian a un objeto que está fuera en otro lado. Esto es un detalle técnico al que volveremos en la sección 2.4.1.

2.4. Indizado y slicing

En listas y en strings cada lugar tiene un índice, que es un entero no negativo. El primer lugar es el de índice 0, el segundo es el de índice 1 y así sucesivamente. Notar que **el primer lugar tiene índice 0**. Por ejemplo dada la lista [5,6,7], elemento de índice 0 es 5, el de índice 1 es 6 y el de índice 2 es 7. No hay elementos de índices mayores a 2.

Análogamente, dado un string “Hola”, el elemento de índice 0 es “H”, el de índice 1 es “o”, el de índice 2 es “l” y el de índice 3 es “a”. No hay elementos de índice mayor a 3.

El **indizado** es una operación para acceder al elemento de un índice dado en una lista o un string. La forma de escribirlo es `x[n]`, donde `x` es la lista o string y `n` es el índice. Por ejemplo, si `x` es `[5,6,7]`, entonces `x[0]` da 5, `x[1]` da 6 y `x[2]` da 7, mientras que si `y` es “Hola”, entonces `y[0]` da “H”, `y[1]` da “o”, `y[2]` da “l”, y `y[3]` da “a”.

Si tenemos una lista de números y le aplicamos indizado, el objeto resultante es un número. Por otra parte, cuando aplicamos indizado a un string, el objeto resultante no es de otro tipo, sino que también es un string, solo que de largo 1, ya que contiene solo el caracter del índice dado.

Tanto para listas como para strings, si el índice se pasa del último elemento, se produce un error. En python se puede usar la función `len()` para determinar el largo de una lista o un string. Por ejemplo, `len([5,6,7])` da 3 y `len(“Hola”)` da 4. Los índices válidos de una lista o string `x`, van desde 0 hasta `len(x)-1`.

Por otra parte, el **slicing** es una operación para obtener una sublista o un substring a partir de un rango de índices. En python los rangos se escriben con el símbolo «:» y se toma la convención de que son con **orden inclusivo a la izquierda y estricto a la derecha**. Es decir, dados dos enteros no negativos $n < m$, tenemos que $n : m$ representa el intervalo cerrado a izquierda y abierto a derecha $[n, m)$. Dada una lista o string `x` y dos enteros no negativos $n < m$, la operación de slicing se escribe `x[n : m]` y resulta en la sublista o substring con los objetos de los índices pertenecientes al rango dado. Equivalentemente, da la sublista o substring de todos los elementos con índice i tal que $n \leq i < m$. No hay problemas si nos pasamos del último elemento. El resultado puede ser la lista vacía o el string vacío.

El resultado de el slicing es una lista nueva, cuyos índices están dados por los lugares como con cualquier otra lista, sin tener por qué ser los mismos que en la lista anterior. Por ejemplo, si `x` es la lista `[5,6,7]`, el slicing `x[1:3]` retorna la lista con los elementos de índices 1 y 2, que es `[6,7]`, pero los índices de esta nueva lista son 0, para el primer lugar y 1 para el segundo. El 6 pasó a tener índice 0 y el 7 a tener índice 1. Hay que recordar que los índices se definen por los lugares y no por los objetos que hayan.

En una lista se pueden hacer asignaciones a elementos de cierto índice, modificando el objeto de ese lugar. Por ejemplo, si `x` es la lista `[5,6,7]`, la asignación `x[1]=3` hace que pase a ser `[5,3,7]`. Con strings no se permite hacer esto.

Hasta ahora venimos escribiendo `x[n]` o `x[n:m]`, donde `x` es una variable y `n` y `m` son números específicos, pero cabe aclarar que en el lugar de `n` y `m` se pueden poner variables con valores numéricos, o en general cualquier expresión que calcule un número. De hecho, en el lugar de `x` se puede también poner cualquier expresión que calcule una lista o un string. Por otra parte, como el indizado y el slicing son operaciones, pueden formar parte de expresiones, como por ejemplo `x[n+1] - m`.

Veamos algunos ejemplos.

```
>>> x = [1,2,3,4,5]
>>> s = "Hola 1."
>>> x[0]
1
>>> s[1]
'o'
```

```

>>> s[4] #Va a retornar un espacio
,
>>> len(s)
7
>>> s[6]
','
>>> s[7]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> x[5]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> x[0:2]
[1, 2]
>>> x[2:5]
[3, 4, 5]
>>> x[2:10]
[3, 4, 5]
>>> x[6:10]
[]
>>> x[1]=40
>>> x
[1, 40, 3, 4, 5]
>>> x[3*5-11]
5
>>> n = 3
>>> x[n]
4
>>> x[n-1] + 2
5

```

Con listas de listas se puede concatenar indizados. Es decir, si `l` es una lista cuyos elementos son listas, `l[i][j]` es el *j*ésimo elemento de la *i*ésima lista. Los paréntesis implícitos van de la siguiente forma: `(l[i])[j]`, pues `l[i]` es una lista a la cual se aplica el índice *j*. A modo de ejemplo:

```

>>> l = [[1,2],[3,4],[5,6]]
>>> len(l) #Tiene 3 elementos que son las listas de adentro
3
>>> l[0] # Retorna la primera lista
[1, 2]
>>> l[1] # Retorna la segunda lista
[3, 4]
>>> l[2] # Retorna la tercera lista
[5, 6]

```

```

>>> l[3] # Produzcamos un error de índice
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> len(l[0]) #Largo de la lista [1,2]
2
>>> l[0][0] #Elemento 0 de lista 0
1
>>> l[0][1] #Elemento 1 de lista 0
2
>>> l[1][0] #Elemento 0 de lista 1
3

```

En general de hecho, en caso de que `l[i]` sea una lista, se puede hacer con ella lo mismo que con una lista que esté dada por una variable. Por ejemplo, también se puede hacer slicing con `l[i][n:m]`, lo cual dará la sublista de `l[i]` correspondiente, o determinar su largo con `len(l[i])`. Por otra parte, también se puede asignar `l[i]` a una variable nueva y luego trabajar con ella. Por ejemplo, hacer `y = l[i]` y luego `y[j]`, `y[n:m]` o `len(y)`, dando iguales resultados que `l[i][j]`, `l[i][n:m]` y `len(l[i])`, respectivamente.

2.4.1. Referencias múltiples

Vamos a tratar un detalle un poco técnico. La mayoría de las veces no es relevante, pero a veces puede ser fuente de errores inesperados.

Recordar que las variables no contienen los objetos, sino que los referencian. Esto es particularmente importante con listas porque estas se pueden modificar, entonces si dos variables referencian la misma lista y desde una la modificamos, esto se reflejará en la otra variable. Veamos un ejemplo.

```

>>> l1 = [1,2,3]
>>> l2 = l1
>>> l2[0] = 0
>>> l1
[0, 2, 3]

```

Esto ocurrió porque la asignación `l2=l1` lo que hizo es que la variable `l2` referencie la misma lista que `l1`. Para que la lista de `l2` sea una lista distinta a la de `l1` pero con los mismos valores, habría que escribir `l2 = l1.copy()`.

Con números una asignación de una variable a otra también hace que referencie al mismo objeto, pero no surgen problemas. Por ejemplo:

```

>>> x=3
>>> y = x
>>> y = 0
>>> x
3

```

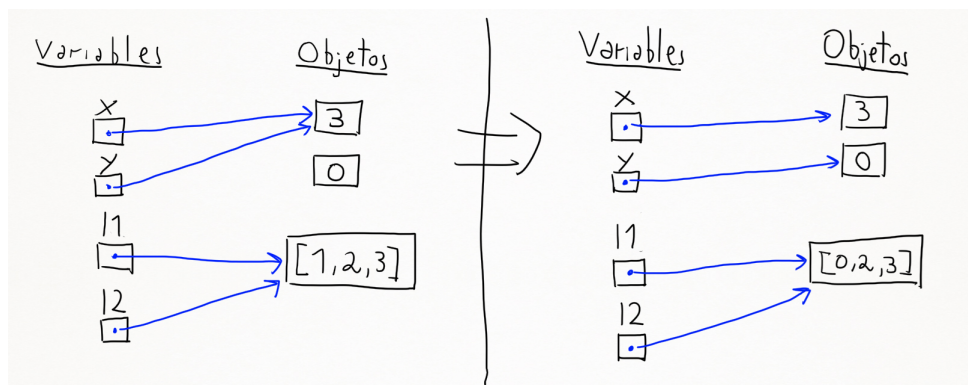


Figura 2.9: Ilustración de ejemplo presentado en la sección 2.4.1

La primera instrucción liga la variable `x` al objeto `3`. La segunda instrucción liga la variable `y` al mismo objeto. Sin embargo, la tercera instrucción lo que hace es ligar la variable `y` a un nuevo objeto `0`, sin alterar el objeto `3` preexistente.

De hecho los dos ejemplos que presentamos son muy distintos. La asignación `y=0` modifica a qué está asignada la variable `y`; se podría decir que modifica la variable. Por otra parte, la operación `l2[0]=0` modifica la lista referenciado por la variable `l2`, sin cambiar el hecho de que esta variable está ligada a esa lista; se podría decir que modifica al objeto ligado a la variable y no a la variable. Ver la figura 2.9.

Los números son objetos **inmutables**: no se pueden modificar. Las listas, por otra parte, son objetos **mutables**, lo cual significa que pueden ser modificadas. Un ejemplo es cuando se modifica uno de sus elementos, como al ejecutar `l2[0]=0`. Esta distinción de mutable o inmutable se aplica a los objetos de python en general. Los strings por ejemplo también son inmutables.

2.5. Programas e instrucciones

Digamos que programa en python es una secuencia de instrucciones que se almacena en un archivo de formato `«.py»`. Un ejemplo básico de instrucción es la asignación, que ya vimos. Por otra parte, cuando escribimos una expresión o una variable en el intérprete y damos enter para que imprima el valor, también se trata de una instrucción. Estas últimas en general se usan solamente haciendo experimentos sencillos en el intérprete; no es común incluirlas en programas.

Al ejecutar un programa (ver figura 2.3) se ejecutan las instrucciones en el orden que están escritas, es decir, de la primera línea en adelante, o desde arriba hacia abajo. No obstante, veremos que hay instrucciones de tipo condicional y de iteraciones, que adentro permiten hacer bifurcaciones (en base a alguna condición, lo que se ejecuta puede cambiar) o ciclos en la ejecución, respectivamente.

Digamos que las instrucciones pueden ser simples o compuestas. No es una distinción general, pero la introduzco porque me parece pedagógicamente útil en este punto. Las instrucciones simples son las que se escriben en una línea y representan una acción específica, mientras que las compuestas son instrucciones con cierta estructura que contienen adentro otras instrucciones. Las asignaciones son instrucciones simples y en esta sección

presentamos algunas otras. En secciones posteriores presentaremos algunas instrucciones compuestas que son los condicionales y las iteraciones: `if`, `for` y `while`.

Además de asignaciones, utilizaremos las siguientes instrucciones simples.

- `print()`. Se trata de una instrucción que imprime en la consola lo que se le pasa como parámetro. Es la instrucción por excelencia para mostrar algo en la consola. Si se le pasa un valor directamente imprime ese valor, mientras que si se le pasa una variable imprime el contenido de la variable. Por ejemplo, la instrucción `print("Hola")` imprime «Hola», mientras que `print(x)` imprime el valor de la variable `x` (y si esta variable no tiene ningún valor, da error).
- `append()`. Esta instrucción es para agregar un elemento al final de una lista. A diferencia de las anteriores, es formalmente lo que se llama un *método* de las listas. Dada una lista `l` y un objeto `x` que queremos agregar al final, la instrucción es `l.append(x)`. Se escribe el nombre de la lista, un punto y luego `append()` con lo que queramos agregar adentro de los paréntesis.
- `insert()` Permite agregar un elemento en un lugar particular de una lista. Al igual que `append`, es un método. Dada una lista `l`, un índice `i` y un objeto `x`, la instrucción `l.insert(i,x)` agrega al objeto `x` en la posición de índice `i` de la lista, moviendo a todos los elementos posteriores (en caso de que los haya) un índice adelante (por ejemplo, el `i`-ésimo pasa a ser el nuevo `(i+1)`-ésimo). A modo de ejemplo, si `l = [5,6,7]`, entonces si hacemos `l.insert(1,0)` pasa a ser `l = [5,0,6,7]`.
- `del`. Permite (entre otras cosas) quitar el elemento de cierto índice de una lista. Dada una lista `l`, la instrucción `del l[i]` quita el elemento de índice `i`. Si hay elementos después del `i`-ésimo, sus índices se decrementan (por ejemplo, el `i+1` pasa a ser el nuevo `i`). A modo de ejemplo, si `l = [5,6,7]`, entonces si hacemos `del l[1]` pasa a ser `l = [5,7]`.

Además de estas instrucciones hay algunas operaciones (me refiero a instrucciones que más que algo, retornan un valor que se puede usar para algo más) que junto con asignaciones forman instrucciones útiles, en particular para obtener datos del usuario. Son las siguientes.

- `input()`. Se trata de una operación que espera a que el usuario escriba algo en la terminal y luego retorna eso como un string. Es una herramienta rudimentaria para hacer programas interactivos. Lo que se escribe entre paréntesis es un texto que se le muestra al usuario antes de esperar a que este escriba algo. Una forma de uso común es `x=input()`. De este modo, lo que escriba el usuario se guarda como un string en la variable `x`. Al llegar a una instrucción de este tipo, la ejecución del programa se detiene hasta que el usuario escriba algo y presione enter.
- `int()` y `float()`. Estas operaciones (ya presentadas en la sección 2.2.6), respectivamente, al aplicarse a un string `x` que representa un entero o un número de punto flotante, retornan este entero o número en punto flotante. Por ejemplo, `int("-1")` retorna el entero `-1`. En general usaremos estas funciones junto con el comando `input()`, como en el ejemplo que viene a continuación, para convertir el string que

ingresa el usuario a un número. En caso de que el string no represente un objeto válido del tipo de dato, como por ejemplo al ejecutar `int("123a")`, se produce un error.

En todas las operaciones e instrucciones anteriores que llevan algo dentro de paréntesis, lo que se ingresa puede ser un objeto constante o una expresión. De hecho en este sentido son como las funciones que veremos en la sección 2.8.

Existen muchas otras instrucciones en python, pero no es el foco del curso aprender una gran cantidad. En general hay muchas que hacen cosas que con un poco de trabajo se pueden realizar utilizando otras.

Además de las instrucciones, se pueden agregar comentarios, los cuales no afectan el funcionamiento del programa, sino que están para quienes leen el código. El compilador, al traducir el código a instrucciones de máquina, ignora totalmente los comentarios. En python se escriben usando el símbolo `«#»`. Todo lo que hay después de ese símbolo en esa línea es un comentario; al pasar a la siguiente línea el comentario se termina y en caso de querer que siga hay que ingresar otro `«#»`. Los comentarios en general se usan para dejar aclaraciones de lo que hace el código, de modo que si alguien más lo lee (o uno mismo después de un tiempo) sea más fácil entenderlo.

También se pueden dejar líneas en blanco, lo cual no afecta el comportamiento del programa pero puede ayudar a que el código quede más fácil de leer.

Veamos un ejemplo de programa en el que se usan las herramientas de entrada/salida (`input` y `print`) presentadas.

```
print("Hola.")
x = input("Ingrese su numero favorito por favor: ")
print("Su numero favorito es " + x) #Notar que x es un string.
```

```
xNum = int(x) #Con esta instruccion, el string x se convierte en entero.
print("El doble de su numero favorito es", 2*xNum)
#Se puede imprimir mas de una cosa separando con comas.
```

Estas líneas son el contenido de un archivo llamado `«ejemplo1.py»`. Al ejecutarlo en la terminal e ingresar `«5»` al momento del `input`, ocurre lo siguiente:

```
PS C:\Users...\Computación\Programas\ejemplo1> python ejemplo1.py
Hola.
Ingrese su número favorito por favor: 5
Su número favorito es 5
El doble de su número favorito es 10
```

La ejecución del programa se detuvo después de mostrar `«Ingrese su número favorito por favor: »`. Luego escribí `«5»`, presioné enter y ahí continuó. Si escribiera algo que no sea un entero válido, la operación `int(x)` fallaría. En ese caso se imprime en la terminal una explicación del error y la ejecución se corta en ese punto.

Recomiendo ejecutar este ejemplo por uno mismo. Alcanza con copiar y pegar el código escrito arriba en un archivo y ejecutarlo.

Veamos algunos ejemplos de las instrucciones de listas mencionadas. Notar en particular como los índices de los elementos cambian si se insertan o eliminan elementos en la mitad de la lista.

```

>>> l = [10,20,30]
>>> l[2] # el elemento de índice 2 es 30
30
>>> l.append(40) # Agrega 4 al final de la lista
>>> l
[10, 20, 30, 40]
>>> l.insert(2,9) # Inserta 9 en el lugar de índice 2 (tercero)
>>> l
[10, 20, 9, 30, 40]
>>> l[3] #Notar que ahora el elemento de índice 3 es 30
30
>>> del l[1] #Elimina el elemento de índice 1
>>> l
[10, 9, 30, 40]
>>> l[2] # 30 volvió a ser el elemento de índice 2
30

```

2.5.1. Más sobre print

La instrucción `print` puede tener varios argumentos separados por comas, los cuales se imprimen uno tras otro. Es decir, si escribimos `print(x_1, x_2, \dots, x_n)` se imprimen x_1 , luego x_2 y así sucesivamente hasta x_n .

Después de imprimir los argumentos en pantalla, la instrucción `print` realiza un salto de línea. Esto hace que si escribimos otro `print` después, el resultado esté en la línea siguiente. Esto se puede cambiar usando `print` con un parámetro particular que es «`end=`», el cual indica lo que se pone después imprimir los argumentos `print`. Lo que se pasa a este parámetro debe ser un string, como por ejemplo “,” o “ ” (espacio) o “” (string vacío). Si escribimos `print(4, end=“,”)`, se imprime un 4 en la terminal y luego en vez de hacer un salto de línea se pone una coma. Si luego imprimimos algo más con otro `print`, irá en la misma línea después de la coma. A modo de ejemplo,

```

print(4, end=",")
print(5, end=" ")
print("seis")
print("Programar es divertido.")

```

produce la siguiente salida:

```

PS C:\Users\...\ejemplo> python programa.py
4,5 seis
Programar es divertido.

```

Por otra parte, `print(“”)` es una forma sencilla de hacer un salto de línea sin imprimir nada más. Esto es porque el argumento es el string vacío, pero de todos modos se realiza el salto de línea.

2.6. Condicionales

Con lo que tenemos hasta ahora, la secuencia de instrucciones que se ejecuta en un programa es algo estático. Independientemente de lo que ocurra, las instrucciones se ejecutan de la primera hasta la última. Las instrucciones condicionales, por otra parte, permiten escribir código que se ejecute o no en función de cierta condición. La sintaxis (es decir, la estructura de cómo se escribe) es la siguiente:

```
if cond:
    code
```

Se escribe la palabra clave «if», luego se escribe una condición, que es cualquier expresión que se evalúe en un booleano, luego se escribe un símbolo «:» y abajo, con una sangría de 4 espacios, se escribe el código que debe ejecutarse en caso de que la condición se cumpla, el cual se denomina *cuerpo* del if. Al ejecutarse el if ocurre lo siguiente.

1. Se evalúa la expresión que conforma la condición, llegándose a un valor booleano.
2. En caso de que el resultado de evaluar la condición haya sido **True**, se ejecuta el cuerpo. En caso de que haya sido **False**, no se lo ejecuta.

El cuerpo del if es una secuencia de una o más instrucciones. No hay restricciones respecto al tipo de estas instrucciones, pudiendo ser simples o compuestas. Es decir, en el cuerpo del if pueden haber otros if o iteraciones (las cuales veremos en la siguiente sección).

La sangría es muy importante, pues delimita el cuerpo. Ejemplifiquemos esto. En el siguiente código:

```
if cond:
    instr1
instr2
instr3
```

el cuerpo del if es solamente **instr1**. En caso de que se cumpla **cond**, se ejecuta **instr1**. Después de eso, independientemente de la condición, se ejecutan **instr2** e **instr3**. Por otra parte, consideremos ahora:

```
if cond:
    instr1
    instr2
instr3
```

Ahora el cuerpo del if consta de **instr1** y **instr2**. En este caso, si se cumple **cond**, entonces se ejecutan **instr1** e **instr2**. Después de eso, independientemente de la condición se ejecuta **instr3**.

La instrucción **if** se puede extender con más cosas. Estas extensiones no nos permiten hacer nada que no pudiéramos hacer solamente con ifs, pero ayudan a escribir código más claro. Por ejemplo, se puede agregar una cláusula **else** para algo que se ejecute en caso de que no se cumpla la condición. Con esto la instrucción queda:

```

if cond:
    instr1
else:
    instr2

```

De esta forma, si se cumple `cond` se ejecuta `instr1` y si no, se ejecuta `instr2`. Es equivalente a:

```

if cond:
    instr1
if not cond:
    instr2

```

La otra extensión común es con `elif`. Esto es como un `else if`. Es para poner algo que se haga en caso de que la condición anterior no se cumpla pero otra condición sí se cumpla. Después de un `if` se pueden poner tantos `elif` como se desee. Por ejemplo, podemos escribir

```

if cond1:
    instr1
elif cond2:
    instr2
elif cond3:
    instr3
else:
    instr4

```

que es equivalente a:

```

if cond1:
    instr1
else:
    if cond2:
        instr2
    else:
        if cond3:
            instr3
        else:
            instr4

```

Notar que en este caso tenemos instrucciones condicionales dentro del cuerpo de otras.

A modo de ejemplo, podemos escribir el siguiente programa que pide un entero al usuario y le dice si este entero es negativo, cero o positivo.

```

x = int(input("Ingrese un entero "))
#De una leemos lo que escribe el usuario y lo convertimos en un entero
if x < 0:
    print("El número es negativo")
elif x == 0:

```

```

    print("El número es cero")
else:
    print("El número es positivo")
print("Gracias.") #Esto siempre se ejecuta al final.

```

Ejercicio 2.6.1. Escribir la siguiente instrucción usando cuatro ifs seguidos (sin meter un if adentro del cuerpo de otro).

```

if cond1:
    instr1
elif cond2:
    instr2
elif cond3:
    instr3
else:
    instr4

```

2.7. Iteraciones

Los distintos tipos de iteraciones son tal vez la herramienta más fuerte en la programación. Una computadora tiene la capacidad de ejecutar millones de instrucciones por segundo, pero escribir millones de instrucciones no es viable. La forma de aprovechar esta capacidad es con los ciclos, o iteraciones, que nos permiten decir a la computadora que repita algo muchas veces. De esta forma, con una cantidad razonable de líneas de código podemos hacer que la computadora ejecute muchas instrucciones. Veremos dos tipos de iteraciones: el **for** y el **while**.

El **for** (en python) nos permite iterar en orden a través de los elementos de una lista y hacer algo con cada uno de ellos. La sintaxis es:

```

for x in lista:
    code

```

donde **lista** es la lista a recorrer, **x** es una variable nueva que irá recorriendo los elementos de esta lista y **code** (el cuerpo) es el código que se ejecuta para cada elemento de la lista. Al igual que con los **if**, el cuerpo es una secuencia de instrucciones que pueden ser de cualquier tipo, incluso **if**, otros **for** o **while** (que veremos a continuación). Nuevamente la sangría del código es importante para delimitar las instrucciones que se iteran. Dentro del código se puede usar la variable **x**, cuyo contenido va variando por los elementos de la lista de modo ordenado. El **for** termina luego de haber ejecutado el cuerpo una vez para cada elemento de la lista.

Si el contenido de la lista es $[a_1, a_2, \dots, a_n]$, entonces lo que ocurre es lo siguiente. Primero **x** toma el valor a_1 y se ejecuta **code**. Después **x** toma el valor a_2 y se vuelve a ejecutar **code**. Sigue así sucesivamente hasta que **x** toma el valor a_n y se ejecuta **code** por última vez.

A modo de ejemplo:

```

lista = [1,2,3,4,5]
for x in lista:

```

```

    print(x)
print("Fin.")

```

imprime los elementos de la lista en orden y luego un «Fin.».

Una funcionalidad de python que se suele usar conjuntamente es la función `range()`. Se puede pensar que `range(n)` es la lista de enteros que comienza en 0 y termina en $n-1$. Internamente se representa distinto, pero conceptualmente es eso. También se lo puede pensar como el intervalo $[0, n)$. Con dos parámetros se puede elegir también el comienzo. Los elementos de `range(a, b)` son los i tales que $a \leq i < b$, es decir $[a, b)$. En general en python los intervalos son cerrados a izquierda y abiertos a derecha (lo mismo pasa con el slicing `l[a:b]`). Escribamos un programa que sume todos los números menores a cierto n , es decir que calcule $\sum_{i=0}^{n-1} i$.

```

n = int(input("Elija el n: "))
suma = 0 #Variable "acumuladora" para la suma
for i in range(n): #range(n) es [0,1,2,...,n-1]
    suma = suma + i #En cada paso agregamos el número i a la suma
print(suma)

```

Este código ilustra dos usos distintos de variables que se dan muy comúnmente en iteraciones. Suelen haber algunas que van iterando entre distintos valores, como en este caso `i` y otras que se usan para ir calculando algo de forma acumulativa, como en este caso `suma`. Es muy común que en este sentido hayan variables *iteradoras* y *acumuladoras*.

Ejercicio 2.7.1. Escribir un programa que pida dos entradas a y b y retorne la suma de todos los números n tales que $a \leq n \leq b$. Notar que queremos incluir a b en la suma. Luego de escribir el programa hacer algunas pruebas para ver si anda bien (una buena práctica que se suele llamar *testing*). Recomendando en general hacer pruebas, independientemente de que se pida en la letra o no.

Se puede usar un `for` para recorrer un string, por ejemplo iterando en los índices válidos. Supongamos que dado un string `s` queremos contar la cantidad de veces que aparece el carácter «a». Lo hacemos creando una variable para llevar la cuenta `cont` y usaremos un `for` para iterar por los índices válidos de `s`, sumando uno a la variable contadora (`cont`) cada vez que encontremos una «a». Como los índices válidos de `s` son desde 0 hasta `len(s)-1`, podemos recorrerlos con un `for i in range(len(s))`, donde en cada ciclo podemos usar `s[i]` para acceder carácter i -ésimo. El código es el siguiente.

```

s = "Hola. Hola, hola" # por ejemplo
cont = 0
for i in range(len(s)):
    if s[i]=="a": # nos fijamos si el caracter i-ésimo es "a"
        cont = cont + 1
print(cont)

```

Veamos un ejemplo en el que se usa un `for` adentro de otro. Ahora queremos un programa dada una lista de listas, imprima los elementos de modo que cada lista de adentro quede en una línea separada con los elementos separados por espacios. Es decir, si la lista de listas es `[[1,2],[3,4]]`, lo que queremos es que en la primera línea se imprima 1 2 y en la segunda se imprima 3 4.

```

lista = [[1,2,3],[4,5],[6,7,8,9]] # Lista de listas
for listita in lista:
    # listita es una de las listas pertenecientes a lista
    # Hay que imprimir los elementos de listita en una línea
    # y luego hacer un salto de línea para la siguiente
    for x in listita: # Iteramos en los elementos de listita
        print(x, end=" ") # El end=" " es para que queden en la
                           # misma línea separados por espacios
                           # queda un espacio de más después del
                           # último, pero como no se ve, lo dejamos
    print("") # Esto es para hacer un salto de línea después de
              # haber imprimido todos los elem de listita

```

La salida del programa es la siguiente:

```

PS C:\Users\...\ejemplo> python programa.py
1 2 3
4 5
6 7 8 9

```

Pasamos ahora a los ciclos **while**. En este caso, el ciclo está definido por una condición booleana y se repite el contenido hasta que esta condición sea falsa. La sintaxis es:

```

while cond:
    code

```

donde **cond** es una expresión que se evalúa a un valor booleano (igual que en los **if**) y **code** (el cuerpo) es lo que queremos que se ejecute mientras la condición se cumpla. Nuevamente, es una secuencia de instrucciones que pueden ser de cualquier tipo, incluso **if**, **for** u otros **while**.

La ejecución es como sigue. Primero se evalúa la condición. Si esta es falsa, termina la ejecución del **while**. Si es verdadera, se ejecuta **code**, luego se vuelve a evaluar la condición y se vuelve a tomar la misma disyuntiva, así sucesivamente hasta que la condición sea falsa.

La iteración solo se detiene si la condición es falsa. En caso de que nunca se haga falsa, la ejecución entra en un bucle infinito.

El **while** es una herramienta muy poderosa, porque nos permite hacer iteraciones que a priori no sabemos la cantidad de veces que se van a ejecutar (en un **for** es el largo de la lista), pero introduce el riesgo de que nuestro programa no termine nunca. En la computación, esto es algo con lo que hay que convivir, prestando atención al escribir los programas.

Veamos un ejemplo que determina si un número $n \geq 2$ es primo o no. El razonamiento es el siguiente. Vamos a ir recorriendo números menores a n para ver si encontramos un divisor. La idea es ir probando con números sucesivos hasta que encontremos un divisor o lleguemos a n .

```

n = int(input("Introduzca un entero mayor a 1: "))
puede_ser_primo = True
potencial_divisor = 2

```

```

while puede_ser_primo and potencial_divisor < n:
    if n % potencial_divisor == 0: #Es un divisor de n
        puede_ser_primo = False
        potencial_divisor = potencial_divisor + 1
if puede_ser_primo:
    print("Es primo")
else:
    print("No es primo")

```

Podría hacerse también con un `for`, haciendo que se recorra todo `range(2,n)` y para cada elemento se determine si es un divisor o no. La ventaja de hacerlo con el `while` es que una vez que encontramos un divisor, dejamos de iterar. Si por otra parte usamos el `for`, aunque en seguida encontráramos un divisor y ya sepamos que no es primo, seguiríamos iterando hasta llegar a n .

Ejercicio 2.7.2. Escribir un programa para determinar si un número es primo que use `for` (independientemente de la desventaja que acabamos de mencionar).

Los `while` también se pueden usar para hacer iteraciones en rangos de números del mismo modo que se hace con `for` y `range`. Esto se puede hacer de la siguiente forma.

```

i = 0
while i < n:
    algo...
    i = i + 1

```

lo cual es equivalente a:

```

for i in range(n):
    algo...

```

La versión con el `while` en realidad es mucho más versátil. Por ejemplo, si queremos que `i` solamente itere por números pares, alcanza sustituir la última línea por `i = i + 2`. También podemos variar más la forma de ir actualizando la variable, como en el siguiente ejemplo.

Supongamos que dado un string `s` que representa un texto formado solamente por letras y espacio, queremos determinar la cantidad de palabras. En el texto pueden haber más de un espacio entre una palabra y la siguiente y además pueden haber espacios antes de la primera palabra y después de la última.

La idea es ir recorriendo el texto hasta que encontramos una letra. En ese momento incrementamos un contador y avanzamos hasta que esa palabra se termine, lo cual puede ocurrir porque lleguemos a un espacio o porque se termine el string. Después repetimos el procedimiento hasta que se termina el string. Como después de encontrar una letra queremos avanzar hasta el final de la palabra, una iteración con `for` que va avanzando siempre de a uno no nos sirve, pero sí una iteración con `while`.

```

s = "  Hola   aquí   hay alguna   cantidad de   palabras   "

cont = 0

```

```

i = 0
while i < len(s): #iteramos mientras no se termina el string
    if s[i] == " ": # estamos en un espacio
        i = i + 1 # simplemente avanzamos al siguiente lugar
    else:
        cont = cont + 1
        j = i + 1 # usaremos esta variable para determinar dónde
                # termina la palabra
        while j < len(s) and s[j] != " ": # la palabra todavía no terminó
            j = j + 1
        i = j # en este punto j es el primer espacio después de la
                # palabra o es el final de la lista. Continuamos ahí
print(cont)

```

Hay una aclaración a hacer sobre la línea `while j <len(s) and s[j] != " "`. En python cuando se calcula un `and`, no siempre se evalúan ambos lados. Esto se llama **evaluación de circuito corto**. Lo que ocurre es que primero se evalúa el lado izquierdo y solo en caso de que este de `True`, luego se evalúa el derecho. El motivo es que si el lado izquierdo da `False`, sabemos que el resultado del `and` será `False`, independientemente del valor del otro lado. Esta característica es importante para que este código funcione bien. Notar que si `j` vale `len(1)`, entonces si intentamos evaluar `s[j]` ocurre un error de índice, pues el último índice válido es `len(1)-1`. Debido a la evaluación de circuito corto, cuando `j` vale `len(1)`, como la parte izquierda `j <len(1)` da `False`, la parte derecha no se evalúa y por lo tanto se evita el error.

Otro uso de los `while` es el de hacer un programa que pueda repetirse tantas veces como el usuario quiera. Para esto lo que podemos hacer es:

```

continuar = True
while continuar:
    programa... #Aquí está el programa que debe repetirse
                #hasta que el usuario decida lo contrario
    respuesta = input("Si desea repetir ingrese 1, sino ingrese 0: ")
    if respuesta == 0:
        continuar = False

```

Ejercicio 2.7.3. Aplicar esto último con alguno de los programas que determinan si un número es primo.

2.8. Funciones

En programación se le llama función a un fragmento de código separado del resto que puede recibir parámetros y retornar valores. Proviene del concepto matemático de función. No permiten hacer nada que no se pudiera hacer con lo que ya vimos, pero son útiles para que los programas queden más fáciles de entender y para reutilizar código. Esto último es porque una función se define una sola vez y se puede utilizar cualquier cantidad de veces como parte de uno o varios programas. La sintaxis es la siguiente:

```
def nombre_fun(p_1, ..., p_n):
    code
```

Se escribe la palabra reservada **def**, luego el nombre de la función, luego una secuencia de parámetros (cero o más), que son nombres de variables, luego un «:» y luego cierto código con sangría, el cual es denominado *cuerpo* de la función. En el cuerpo se puede usar la instrucción especial **return** para que la función retorne un valor. La idea es que en términos matemáticos es una función que se aplica a los parámetros y su imagen es lo que se indica con **return**.

La sintaxis de la instrucción **return** es **return expr**, donde **expr** es una expresión que indica lo que se debe retornar. Al ejecutarse esa instrucción se evalúa la expresión y la función termina, retornando el valor calculado, independientemente de si luego haya más código o no.

Luego de que una función ha sido definida, se la puede llamar asignando distintos valores a los parámetros. Para evaluar una función se escribe el nombre de la función con una expresión para cada parámetro: **nombre_fun(expr1, ..., exprn)**. Al ejecutarse esto, primero se evalúan todas las expresiones y luego se ejecuta la función asignando a los parámetros los resultados de las evaluaciones, en el orden correspondiente. La evaluación de una función es en sí una expresión que se puede usar por ejemplo en asignaciones o para crear expresiones más complejas.

A modo de ejemplo, la función valor absoluto se puede escribir de la siguiente forma:

```
def val_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

Si en un programa ponemos solamente la definición de la función y no escribimos nada más después, ese programa no calcula nada. Por otra parte, después de la definición podemos ejecutarla en un argumento x escribiendo **val_abs(x)**.

```
def val_abs(x):
    if x >= 0:
        return x
    else:
        return -x
valor1 = val_abs(0)
valor2 = val_abs(3+3)
valor3 = val_abs(-46)
print(valor1, valor2, valor3)
```

Si se ejecuta este programa lo que se ve en consola es que imprime «0 6 46». Notar que escribimos la función una vez y pudimos ejecutarla para tres argumentos distintos; en esto radica la utilidad de las funciones.

Si una función tiene varios parámetros, al ejecutarla el orden de los argumentos es el mismo. Por ejemplo, si escribimos una función **f(x,y)** y luego ejecutamos **f(1,2)**, entonces **x** será 1 y **y** será 2.

Hagamos un ejemplo un poco más complejo vinculado con números primos. A partir del código que ya hicimos en la sección de iteraciones podemos escribir una función que reciba como parámetro un número y retorne un booleano que diga si es primo o no. Por otra parte, esta vez en lugar de buscar divisores hasta $n - 1$, buscaremos hasta \sqrt{n} . Es un ejercicio de aritmética verificar que con eso alcanza.

```
def es_primo(n):
    if n < 2:
        return False
    puede_ser_primo = True
    potencial_divisor = 2
    while puede_ser_primo and potencial_divisor <= n**(1/2):
        if n % potencial_divisor == 0: #Es un divisor de n
            puede_ser_primo = False
        potencial_divisor = potencial_divisor + 1
    return puede_ser_primo
```

Recordar que cuando se ejecuta un `return` la función termina, independientemente de que después haya más código o no. Veamos varios ejemplos de usos para esta función. Primero, para hacer lo mismo que antes alcanza con después escribir:

```
n = int(input("Introduzca un entero mayor a 1: "))
if es_primo(n):
    print("Es primo")
else:
    print("No es primo")
```

Por otra parte, podemos hacer una función que pida dos números e imprima todos los primos comprendidos entre ellos.

```
min = int(input("Introduzca el extremo inferior: "))
max = int(input("Introduzca el extremo superior: "))
for n in range(min, max+1): #Incluyo max
    if es_primo(n):
        print(n, end=",") #Lo de end es para que queden en la misma línea
                           #separados por comas
```

Por último escribamos un programa que dado un n nos diga cual es el primer primo mayor a n . Como los primos son infinitos, sabemos que el programa termina para cualquier entrada.

```
n = int(input("Introduzca un entero: "))
while not es_primo(n):
    n = n + 1
print("El mínimo primo mayor o igual es", n)
```

Notar como dato interesante que la terminación de este programa para cada entrada es de hecho equivalente a la existencia de infinitos primos. La cuestión de si un programa termina o no puede un problema difícil. No siempre se puede resolver empíricamente, sino

que en general hay que hacer una demostración. El motivo de esto es que si lleva mucho tiempo sin terminar, no tenemos forma de saber si es que no va a terminar nunca o que sí va a terminar pero más adelante.

El valor que retorna la función puede ser de cualquier tipo de datos, incluyendo listas. Esto es útil, por ejemplo, cuando queremos hacer una función que retorne más de una cosa. Por ejemplo, podríamos querer escribir una función que determine si un número n mayor a 1 es primo y que en caso de que no lo sea nos de dos números mayores a 1 que multiplicados nos den n . Esto lo podríamos hacer con una función que a partir de un n retorne una lista cuyo primer elemento sea un booleano que indica si es primo o no y que en caso de que no lo sea, tenga otros dos elementos que sean números mayores a 1 que multiplicados den n . Un ejemplo de tal función es la siguiente.

```
def primo(n):
    if n<2:
        return [False,0,0] #Este caso no interesa. Es para n > 1
    puede_ser_primo = True
    potencial_divisor = 2
    while puede_ser_primo and potencial_divisor <= n**(1/2):
        if n % potencial_divisor == 0: #Es un divisor de n
            puede_ser_primo = False
        else: #Para que si encontramos un divisor no se cambie
            potencial_divisor = potencial_divisor + 1
    if puede_ser_primo:
        return [True]
    else:
        return [False, potencial_divisor, n//potencial_divisor]
```

Podemos probar la función ingresándola en un archivo, digamos `programa.py` y luego ejecutándolo de la terminal con la *flag* `-i`, es decir `python -i programa.py`. Esto hace que luego de ejecutarse el código (lo cual deja la función definida), pasemos al modo interactivo del intérprete.

```
PS C:\Users\...\ejemplo> python -i programa.py
>>> primo(2)
[True]
>>> primo(3)
[True]
>>> primo(4)
[False, 2, 2]
>>> primo(5)
[True]
>>> primo(6)
[False, 2, 3]
>>> primo(12)
[False, 2, 6]
>>> primo(13)
[True]
```

```
>>> primo(14)
[False, 2, 7]
```

Por otra parte, en el código podemos llamar (evaluar) la función, guardar la lista resultante en una variable y luego acceder a sus elementos.

```
n = int(input("Ingrese un número: "))
resultado = primo(n)
if resultado[0]: # resultado[0] indica si es primo o no
    print("Es primo")
else:
    print("No es primo. Producto de:", resultado[1], resultado[2])
```

Una función puede no tener parámetros y también puede no retornar nada. En los casos en los que no retorna nada (simplemente hace ciertas cosas), lo cual se aparta mucho del concepto matemático de función, en algunos lenguajes se les llama procedimientos. En python se les llama igual funciones porque en realidad retornan un valor trivial que no se muestra (detalle formal). Un ejemplo de función que no tiene parámetros ni retorna nada sería:

```
def bienvenida():
    t = input("Hola, ¿cómo te llamas? ")
    print("Bienvenid@", t)
```

Se trata de una función que le da la bienvenida al usuario. Dentro de un programa podría ejecutarse al principio, poniendo «`bienvenida()`» como primera línea y no afectaría en nada para el resto del programa.

Una función puede llamarse a ella misma dentro de su cuerpo. En este caso es lo que se llama una **función recursiva**. Esto será profundizado en el capítulo siguiente.

2.8.1. Variables locales y globales

Veamos un par de detalles técnicos sobre nombres de variables y funciones. Primero, las variables que se definen dentro de una función se llaman **locales** y solo están definidas dentro del cuerpo de la función. Esto es para que el funcionamiento interno de la función sea bastante independiente al resto del código. En general la idea es que la función interactúa con el resto del programa solamente (o principalmente) mediante los parámetros y el valor que retorna. Veamos un ejemplo.

```
def f(n):
    x = 10
    return n + x
y = f(3)
print(x) #Esto da error porque x solo está definida adentro de la función
```

Por otra parte, las variables que se definen antes de la definición de la función pueden usarse dentro de esta y también siguen definidas después. A estas variables se les llama **globales**. Por ejemplo:

```

x = 10
def f(n):
    return n + x #Usamos la variable x definida antes
y = f(3)
print(x) #Ahora acá no da error

```

Sin embargo, este comportamiento se da si solamente leemos el contenido de la variable sin modificarla. En caso de que hagamos una asignación a `x`, no la realizará con la variable global, sino que creará una variable local con el mismo nombre que dentro de la función reemplazará a la variable global. Por lo tanto, los cambios que se hagan no se reflejarán afuera. Por ejemplo:

```

x = 5 #Variable global
def f(n):
    x= 10 #Asigna 10 a una nueva variable local sin modificar la global
    return n + x #Acá x es la variable local, no la global
y = f(3) #el valor de f(3) es 13
print(x) #Imprime 5, que es el valor de la variable global

```

Por último, una función que se define antes que otra puede usarse en la segunda. El vínculo de esto con lo que venimos viendo es que el nombre de la primera función es como una variable global para la segunda.

```

def f1(n):
    return n + 5
def f2(x):
    return 5*f1(x)

```

2.9. Metodología de resolución de problemas

Cuando tenemos un problema complejo no suele ser una buena idea ir directamente a programar. De hecho, en general el desarrollo de software tiene muchas etapas, en varias de las cuales no se programa. Veamos una metodología sencilla, la cual se recomienda usar para resolver problemas en el curso. De hecho, de modo más abstracto puede ser útil para resolver problemas en general, no necesariamente de programación. La metodología consta de cuatro etapas: análisis, diseño, implementación y verificación.

- **Análisis.** Se trata de estudiar el problema a resolver, asegurándose de entender qué es exactamente lo que se debe hacer. Es importante entender los conceptos y definiciones pertinentes al contexto donde está el problema. Esta etapa puede ser de las más complejas en aplicaciones de software en la vida real. Imaginarse por un momento hacer un programa para ser usado en medicina; primero habría que entender qué cosas se hacen exactamente, junto con mucha terminología específica del rubro. En problemas como los que veremos en el curso no suele hacer falta mucho trabajo en esta etapa, pero si no se hace correctamente (o sea, si no se entiende bien lo que se debe hacer), independientemente del esfuerzo que se haga después, el resultado no será correcto.

- **Diseño.** Se trata de diseñar una solución al problema, pensando a grandes rasgos la estrategia que se utilizará, sin meterse aún a programar nada. En general la estrategia incluye descomponer al problema como una secuencia de subproblemas más fáciles.
- **Implementación.** Se programa la solución al problema, siguiendo la estrategia que se definió en la etapa de diseño.
- **Verificación.** Es común que se use la palabra en inglés: *testing*. Se hacen pruebas para verificar si la solución tiene el comportamiento esperado. No se trata de una demostración formal de que el programa funciona, sino que de experimentos puntuales que aseguran solamente que el programa funciona en ciertos casos. La verificación no es infalible, pero si se hace bien, hay buenas chances de que en caso de que haya un error lo detectemos, así que es recomendable hacerlo.

Claramente el análisis y el diseño deben ser lo primero que se hace. Por otra parte, cuando la estrategia tiene varias etapas, es recomendable ir intercalando implementación y verificación, es decir, escribir una parte del programa, hacer pruebas para ver si está bien (en caso de que hayan errores corregirlos) y recién después de tener cierta certeza de esto, pasar a la siguiente etapa.

Para la verificación se pueden hacer casos concretos ingresados a mano o un programa que chequee automáticamente muchos casos para los que sabemos la respuesta. En el ejemplo que viene ahora haremos ambas cosas (y hay un ejercicio de esto en el práctico 1, titulado *testing automático*).

2.9.1. Ejemplo

Veamos un ejemplo (bastante artificial), para el cual aplicaremos la metodología. Supongamos que un cliente nos pide un programa que resuelve el siguiente problema.

Sea 1 una lista de enteros positivos. Determinar si la cantidad de números $n \in 1$ tales que su cantidad de dígitos también está en 1 es un número primo.

Análisis. Hay que asegurarnos de entender bien lo que se pide. En caso de que algo no esté claro, habría que consultar al cliente (salvo que sea evidente que sea irrelevante). Después de haber leído bien la consigna, puede ser una buena idea pensar en algunos ejemplos. La lista $[10, 11, 12, 2]$ cumple la propiedad, porque la cantidad es 3, mientras que la lista $[1, 2, 3, 4]$ no la cumple, porque la cantidad es 1. Que en la segunda lista sea 1 es porque la cantidad de dígitos de 1 es 1 mismo, el cual está en la lista. Sin embargo, esta característica de 1 ¿será la idea considerarlo o no? Por otra parte, a priori la lista podría tener números repetidos. ¿En ese caso los contamos varias veces o no? Hacemos las preguntas al cliente y nos responde lo siguiente.

Primero, está bien que 1 cuente. Segundo, de hecho la lista no tiene números repetidos.

Ahora que resolvimos esas dudas, parece ser una buena idea pasar a la siguiente etapa, aunque si llegáramos a darnos cuenta de algún otro detalle, podríamos volver a preguntarle al cliente.

Diseño. La idea es determinar subproblemas más sencillos tales que si los resolvemos a todos, tenemos una solución al problema original. Por ejemplo, podemos considerar los siguientes:

1. Determinar la cantidad de dígitos de un número.
2. Determinar si un número está en una lista.
3. Determinar si un número es primo.

Juntando los items 1 y 2, podemos determinar si la cantidad de dígitos de un número está en la lista. Podemos entonces iterar a lo largo de la lista y para cada número determinar si su cantidad de dígitos está o no, llevando la cuenta de cuántos son. Finalmente, determinamos si ese número es primo.

Implementación y verificación. Escribamos funciones para cada una de los tres items de arriba. Después de cada función, haremos algo de verificación antes de continuar. Para la parte de determinar si un número es primo usamos la función que ya tenemos de antes.

```
def es_primo(n):
    if n<2:
        return False
    puede_ser_primo = True
    potencial_divisor = 2
    while puede_ser_primo and potencial_divisor <= n**(1/2):
        if n % potencial_divisor == 0: #Es un divisor de n
            puede_ser_primo = False
        potencial_divisor = potencial_divisor + 1
    return puede_ser_primo
```

Ingresamos esta función en un archivo llamado `programa.py` y hacemos algunas pruebas desde la terminal (ejecutar python con `-i` permite que luego de ejecutar el programa, lo cual en este caso define la función, pasemos a la terminal de modo interactivo).

```
PS C:\Users\...\Ejemplo metodología> python -i programa.py
>>> es_primo(1)
False
>>> es_primo(2)
True
>>> es_primo(3)
True
>>> es_primo(4)
False
>>> es_primo(5)
True
>>> es_primo(12)
False
>>> es_primo(41)
True
>>> es_primo(49)
False
```

Hasta ahí parece estar andando bien. Para pruebas automáticas podríamos ver por ejemplo si para todos los pares entre 4 y 100 retorna falso. Escribimos por ejemplo el siguiente programa.

```
pasa_test = True
for i in range(2,101):
    if es_primo(2*i):
        pasa_test = False
if pasa_test:
    print("Pasa el test")
else:
    print("No pasa el test.")
```

Agregamos ese programa al final del archivo `programa.py` y lo ejecutamos. Obtenemos la siguiente salida.

```
PS C:\Users\...\Ejemplo metodología> python programa.py
Pasa el test
```

Por lo tanto, el test fue satisfactorio. Nos quedamos conformes con esto y seguimos con el resto (qué tanto testing hacer es a decisión de cada uno). Borramos del archivo `programa.py` este caso de prueba, o lo guardamos en otro archivo distinto.

Escribamos ahora una función para determinar la cantidad de dígitos de un número. Podemos hacer lo siguiente.

```
# la entrada es un entero positivo.
# retorna la cantidad de dígitos del número escrito en base 10.
def cant_digitos(n):
    cant = 1
    q = n // 10
    while q > 0:
        cant = cant + 1
        q = q // 10
    return cant
```

Notar que arriba de la función escribimos un comentario que aclara lo que hace. Esto es una muy buena práctica que se recomienda aplicar en general (aunque aquí lo hacemos solo con esta función). Si bien ahora tenemos claro lo que hace, si volvemos a mirar la función dentro de un tiempo (o si la mira otra persona), el comentario puede ser útil. Agregamos esta función al archivo `programa.py` y hacemos pruebas del mismo modo que con la función anterior. Hacemos algunas pruebas a mano. Para testing automático podemos verificar por ejemplo que para cada a entre 1 y 9 y para cada n menor a 100, la función retorna $n + 1$.

```
pasa_test = True
for a in range(1,10):
    for n in range(100):
        if cant_digitos(a*10**n) != n+1:
            pasa_test = False
```

```

if pasa_test:
    print("Pasa el test")
else:
    print("No pasa el test.")

```

Verificamos que se pasa el test y continuamos con la siguiente. Vamos a escribir una función que diga si un número está en una lista. Podríamos usar la operación `in` de listas, pero hagámoslo *a mano*.

```

def pertenece(n,l):
    for i in l:
        if i == n:
            return True
    return False #Si llega acá es porque no está

```

Queda como ejercicio hacer un poco de testing para esta función.

Finalmente, usando lo anterior como dijimos en la etapa de diseño podemos escribir una función que resuelva el problema.

```

def problema(l):
    cant = 0
    for n in l:
        if pertenece(cant_digitos(n),l):
            cant = cant + 1
    return es_primo(cant)

```

Notar que al haber separado el problema en subproblemas, la función final quedó muy sencilla. Finalmente el contenido del archivo `programa.py` es:

```

def es_primo(n):
    if n<2:
        return False
    puede_ser_primo = True
    potencial_divisor = 2
    while puede_ser_primo and potencial_divisor < n:
        if n % potencial_divisor == 0: #Es un divisor de n
            puede_ser_primo = False
        potencial_divisor = potencial_divisor + 1
    return puede_ser_primo

# la entrada es un entero positivo.
# retorna la cantidad de dígitos del número escrito en base 10.
def cant_digitos(n):
    cant = 1
    q = n // 10
    while q > 0:
        cant = cant + 1
        q = q // 10

```

```

        return cant

def pertenece(n,l):
    for i in l:
        if i == n:
            return True
    return False #Si llega acá es porque no está

def problema(l):
    cant = 0
    for n in l:
        if pertenece(cant_digitos(n),l):
            cant = cant + 1
    return es_primo(cant)

```

Lo siguiente son algunas pruebas.

```

PS C:\Users\...\Ejemplo metodología> python -i programa.py
>>> l = [1,2,3]
>>> problema(l) #Hay 3
True
>>> l.append(4)
>>> problema(l) # Hay 4
False
>>> l.append(10) # Hay 5
>>> problema(l)
True
>>> l.append(123123123)
>>> problema(l) #El número nuevo no afecta
True

```

Queda como ejercicio para quien quiera hacer pruebas automatizadas. Si fuera un problema real claramente habría que verificar bien a la función final.

Capítulo 3

Funciones recursivas

Veremos una nueva forma de definir funciones que se piensa muy distinto a lo visto en el capítulo 2. Esta forma de definir funciones está fuertemente vinculada con la inducción completa y de hecho podremos analizar las funciones resultantes utilizando inducción de modo muy natural.

Una función recursiva es una función que se define de modo cíclico a partir de ella misma, de un modo tal que el ciclo termina (no entra en un ciclo infinito). Veamos un primer ejemplo.

El factorial se define como la función $f : \mathbb{N} \rightarrow \mathbb{N}$ denotada por $f(n) = n!$ tal que para 0 da 1 y para los otros números es el producto de todos los menores o iguales. Por ejemplo, $1! = 1$, $2! = 2 \times 1$, $3! = 3 \times 2 \times 1$, etcétera. El modo formal de definirlo en matemática es a partir de las siguientes reglas, que conforman una definición recursiva.

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad \text{si } n > 0 \end{aligned}$$

Escrito en términos de función f , es que $f(0) = 1$ y para todo $n > 0$, se cumple que $f(n) = n f(n-1)$. Notar que estas dos reglas nos permiten calcular $n!$ para cualquier $n \in \mathbb{N}$. A modo de ejemplo calculemos $4!$. Por la segunda regla tenemos que $4! = 4 \times 3!$, por lo que pasamos a calcular $3!$. Por la segunda regla nuevamente, $3! = 3 \times 2!$, por lo que pasamos a calcular $2!$. Por la segunda regla nuevamente, $2! = 2 \times 1!$, por lo que pasamos a calcular $1!$. Por la segunda regla nuevamente, $1! = 1 \times 0!$, por lo que pasamos a calcular $0!$. Ahora por la primera regla $0! = 1$, por lo que volviendo para atrás, como $1! = 1 \times 0!$, tenemos que $1! = 1$; como $2! = 2 \times 1!$, tenemos que $2! = 2$; como $3! = 3 \times 2!$, tenemos que $3! = 6$ y como $4! = 4 \times 3!$, tenemos que $4! = 24$.

Como vimos en el ejemplo anterior, a partir de la definición recursiva podemos calcular el factorial de forma que parece automatizable. En los lenguajes de programación efectivamente se puede definir funciones recursivas y la máquina las calcula sola, de modo similar al del ejemplo anterior. En python la función factorial se puede escribir de la siguiente forma.

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

Notar que en la última línea, que se ejecuta en caso de que $n > 0$, se llama la misma función `fact` pero con argumento `n-1`. Esto se denomina una llamada recursiva. Como estamos llamando la función con un argumento más chico, en algún momento se llega al 0, caso en el que se entra en el cuerpo del `if` y la función retorna sin hacer más llamadas recursivas. Estamos suponiendo que la función se ejecuta solo con $n \in \mathbb{N}$. En caso de ejecutarla con un número negativo, entraría en un ciclo infinito.

Comenzamos con un repaso de inducción y una presentación más formal del concepto de recursión, mostrando como el razonamiento inductivo sirve para diseñar estas funciones y para demostrar propiedades. Veremos funciones recursivas con enteros (como el factorial) y también con listas.

3.1. Inducción y recursión

El principio de inducción completa sirve para demostrar que una propiedad se cumple para todos los números naturales, es decir, dada una propiedad $P(n)$, en la que n es un número natural, demostrar que $\forall n \in \mathbb{N} P(n)$. El enunciado es el siguiente.

Principio de inducción completa. Sea $P(n)$ una propiedad sobre números naturales. Si se cumplen:

1. $P(0)$, es decir la propiedad en $n = 0$.
2. $\forall n > 0, P(n-1) \Rightarrow P(n)$, es decir, para todo número natural $n > 0$, si se cumple $P(n-1)$ entonces también se cumple $P(n)$.

Entonces la propiedad $P(n)$ se cumple para todo $n \in \mathbb{N}$, es decir, tenemos $\forall n \in \mathbb{N} P(n)$.

Para aplicar el principio de inducción completa, hay que probar que se cumple $P(0)$, lo cual se llama **paso base** y luego dado un $n > 0$ genérico, probar $P(n-1) \Rightarrow P(n)$, es decir, que $P(n-1)$ implica $P(n)$, lo cual se llama **paso inductivo**. En el paso inductivo, tenemos la **hipótesis inductiva**, que es $P(n-1)$ y la **tesis inductiva**, que es $P(n)$. Lo que hay que hacer es demostrar la tesis inductiva pudiendo hacer uso de la hipótesis inductiva. Si logramos hacer tanto el paso base como el inductivo, por inducción completa concluimos que $\forall n \in \mathbb{N} P(n)$.

El principio de definición de funciones por recursión está fuertemente vinculado con el de inducción completa. Se trata de el siguiente.

Definición de función por recursión. Queremos definir una función $f : \mathbb{N} \rightarrow \mathbb{N}$. Si:

1. Definimos $f(0)$, es decir, el valor de f en $n = 0$.
2. Para una natural $n > 0$ cualquiera, asumiendo que $f(n-1)$ ya está definido, definimos $f(n)$ pudiendo usar $f(n-1)$.

entonces la función f queda definida para todos los números naturales.

Se le puede llamar **caso base** a la parte de definir $f(0)$ y **caso recursivo** a la parte de definir $f(n)$ pudiendo usar $f(n-1)$. El factorial como lo presentamos es un ejemplo de función definida por recursión.

Hay otra variante de inducción completa, llamada inducción fuerte. La diferencia es que en el paso inductivo, en lugar de asumir que la propiedad se cumple en $n - 1$ para probarla en n , se asume que la propiedad se cumple en todos los $k < n$ y se utiliza esto para demostrar que también se cumple en n . El enunciado es el siguiente.

Principio de inducción fuerte. Sea $P(n)$ una propiedad sobre números naturales. Si se cumplen:

1. $P(0)$, es decir la propiedad en $n = 0$.
2. Para cualquier natural n , $(\forall k < n, P(k)) \Rightarrow P(n)$, es decir, para todo número natural n , si se cumple $P(k)$ para todo $k < n$, entonces también se cumple $P(n)$.

Entonces la propiedad $P(n)$ se cumple para todo $n \in \mathbb{N}$, es decir, tenemos $\forall n \in \mathbb{N} P(n)$.

En algunos casos puede convenir utilizar el principio de inducción fuerte en lugar del de inducción completa. Además, el principio de inducción fuerte tiene asociado un mecanismo de definición de funciones por recursión que al momento de definir la función en n , permite usar la función en todos los $k < n$, en lugar de solamente $n - 1$.

Variación de recursión. Queremos definir una función $f : \mathbb{N} \rightarrow \mathbb{N}$. Si:

1. Definimos $f(0)$, es decir, el valor de f en $n = 0$.
2. Para una natural $n > 0$ cualquiera, asumiendo que $f(k)$ ya está definido para todo $k < n$, definimos $f(n)$ pudiendo usar $f(k)$ para cualquier $k < n$.

entonces la función f queda definida para todos los números naturales.

Muchas veces se le llama inducción a secas tanto a la inducción completa como a la inducción fuerte y recursión a secas a cualquiera de los dos tipos de recursiones.

Si bien enunciamos el principio de definición por recursión para funciones $f : \mathbb{N} \rightarrow \mathbb{N}$, el único que si o si debe ser \mathbb{N} es el dominio. El codominio puede ser otro conjunto X (por ejemplo un conjunto de listas o de strings) y el principio se puede aplicar para justificar la existencia de una función $f : \mathbb{N} \rightarrow X$. Esto es relevante porque podemos querer aplicarlo para definir funciones que dado un número retornan un string o una lista.

3.2. Funciones recursivas en python

Definamos lo que son las funciones recursivas dentro de la programación y veamos algunos ejemplos. Posteriormente nos enfocaremos más en cómo pensarlas y estudiar sus propiedades.

En el contexto de la programación, una función recursiva es una función que se llama a ella misma en su cuerpo, es decir, algo como:

```
def f(n):
    cuerpo
```

donde dentro de `cuerpo` aparece `f(k)` con algún otro argumento `k`. También podría tener más parámetros, como en ejemplos que veremos más adelante.

En general las funciones recursivas se definen basándose en algún principio de definición por recursión como los vistos en la sección 3.1. Esto permite tener seguridad de que

la función recursiva efectivamente termina sin entrar en un ciclo infinito. Lo más común es separar el caso base del recursivo con un `if`. Si aplicamos la primera forma de recursión vista en la sección anterior, la estructura puede ser la siguiente. El paso base se aplica si el argumento es 0 y en otro caso se aplica el recursivo.

```
def f(n):
    if n == 0:
        #Caso base. Definimos el valor de f(0)
        return ...
    else:
        #Caso recursivo. Definimos f(n) pudiendo usar f(n-1)
        rec = f(n-1) # Llamada recursiva
        return ...(acá usamos rec)...
```

Un ejemplo de esto es el factorial que ya vimos al principio del capítulo, donde la definición conceptual por recursión es

$$0! = 1$$

$$n! = n \times (n-1)! \quad \text{si } n > 0$$

y la función computacional se puede escribir como

```
def fact(n):
    if n == 0:
        return 1
    else:
        rec = fact(n-1)
        return n*rec
```

Aquí al final en vez de escribir una única línea `return n*fact(n-1)`, lo separamos en dos para que la llamada recursiva esté en su propia línea, lo cual es equivalente.

Notar la similitud entre el concepto de inducción completa, el de definición de función por recursión y las funciones recursivas escritas en el lenguaje de programación. Debido a esta cercanía entre los conceptos, suelen poderse hacer de modo bastante natural pruebas por inducción sobre propiedades de estos programas, lo cual haremos más adelante.

Se puede escribir funciones recursivas que no necesariamente sigan ninguno de los principios anteriores, pero que por algún otro argumento sepamos que terminan. Consideremos por ejemplo la siguiente función.

```
def g(n):
    if n < 10:
        return n
    else:
        q = n // 10
        return g(q)
```

En el cuerpo del `if` definimos el valor de la función cuando $n < 10$, y en el `else` hacemos una llamada recursiva con el cociente de dividir n entre 10 como argumento. No se corresponde con el principio anterior de definir para $n = 0$ y luego en el caso recursivo

usar el valor en $n - 1$ para definir el de n , pero de todos modos podemos ver que la función termina. Si $n < 10$ entonces termina directamente. Por otra parte, que si $n \geq 10$, la llamada recursiva es con q que cumple que $q < n$, por lo que en algún momento se llega a un argumento < 10 y termina.

Lo que hace esta función es retornar el primer dígito en base 10 del número n . La idea es que si $n < 10$ entonces es un solo dígito y en otro caso, dividir entre 10 lo que hace es quitar el último dígito, por lo que podemos hacer la llamada recursiva con el cociente.

También se pueden hacer funciones recursivas con objetos de otros tipos de datos, como listas. Para las listas, lo que se suele hacer es definir el caso base con la lista vacía y en el caso recursivo utilizar una lista más chica que la original, lo cual asegura que el algoritmo termina. Para este tipo de programas, luego se suele poder probar propiedades por inducción sobre el tamaño de la lista.

Por ejemplo, supongamos que queremos hacer un programa por recursión que suma los elementos de una lista de números. Una posibilidad es la siguiente.

```
def suma(lista):
    if len(lista) == 0:
        #La lista es vacía, por lo tanto la suma es 0
        return 0
    else:
        # Comenzamos separando guardando el primer elemento en
        # una variable y el resto de la lista en otra variable
        x = lista[0]
        resto = lista[1:len(lista)]
        # como el resto es una lista más chica, hacemos recursión
        suma_resto = suma(resto)
        # la suma de toda la lista, es x más la suma del resto
        return x + suma_resto
```

De modo más compacto, puede escribirse como:

```
def suma(lista):
    if len(lista) == 0:
        return 0
    else:
        return lista[0] + suma(lista[1:len(lista)])
```

Si la lista es vacía, la función termina en una etapa. En otro caso, se hace un llamado recursivo con la lista que tiene un elemento menos (se quita el primero), por lo que en algún momento se llega a la lista vacía y la ejecución termina.

Análogamente a como se definen funciones recursivas con listas, se pueden definir también con strings, por ejemplo haciendo el caso base con el string vacío y luego el caso recursivo llamando con un substring más chico. La siguiente es una función que cuenta la cantidad de veces que aparece el carácter “a”.

```
def contar(s):
    if len(s) == 0:
        return 0
```

```

else:
    # la llamada recursiva es con s[1:len(s)] que es
    # el resto de la lista que queda después del elemento s[0]
    resto = s[1:len(s)]
    if s[0] == "a":
        return contar(resto) + 1
    else:
        return contar(resto)

```

Las funciones recursivas no tienen por qué tener un único argumento. Por ejemplo, a partir de la función `g` que retorna el primer dígito en base 10, podemos agregarle un nuevo parámetro `b` y que retorne el primer dígito en base `b`. La función resultante es la siguiente.

```

def g(n, b):
    if n < b:
        return n
    else:
        q = n // b
        return g(q, b)

```

En esta función hacemos recursión en `n`. El argumento `b` se mantiene igual en las llamadas recursivas.

3.2.1. Límite de recursión

Ejecutar una función requiere cierto espacio de memoria. En particular, en una función recursiva, cada llamada sucesiva precisa más memoria y esto implica que hay un límite en la cantidad que se pueden hacer dado por la memoria disponible y no solo por el tiempo que lleve la ejecución.

En python hay un límite fijado de alrededor de 1000 llamadas recursivas sucesivas. En caso de que esto se exceda, se produce un error y el programa termina.

En este sentido, las funciones recursivas tienen un límite que puede ser mucho más restrictivo que con las funciones escritas por iteraciones. Por ejemplo, la función de suma de todos los elementos de una lista por recursión solo funciona para listas de hasta 1000 elementos, mientras que una función escrita iterativamente (es decir, con un `for` por ejemplo) podría perfectamente usarse para listas con millones de elementos.

Como suele pasar, hay situaciones en las que la recursión es una buena opción y situaciones en las que no lo es. Tiene la ventaja de que para ciertos problemas permite hacer algoritmos sencillos basados en un razonamiento inductivo. Por otra parte, el límite de recursión es para llamadas recursivas sucesivas. Hay algoritmos que por la forma como organizan las llamadas, el tamaño de los objetos con los que se pueden usar depende exponencialmente del límite de recursión. En otras palabras, si el límite es 1000, se pueden ejecutar en teoría en objetos de tamaño hasta 2^{1000} , por ejemplo. Uno de estos algoritmos es la búsqueda binaria en listas, que veremos en la sección 3.4.1.

3.3. Diseño y estudio de funciones recursivas

La forma de pensar para escribir una función recursiva que resuelve un problema es muy distinta a la forma de pensar para escribir una iterativa. Es mucho más parecido al razonamiento que se usa al demostrar propiedades por inducción. En una prueba por inducción tenemos que demostrar el caso base y el caso inductivo, mientras que en una función recursiva tenemos que escribir el caso base y el caso recursivo. Veamos cómo se hace con números y listas. El caso de strings es análogo al de listas, pensando al string como una lista de caracteres.

3.3.1. Recursión con números naturales

Una función recursiva con números en general tiene la siguiente forma.

```
def f(n):  
    if n == 0:  
        caso base  
    else:  
        caso recursivo
```

En el paso base simplemente tenemos que retornar lo que deba dar la función para $n = 0$. Qué valor es exactamente depende del problema.

En el paso recursivo tenemos que retornar lo que vale la función para un $n > 0$ pero con la posibilidad de utilizar la función en valores $k < n$ asumiendo que para esos valores la función resuelve el problema bien. Al pensar esta parte, hay que ver cómo podemos usar valores de la función en números anteriores para calcular el valor en n . Muchas veces el valor anterior que se utiliza es $k = n - 1$, caso en el que se corresponde con el razonamiento con inducción completa.

Una vez que hicimos las dos cosas, la función recursiva queda completa y para estudiarla se usan razonamientos inductivos, que comúnmente resultan ser muy parecidos a lo que se hace para pensarla.

La estructura puede flexibilizarse, por ejemplo haciendo que el caso base no sea solamente con $n = 0$, como en la función `g` definida en la sección 3.2.

Ejemplo. escribamos una función que dado n calcula $f(n) = \sum_{i=0}^n i$. Usamos f para la función matemática y `f` para la función programa.

Para el caso base debemos retornar $f(0)$, que es $\sum_{i=0}^0 i = 0$. Por lo tanto, será solamente `return 0`.

Pensemos ahora el caso recursivo. Dado $n > 0$ tenemos que calcular $f(n) = \sum_{i=0}^n i$, pudiendo usar $f(k)$ para valores $k < n$, asumiendo que para esos valores `f(k)` ya calcula la sumatoria correspondiente. Es decir, para cada $k < n$, si escribimos `f(k)` eso da $\sum_{i=0}^k i$ y lo podemos usar para calcular $f(n)$, lo cual es la ventaja de usar recursión.

En particular, si escribimos `f(n - 1)` esto dará $\sum_{i=0}^{n-1} i$, pero observamos que para ser $\sum_{i=0}^n i$ solamente le falta sumar un n , es decir, se cumple $n + \sum_{i=0}^{n-1} i = \sum_{i=0}^n i$. En base a esto, para calcular $f(n)$ podemos hacer $n + f(n - 1)$.

En base al razonamiento anterior, la función recursiva es la siguiente.

```
def f(n):
```

```

if n == 0:
    return 0
else:
    return n + f(n-1)

```

Hagamos una demostración por inducción completa de que $\forall n \in \mathbb{N} \ f(n) = \sum_{i=0}^n i$. La propiedad que usaremos para la inducción es $P(n) : f(n) = \sum_{i=0}^n i$. Cuando decimos que $f(n) = x$ nos referimos a que la función f al evaluarse en n retorna x . Hay que probar el paso base y el paso inductivo.

Para el paso base, hay que probar $P(0)$, lo cual es que $f(0) = \sum_{i=0}^0 i$. Tenemos que $\sum_{i=0}^0 i = 0$ y por el código $f(0) = 0$, por lo que el paso base se cumple.

Hagamos ahora el paso inductivo. La hipótesis es que $f(n-1) = \sum_{i=0}^{n-1} i$, que es $P(n-1)$, mientras que la tesis es que $f(n) = \sum_{i=0}^n i$, que es $P(n)$. Hay que demostrar la tesis pudiendo usar la hipótesis. El razonamiento es el siguiente. Por el código tenemos que $f(n) = n + f(n-1)$. Juntando eso y la hipótesis inductiva tenemos que:

$$f(n) = n + f(n-1) = n + \sum_{i=0}^{n-1} i = \sum_{i=0}^n i$$

por lo tanto concluimos que se cumple la tesis inductiva y terminamos el paso inductivo.

Habiendo probado el paso base y el paso inductivo, concluimos que $\forall n \in \mathbb{N} \ f(n) = \sum_{i=0}^n i$, es decir, que la función que escribimos efectivamente calcula lo que queríamos.

Observar que para el caso recursivo usamos la propiedad que tiene la función que queríamos calcular de que $f(n) = n + f(n-1)$, pensando en f como la función matemática y no como el programa. Es común que lo que escribimos en el paso recursivo venga de una propiedad que tiene la función matemática que vincula su valor en n con su valor en algún número anterior (como $n-1$, en este caso).

3.3.2. Recursión con listas

Las funciones recursivas con números se suelen basar en alguno de los principios de recursión presentados en la sección 3.1 o similares. Por otra parte, la definición de funciones recursivas con listas se suele basar en principios como el siguiente.

Principio de recursión en listas. Denotemos L al conjunto de las listas. Supongamos que queremos definir una función $f : L \rightarrow L$. Si:

1. Definimos $f([])$, es decir, el valor de f en la lista vacía.
2. Para una lista $l \in L$ de largo $\text{len}(l) > 0$ cualquiera, asumiendo que $f(s)$ ya está definido para toda lista s con $\text{len}(s) < \text{len}(l)$, definimos $f(l)$ pudiendo usar los valores de $f(s)$ para las listas lista s con $\text{len}(s) < \text{len}(l)$.

entonces la función f queda definida para todas las listas.

El paso base se hace con la lista vacía y el recursivo se hace pudiendo usar la misma función para listas más chicas.

La justificación de este principio de recursión en listas es por inducción en el largo de la lista.

Dijimos que el conjunto L es el de las listas, pero no tiene por qué ser el conjunto de todas las listas. Puede ser también el conjunto de las listas de números enteros, el de las listas de strings, etc. La mayoría de los ejemplos van a ser con listas de números enteros.

Al igual que con números, lo que importa es que el dominio de la función (o sea, el conjunto de las entradas) sea el de las listas. El codominio (o sea, el conjunto de las salidas) en realidad puede ser cualquier conjunto X , dando lugar a funciones $f : L \rightarrow X$. Esto es útil porque podemos querer usar el principio para hacer funciones que a partir de una lista calculen un número o un string.

Pasemos a la programación. Una función recursiva con listas en general tiene la siguiente forma.

```
def f(l):
    if len(l) == 0:
        caso base
    else:
        caso recursivo
```

En el paso base tenemos que retornar lo que da la función para la lista vacía. Notar que la condición `len(l)==0` solo se cumple con la lista vacía.

En el paso recursivo tenemos que retornar lo que vale la función para una lista l de largo `len(l)>0`, es decir, que tiene al menos un elemento, pero con la posibilidad de utilizar la función en listas lp de largo `len(lp)=k<n` asumiendo que para esas listas la función resuelve el problema bien. Al pensar esta parte, hay que ver cómo podemos usar valores de la función en listas más chicas para calcular el valor en la lista l . Muchas veces la que se usa es `lp = l[1:len(l)]`, la lista que consiste en los elementos de l desde el segundo en adelante. Notar que esta lista tiene largo `len(l)-1`.

La función resultante se puede estudiar utilizando inducción en el largo de la lista, es decir, si queremos probar que una propiedad $Q(l)$ se cumple para toda lista l , hacemos inducción con $P(n) : \text{«se cumple } Q(l) \text{ para toda lista } l \text{ de largo } n\text{»}$.

La estructura se puede flexibilizar, por ejemplo haciendo el caso base con listas de largo 1.

Ejemplo. Escribamos una función recursiva que dada una lista l , retorne otra lista que sea l invertida, es decir, con los mismos elementos en el orden contrario.

En el caso base debemos resolverlo para el caso de la lista vacía. Claramente la lista vacía invertida es también la lista vacía, así que hacemos `return []`.

En el caso recursivo, tenemos l una lista de largo `len(l)>0`. Debemos calcular la función para l , pudiendo hacer llamadas recursivas para cualquier lista lp que cumpla `len(lp)<len(l)`, asumiendo que para esas listas la función hace lo que tiene que hacer (retorna su inversión). Vamos a aplicar recursión con la lista `resto=l[1:len(l)]`, a la que llamamos de esa forma porque es el resto de l después de el primer elemento. Notar que `len(resto)=len(l)-1`. Supongamos que l es $[x_1, x_2, \dots, x_n]$. En este caso, `resto` es $[x_2, \dots, x_n]$. Por recursión, podemos asumir que `f(resto)` ya calcula lo que debe calcular, es decir, que da $[x_n, \dots, x_2]$. Por lo tanto, para obtener la inversión de l , lo único que hay que hacer es agregar x_1 al final de `f(resto)`, lo cual se puede hacer con la instrucción `append`.

En base al razonamiento anterior, escribimos la siguiente función.

```

def inv(l):
    if len(l) == 0:
        return []
    else:
        resto = l[1:len(l)]
        rec = inv(resto) # llamada recursiva en resto
        rec.append(l[0]) # agregar el primer elemento de l al final
        return rec

```

Hagamos una demostración por inducción completa de que la función es correcta. Al igual que en el caso de la función con enteros, el razonamiento será muy similar al que hicimos para escribir la función recursiva. La propiedad de inducción es $P(n)$: «para cualquier lista l de largo n , se cumple que `inv(l)` retorna la inversión de l ».

Para el paso base hay que probar $P([])$. Como la inversión de la lista vacía es la lista vacía, hay que justificar que `inv([])` retorna la lista vacía. Como `len([])==0` da `True`, se entra en el cuerpo del `if` y la función retorna efectivamente la lista vacía.

Para el paso inductivo, dado $n > 0$, asumimos que se cumple $P(n - 1)$ y debemos probar $P(n)$. La hipótesis inductiva lo que nos dice es que para toda lista `lp` de largo $n - 1$ se cumple que `inv(lp)` retorna la inversión de `lp`. Para probar la tesis, tomamos $l = [x_1, x_2, \dots, x_n]$ una lista cualquiera de largo n y debemos probar que `inv(l)` retorna la inversión de l , es decir $[x_n, \dots, x_2, x_1]$. Como `len(l) = n > 0`, se entra al cuerpo del `else`. Tenemos que `resto` es $[x_2, \dots, x_n]$ y tiene largo $n - 1$, así que por hipótesis inductiva, `inv(resto)` da $[x_n, \dots, x_2]$ y `rec` queda con ese valor. Al hacer el `append`, como `l[0]` vale x_1 , la lista `rec` pasa a valer $[x_n, \dots, x_2, x_1]$. Por lo tanto, se retorna $[x_n, \dots, x_2, x_1]$ que es la inversión de l , así que la tesis se cumple.

Como probamos el paso base y el inductivo, por el principio de inducción completa concluimos $\forall n \in \mathbb{N} P(n)$, lo cual por la definición de $P(n)$ nos dice que para todo largo n , la función retorna lo que tiene que retornar para todas las listas de largo n . Por lo tanto, la función retorna lo que tiene que retornar para cualquier lista.

Veamos dos ejemplos más. Para estos dos, se presenta el código con algunos comentarios y queda como ejercicio hacer el razonamiento detallado del mismo modo que con el ejemplo anterior.

La primera es una función para resolver el problema de dada una lista ordenada y un elemento nuevo, insertarlo de modo ordenado. La función es `insert(x, l)`. El parámetro `x` es un entero y el parámetro `l` es una lista de enteros que está ordenada. Lo que hace la función es insertar `x` ordenadamente en `l`, es decir, de modo que siga siendo una lista ordenada, y retornar esta lista resultante.

Tener en cuenta que la función asume que el segundo argumento es una lista ordenada. No tenemos que preocuparnos por lo que pasa si no lo es, porque no es parte de lo que se supone que esta función hace. El argumento con el que aplicaremos recursión es `l`.

```

def insert(x, l):
    if len(l) == 0:
        # la lista es vacía, así que el resultado tiene solo a x
        return [x]
    else:
        # la lista tiene al menos un elemento

```

```

if x <= l[0]:
    # en este caso x es menor al primer elemento,
    # por lo que va al principio
    return [x] + l # concatenación de listas para agregar
                    # x al comienzo
    # en este caso no fue necesario hacer llamada recursiva
else:
    # en este caso x es mayor al primer elemento,
    # así que por recursión lo insertamos en el resto
    resto = l[1:len(l)]
    rec = insert(x,resto)
    # rec tiene el x insertado en el lugar que le corresponda
    # del resto de la lista.
    # solo falta agregarle l[0] al comienzo
    return [l[0]] + rec

```

La siguiente función es una que ordena listas de enteros. La definimos `ordenar(l)`. El parámetro `l` es una lista de enteros. La función retorna una lista con los mismos elementos que `l` pero en orden. Para escribir esta función utilizaremos la que ya programamos.

```

def ordenar(l):
    if len(l) == 0:
        return []
    else:
        resto = l[1:len(l)]
        rec = ordenar(resto)
        # por recursión, rec es la lista con todos los elementos
        # menos el primero ya ordenados
        # solo resta insertar al primer elemento en dónde vaya,
        # lo cual podemos hacer con la función anterior.
        return insert(l[0],rec)
        # recordar que la función insert funciona si la lista
        # del segundo parámetro está ordenada,
        # pero por recursión, sabemos que rec está ordenada

```

3.4. Eficiencia de algoritmos

Distintos algoritmos para resolver un problema pueden tener distintas propiedades y en algunos casos decimos que uno es más eficiente que otro. Hay distintos tipos de eficiencia, como por ejemplo la eficiencia temporal y la eficiencia en uso de memoria. Vamos a estudiar algoritmos del punto de vista de eficiencia temporal, es decir, analizando el tiempo que requiere la ejecución.

Para estudiar el tiempo que requiere la ejecución de un algoritmo, vamos a contar la cantidad de operaciones que deben realizarse. Hay otros factores que en la práctica afectan para la eficiencia temporal de un algoritmo, como el aprovechamiento de la memoria *cache* (cierta memoria de acceso rápido que tiene la computadora) o la posibilidad de

paralelizar las operaciones y utilizar varios procesadores a la vez. No obstante, para una primera aproximación nos limitaremos a estudiar la cantidad de operaciones.

Dada una función `f(l)` cuyo parámetro de entrada es una lista, vamos a estudiar cómo depende la cantidad de operaciones que se realiza en función del largo de la lista, $n = \text{len}(l)$. Por ejemplo, si la cantidad de operaciones no depende de n , decimos que es de orden constante, lo cual se denota $O(1)$. Si la cantidad de operaciones es proporcional a n , diremos que el algoritmo es de orden lineal, lo cual se denota $O(n)$. Si es proporcional a n^2 , diremos que el algoritmo es cuadrático, lo cual se denota $O(n^2)$. Para otras funciones se usa la misma notación con la O . En general si la cantidad de operaciones es proporcional a $f(n)$, decimos que el algoritmo es $O(f(n))$, lo cual se lee «orden $f(n)$ ». Hay algunos otros órdenes que tienen nombres concretos. Por ejemplo $O(n^3)$ es cúbico y $O(\log(n))$ es logarítmico.

El concepto de orden no se aplica solamente para algoritmos en listas. Es en general cómo depende la cantidad de operaciones del tamaño de la entrada. En caso de listas o strings, el tamaño suele ser el largo del objeto, mientras que en caso de números, puede usarse tanto el número en sí o como su cantidad de dígitos, dando lugar a distintos valores de los órdenes.

La definición formal de orden es la siguiente. Dada una función $f(n)$ y otra función $g(n)$ (en nuestro caso $g(n)$ siempre es la función que a partir de un n nos dice cuantas instrucciones hace el algoritmo con entradas de tamaño n), decimos que $g(n) = O(f(n))$ si se cumple que $\exists n_0 \in \mathbb{N} \exists C > 0 \forall n > n_0, |g(n)| \leq C f(n)$.

Para estudiar el orden de algoritmos iterativos hay que contar la cantidad de iteraciones que se realizan. Por ejemplo, consideremos el siguiente algoritmo que suma todos los elementos de una lista.

```
def suma(l):
    cont = 0
    for i in l:
        cont = cont + i
    return cont
```

Como el cuerpo del `for` se ejecuta una vez para cada elemento de la lista, si la lista tiene largo n , la cantidad de operaciones es aproximadamente n . Contando la asignación del principio y el `return` serían $n+2$, pero como lo que importa es el orden, el 2 se desprecia. En conclusión, el algoritmo es lineal, o escrito de otra forma, $O(n)$.

Para estudiar el orden de algoritmos recursivos, en general primero se determina una recurrencia para la sucesión $(a_n)_{n \in \mathbb{N}}$ de la cantidad de operaciones en función de n (es decir, la sucesión tal que a_n es la cantidad de operaciones con entrada de tamaño n). Luego, esa recurrencia se resuelve, obteniendo una fórmula para a_n . Finalmente, determinamos el orden de a_n (con la notación de arriba de $g(n) = O(f(n))$, aquí a_n es la $g(n)$). La recurrencia en general consiste en un valor inicial que se deduce del caso base y una ecuación que vincula a_n con valores anteriores, como a_{n-1} , la cual se deduce del caso recursivo.

Consideremos el algoritmo recursivo que invierte la lista.

```
def inv(l):
    if len(l) == 0:
```

```

    return []
else:
    resto = l[1:len(l)]
    rec = inv(resto) # llamada recursiva en resto
    rec.append(l[0]) # agregar el primer elemento de l al final
    return rec

```

Definimos $(a_n)_{n \in \mathbb{N}}$ como la sucesión que indica la cantidad de instrucciones que se ejecutan con una lista de tamaño n . Determinemos la recurrencia que se deduce del código. Esto consiste de un valor inicial que proviene del caso base y una regla para calcular un valor a partir del anterior la cual proviene del caso recursivo.

A partir del caso base deducimos que con listas de tamaño 0 la función realiza una sola instrucción, por lo que $a_0 = 1$.

A partir del caso recursivo, vemos que si $n > 0$, entonces se llama recursivamente la función y se hacen otras 3 instrucciones. Por lo tanto, la cantidad total de instrucciones es 3 más la cantidad de instrucciones que se hacen en la llamada recursiva. Como esta llamada recursiva es con una lista de largo $n - 1$, la cantidad de instrucciones que se hacen ahí es por definición a_{n-1} . Por lo tanto, llegamos a que $a_n = 3 + a_{n-1}$ si $n > 0$.

En conclusión la recurrencia que tenemos es:

$$\begin{aligned}
 a_0 &= 1 \\
 a_n &= 3 + a_{n-1} \quad \text{si } n > 0
 \end{aligned}$$

Empieza en 1 y en cada paso se suma 3. De esto deducimos que $a_n = 1 + 3n$, fórmula que se puede demostrar más formalmente por inducción. Podemos ver que $a_n = O(n)$, es decir, que el algoritmo es de orden lineal.

En algunos casos la cantidad de instrucciones que se realizan puede variar para distintas entradas del mismo tamaño. Es decir, pueden haber dos lista 11 y 12 con el mismo tamaño y que la cantidad de instrucciones sea distinta para estas. En este caso un enfoque que puede utilizarse es el de considerar la cantidad de instrucciones en el peor caso, es decir, para cada n contabilizar la mayor cantidad de instrucciones que pueden haber con una lista de largo n . Consideremos a modo de ejemplo el algoritmo de inserción ordenada.

```

def insert(x,l):
    if len(l) == 0:
        return [x]
    else:
        if x <= l[0]:
            return [x] + l
        else:
            resto = l[1:len(l)]
            rec = insert(x,resto)
            return [l[0]] + rec

```

Notar que en el caso de que la lista tiene largo $n > 0$, si x es menor que el primer elemento, entonces termina con una sola instrucción. Sin embargo, en otro caso se hace la llamada recursiva y dos instrucciones más, lo que da una cantidad distinta de instrucciones. Con el enfoque de peor caso, lo que hacemos es suponer que siempre se entrará en el segundo

`else`, que es lo que da lugar a más instrucciones. De hecho esto es lo que pasa si el elemento x va en el último lugar de la lista. Con esta suposición llegamos a la siguiente recurrencia.

$$\begin{aligned}a_0 &= 1 \\ a_n &= 2 + a_{n-1}\end{aligned}$$

Por lo tanto, resulta $a_n = 1 + 2n$ y $a_n = O(n)$. Concluimos que el algoritmo es en peor caso de orden lineal.

Hay una cierta arbitrariedad en cómo se cuentan las instrucciones, lo cual puede cambiar la expresión exacta de la sucesión a_n pero no el orden. Por ejemplo, en el último algoritmo podemos contabilizar la evaluación de las condiciones de los `if` como instrucciones, llegando a la recurrencia distinta $a_0 = 2$ y $a_n = 4 + a_{n-1}$. Sin embargo, esto solamente cambia constantes de la fórmula final que no afectan el orden, pues queda $a_n = 2 + 4n$ que también es $O(n)$.

En base a la observación anterior, lo que nos importa es el orden de la sucesión a_n y no su expresión exacta.

Por otra parte, por simplicidad asumimos que todas las instrucciones básicas de python cuentan como una sola instrucción (suponiendo de fondo que tardan lo mismo), pero esto no tiene por qué ser el caso. Por ejemplo, las operaciones de indizado y slicing en listas, puede que en realidad el tiempo que requieran aumente con el largo de la lista. Sin embargo, nosotros contaremos una línea como `y = l[k]` como una sola instrucción.

3.4.1. Búsqueda

Veamos algo de algoritmos de búsqueda en listas. La consigna es que dada una lista `l` cuyos elementos son enteros y un entero particular x , queremos determinar si x está en `l` o no. Nos interesa analizar la eficiencia en términos de $n = \text{len}(l)$.

En general la forma de resolverlo es con una búsqueda directa en la lista, la cual de modo iterativo se puede hacer de la siguiente forma.

```
def busqueda(l,x):
    esta = False
    for i in l:
        if i == x:
            esta = True
    return esta
```

Como se recorre toda la lista una sola vez, el tiempo de ejecución es $O(n)$. Usando un `while`, se podría hacer que deje de recorrer la lista en caso de encontrar a x , lo cual haría que en algunos casos la función termine antes. Notar que el orden de ejecución en peor caso seguiría siendo $O(n)$. Este peor caso se da por ejemplo x no está en `l`, o también cuando x está pero solamente en el último lugar.

Sin hacer ninguna suposición sobre `l`, lo mejor que se puede hacer es algoritmos de orden lineal. Agreguemos ahora la suposición de que los elementos de `l` están ordenados de menor a mayor. ¿Esto nos permite hacer un algoritmo mejor? Puede ser un buen ejercicio pensarlo un poco.

La respuesta es que sí: si la lista está ordenada se puede hacer un algoritmo con orden de ejecución mejor que lineal. El razonamiento de fondo es el siguiente. Sea y el elemento del medio de l (es importante que sea el del medio). Si y es x , tenemos que x está en la lista, pero en otro caso, mediante la comparación entre x y y , usando que la lista está ordenada podemos descartar una mitad. Es decir, si $x < y$, como la lista está ordenada, sabemos que si x está, necesariamente debe estar a la izquierda de y , mientras que en el otro caso es a la derecha. Esto nos permite hacer el siguiente algoritmo recursivo, que se llama búsqueda binaria.

```
def busqueda_binaria(l,x):
    if len(l) == 0:
        return False
    elif len(l) == 1:
        return l[0] == x
        # retorna True solamente si el elemento es x
    else:
        medio = len(l) // 2
        y = l[medio]
        if x == y:
            return True
        elif x < y:
            # si x está, debe ser a la izquierda de y
            # la sublista a la izquierda de y es l[0:medio]
            return busqueda_binaria(l[0:medio],x)
        else:
            # si x está, debe ser a la derecha de y
            return busqueda_binaria(l[medio:len(l)],x)
```

Lo que hace a este algoritmo muy eficiente es el hecho de que las llamadas recursivas se hacen con listas cuyo largo es $n/2$. Suponiendo el peor caso de que no se encuentra el elemento hasta el final, la recurrencia queda lo siguiente:

$$\begin{aligned}a_0 &= 1 \\a_1 &= 1 \\a_n &= 2 + a_{\frac{n}{2}}\end{aligned}$$

Como en cada paso el n se divide a la mitad, la cantidad de pasos necesarios para llegar al caso base es del orden de $\log(n)$, siendo \log el logaritmo en base 2. En conclusión, este algoritmo de búsqueda es $O(\log(n))$, es decir, de orden logarítmico, lo cual es mucho mejor que lineal.

Hagamos una justificación más formal para el caso particular en el que n es una potencia de dos (en general es más técnico, pero vale el mismo resultado). Tenemos que $n = 2^k$. Hacemos un cambio de variable para trabajar en función de k . A la sucesión con k le llamamos b_k , es decir, $b_k = a_{2^k}$. Como $n/2 = 2^{k-1}$, la recurrencia se convierte en $b_k = 2 + b_{k-1}$. Del caso de base deducimos $b_0 = 0$. Por lo tanto, tenemos que $b_k = 2k$. Deshaciendo el cambio de variable, llegamos a $a_n = 2\log(n)$, lo cual es $O(\log(n))$.

3.4.2. Ordenamiento

Veamos ahora algoritmos de ordenamiento de listas. Comenzamos analizando el algoritmo que ya vimos.

```
def ordenar(l):
    if len(l) == 0:
        return []
    else:
        resto = l[1:len(l)]
        rec = ordenar(resto)
        return insert(l[0], rec)
```

Veamos cual es el orden en el peor caso. Usaremos que ya sabemos que el orden de `insert(x, l)` es lineal. Esto nos permite asumir que con una lista de largo n , `insert` realiza n instrucciones. Formalmente sería algo $\leq Cn$ para una cierta constante C , pero poner la C o no, con el razonamiento que haremos no afectará el orden del resultado final.

Por el paso base sabemos que $a_0 = 1$. Por otra parte, en el caso recursivo se hace una asignación, la llamada recursiva con una lista de largo $n - 1$ y se utiliza la función `insert` con esta lista. Como `insert` es lineal, podemos asumir que la cantidad de instrucciones que realiza con la lista `rec` es $n - 1$ (ya que ese es el largo de la lista). Por lo tanto, $a_n = 1 + a_{n-1} + n - 1 = n + a_{n-1}$. La recurrencia es:

$$\begin{aligned}a_0 &= 1 \\ a_n &= n + a_{n-1}\end{aligned}$$

En este caso la solución es $a_n = 1 + \sum_{i=1}^n i = 1 + n(n+1)/2 = O(n^2)$. Por lo tanto, concluimos que el algoritmo es cuadrático.

La mayoría de los algoritmos que surgen más intuitivamente para ordenar listas son cuadráticos, pero usando recursión ingeniosamente se puede obtener un orden mucho mejor. Para esto antes debemos introducir el algoritmo de mezcla, o *merge*, de listas ordenadas.

La mezcla de dos listas ordenadas consiste en juntar sus elementos en una lista que también esté ordenada. Por ejemplo, si una lista es `[1,4,7]` y la otra es `[4,5]`, el resultado debe ser `[1,4,4,5,7]`. Esta es una función con dos listas como parámetros, `merge(l1, l2)`, que retorna una lista. Un posible algoritmo es el siguiente.

```
def merge(l1, l2):
    # los casos base son cuando alguna es vacía
    # simplemente se retorna la otra
    if len(l1) == 0:
        return l2
    elif len(l2) == 0:
        return l1
    else:
        # las dos tienen al menos un elem
        x = l1[0]
        y = l2[0]
```

```

if x <= y:
    # primero va x y luego mergeo el resto de l1 con l2
    return [x] + merge(l1[1:len(l1)], l2)
else:
    # simétrico a lo anterior
    return [y] + merge(l1, l2[1:len(l2)])

```

El funcionamiento del algoritmo (en términos de las llamadas recursivas) es que se va quitando el elemento más chico de los primeros de ambas listas hasta que una quede vacía. Notar que al ir haciendo esto, si los largos de las listas son n_1 y n_2 , necesariamente se llega a un paso base en a lo sumo $n_1 + n_2$ etapas. Por lo tanto, el orden de este algoritmo en peor caso es $O(n_1 + n_2)$, o $O(n)$ si definimos n como la suma de los largos de las dos listas.

En base a esto podemos hacer el algoritmo de ordenamiento de listas por recursión llamado *merge sort*. El truco es separar la lista a la mitad, llamar recursivamente en cada una de las dos mitades, lo cual retorna dos listas ordenadas, y finalmente mergear estas dos mitades ordenadas para obtener la lista original ordenada.

```

def merge_sort(l):
    if len(l) <= 1:
        return l
    else:
        medio = len(l) // 2
        rec1 = merge_sort(l[0:medio])
        rec2 = merge_sort(l[medio:len(l)])
        return merge(rec1, rec2)

```

Analicemos un poco el orden en tiempo de ejecución. Las llamadas recursivas son con listas de la mitad del largo, por lo que tienen $a_{n/2}$ instrucciones cada una y las listas resultantes también son de largo $n/2$. Como la cantidad de instrucciones de `merge` es la suma de los largos de las listas, tenemos que el `merge(rec1, rec2)` hacer $n/2 + n/2 = n$ instrucciones. En conclusión, tenemos $a_n = 2a_{n/2} + n$.

Vamos a resolverlo nuevamente cuando $n = 2^k$ haciendo el cambio de variable. Obtenemos $b_k = 2b_{k-1} + 2^k$. Aquí es un poco más difícil, pero la solución es $b_k = k2^k$. Se puede verificar que esto cumple la recurrencia. Deshaciendo el cambio de variable llegamos a que $a_n = n \log(n)$, por lo que el merge sort es $O(n \log(n))$. Notar que esto es mucho mejor que $O(n^2)$, por ejemplo, si $n = 1,000,000$, entonces $n^2 = 1,000,000,000,000$, mientras que $n \log(n) \sim 30,000,000$.

3.5. Algoritmos exponenciales

Decimos que un algoritmo es exponencial en tiempo de ejecución si su orden es una función exponencial, por ejemplo $O(2^n)$ o $O(3^n)$. Por otra parte, cuando un algoritmo tiene un orden $O(n^e)$ para algún $e \in \mathbb{N}$, decimos que es polinomial. Como criterio general, un algoritmo polinomial es aplicable en la práctica, mientras que uno exponencial no lo es.

Primer ejemplo ingenuo

Al escribir funciones recursivas, si el código tiene más de una llamada recursiva, existe el riesgo de que el algoritmo sea exponencial. A modo de ejemplo, consideremos el siguiente algoritmo que dado n calcula 2^n , basándose en la propiedad de que $2^n = 2^{n-1} + 2^{n-1}$.

```
def expo(n):
    if n == 0:
        return 1
    else:
        return expo(n-1) + expo(n-1)
```

veamos la recurrencia del tiempo que lleva la ejecución. Por el caso base tenemos que $a_0 = 1$. Por otra parte, en el paso recursivo, se llama dos veces `expo(n-1)`, por lo que podríamos escribir $a_n = 2a_{n-1}$. Como la recurrencia multiplica por 2 en cada paso, tenemos que $a_n = 2^n$ y por lo tanto el algoritmo es $O(2^n)$, por lo que es exponencial.

Si se prueba en la computadora, para valores chicos de n termina en seguida, pero desde al rededor de 20 en adelante (dependiendo de la computadora) empieza a tardar un poco y al seguir aumentando tarda cada vez más. Puede estar bueno ver como al aumentar el n de a 1, el tiempo que tarda más o menos se duplica. Este tipo de aumento en el tiempo de ejecución es lo que caracteriza los algoritmos exponenciales. Con un valor de $n = 50$ la ejecución ya podría tardar días.

Es importante entender que en un algoritmo como este, en el paso recursivo, la función `expo(n-1)` se ejecuta dos veces, en lugar de hacerse una vez sola y luego usar dos veces el resultado, como uno intuitivamente podría pensar (la computadora no piensa, simplemente ejecuta las instrucciones). Es equivalente a si escribiéramos el programa así:

```
def expo(n):
    if n == 0:
        return 1
    else:
        rec1 = expo(n-1)
        rec2 = expo(n-1)
        return rec1 + rec2
```

Se hace recursión para calcular `rec1` y luego al calcular `rec2`, se vuelve a hacer exactamente lo mismo, sin reutilizar lo que ya se calculó. Para que sí reutilice lo calculado en `rec1` para el valor de `rec2`, habría que escribir el código de modo correspondiente, por ejemplo:

```
def expo(n):
    if n == 0:
        return 1
    else:
        rec1 = expo(n-1)
        rec2 = rec1
        return rec1 + rec2
```

En este caso el algoritmo deja de ser exponencial.

3.5.1. Funciones con un parámetro. Ejemplo Fibonacci

En el ejemplo anterior fue sencillo evitar el carácter exponencial. En un caso como ese, las formas exponenciales de escribirlo se puede considerar como un error que alguien haría solamente si no sabe que se está ejecutando dos veces exactamente lo mismo. Sin embargo, hay otros algoritmos recursivos exponenciales en los que no es tan sencillo evitar el carácter exponencial. Consideremos por ejemplo un algoritmo que calcula la sucesión de Fibonacci, que se define como $(f_n)_{n \in \mathbb{N}}$ tal que $f_0 = f_1 = 1$ y $f_n = f_{n-1} + f_{n-2}$ si $n > 1$.

```
def fibo(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

En este caso, la recurrencia que queda para la cantidad de instrucciones es (un poco graciosamente) la misma sucesión de Fibonacci: $a_0 = a_1 = 1$ y $a_n = a_{n-1} + a_{n-2}$, la cual tiene un crecimiento exponencial.

También podemos observar que a lo largo de las llamadas recursivas se repiten cálculos. Para la explicación supongamos que el algoritmo hace $x = \text{fibo}(n-1)$, luego $y = \text{fibo}(n-2)$ y luego $\text{return } x + y$, lo cual es equivalente al código de arriba. Si $n > 2$, al ejecutar $x = \text{fibo}(n-1)$, para calcular $\text{fibo}(n-1)$ se deben calcular $\text{fibo}(n-2)$ y $\text{fibo}(n-3)$, sin embargo, al ejecutar $y = \text{fibo}(n-2)$, se vuelve a calcular $\text{fibo}(n-2)$, lo cual ya había aparecido en el cálculo de $\text{fibo}(n-1)$. Cuando aparecen cálculos repetidos en funciones recursivas como en este caso, es común que el algoritmo resulte ser exponencial.

Veamos dos formas de resolver esto, la primera es calculando un vector y la segunda es usando una lista de valores ya calculados. En ambos casos la idea de fondo es la función recursiva anterior. Ambas soluciones logran que no se repitan cuentas y el algoritmo sea de orden polinomial.

Solución con vector. En lugar de partir del valor n e ir aplicando la recursión con valores más chicos, vamos a ir llenando iterativamente un vector con valores de la sucesión, partiendo desde 0, hasta llegar a n . Es decir, en lugar de ir desde n hacia atrás hasta llegar a los casos base, partimos desde los casos base y basándonos en el caso recursivo vamos llenando el vector desde ahí hasta llegar a n . Nuestro vector es una lista de $n + 1$ elementos. El algoritmo es el siguiente. Los comentarios explican qué se va haciendo.

```
def fibo_vector(n):
    if n == 0 or n == 1:
        return 1
    else:
        # creamos el vector de n+1 elementos
        # que al principio es una lista de ceros
        vector = []
        for i in range(n+1):
            vector.append(0)
        # ahora vector es un vector con n+1 0s.
        # La idea es que cada entrada del vector se
```

```

# corresponda con un término de Fibonacci,
# es decir, que vector[i] sea f_i
# de modo que al final f_n sea vector[n] y retornar eso.

#primero hagamos que los dos primeros lugares sean 1,
#de acuerdo al los casos base
vector[0] = 1
vector[1] = 1

#ahora, vamos calculando los otros términos del vector,
#basándonos en el caso recursivo.
#Comenzamos desde el de índice 2.
for i in range(2,n+1):
    vector[i] = vector[i-1] + vector[i-2]

#con esto hicimos que cada entrada de vector sea el
#término correspondiente de la sucesión de Fibonacci.
# es decir, vector = [f_0,f_1,...,f_n]

return vector[n]

```

Si bien la idea proviene del algoritmo recursivo anterior, notar que este es iterativo. La idea de esta estrategia de solución en general es la siguiente:

1. Creamos una lista v de $n + 1$ elementos, al principio llena de ceros, con el objetivo de luego hacer que para cada $i \leq n$, se tenga que $v[i]$ es f_i .
2. De acuerdo al caso base, hacemos que $v[0]$ y $v[1]$ valgan 1.
3. De acuerdo al caso recursivo, vamos iterando desde $i = 2$ hasta $i = n$ haciendo que $v[i]$ sea $v[i-1] + v[i-2]$. Esto funciona porque en cada iteración, debido a los casos base y las iteraciones previas, en los lugares $i-1$ e $i-2$ el vector ya tiene los valores correctos de la sucesión de Fibonacci.
4. Luego de la iteración se cumple que $v[i]$ es f_i para cada $i \leq n$, por lo que en particular $v[n]$ vale f_n , que es lo que se debía calcular.

Solución con lista de valores ya calculados. Otra alternativa, que sigue usando recursión, es utilizar una lista de valores ya calculados. La lista de valores calculados va a ser una lista de pares $[[n_1, v_1], [n_2, v_2], \dots, [n_k, v_k]]$, donde sabemos que con entrada n_i el resultado es v_i . La idea del algoritmo es que si la entrada está en la lista, usamos el valor sin volver a calcularlo y en otro caso, lo calculemos y lo agregamos a la lista. La función va a tener dos parámetros: `n` y `calculados`, donde el primero es el valor en el que queremos calcular la sucesión y el segundo es una lista de entradas para las que ya sabemos el resultado. La salida va a ser una lista con dos cosas, `[valor, calculados]`, donde `valor` es el valor de la sucesión en n y `calculados` es la lista de calculados, que puede estar igual que como vino o tener también nuevas entradas.

```

def fibo_lista(n, calculados):
    # si n está en la lista de valores conocidos,
    # no lo volvemos a calcular
    for x in calculados:
        # x es [a,b]
        if x[0] == n:
            # retornamos el valor de la sucesión y la lista de
            # calculados, que no cambió
            return [x[1], calculados]

    # si llegamos hasta acá, quiere decir que n no está en la
    # lista de calculados. Hay que calcular el valor en n,
    # agregarlo a la lista y retornar ambas cosas
    if n == 0 or n == 1:
        #agregamos que vale 1 a la lista de calculados
        calculados.append([n,1])
        return [1, calculados]
    else:
        rec1 = fibo_lista(n-1, calculados)
        valor1 = rec1[0]
        calculados = rec1[1] # pueden haberse calculado nuevos valores

        rec2 = fibo_lista(n-2, calculados)
        valor2 = rec2[0]
        calculados = rec2[1]

        #el valor en n es el siguiente
        valorn = valor1 + valor2

        #agregamos el valor en n a la lista de calculados
        calculados.append([n,valorn])
        return [valorn, calculados]

```

Para utilizar esta función, se comienza con una lista vacía de valores calculados, dejando que se vaya llenando a lo largo de la ejecución. Luego, lo que retorna es una lista cuyo primer elemento es el valor en el n que pasamos.

```

>>> calculados = []
>>> res = fibo_lista(5,calculados)
>>> res[0]
8

```

Tanto la solución con vector como la solución con lista de valores ya calculados son polinomiales. Sin embargo, puede haber una diferencia de orden. La solución con el vector, como simplemente lo recorre una vez, es de orden lineal, mientras que la solución por lista de valores ya calculados puede ser de orden cuadrático, ya que en cada llamada recursiva debe hacer una búsqueda en una lista, lo cual puede llevar tiempo lineal.

3.5.2. Funciones con dos parámetros. Ejemplo combinaciones

Veamos ahora un ejemplo con una función recursiva que calcula combinaciones. En este caso ocurre una situación muy similar a la de la sucesión de Fibonacci, pero ahora con dos parámetros.

Recordamos que las combinaciones entre n y k , denotadas C_k^n , denotan la cantidad de formas de elegir k elementos de un conjunto de n sin repetir y sin importar el orden. Si $k > n$, $C_k^n = 0$ por definición. Cuando $k \leq n$, se cumple la siguiente fórmula:

$$C_k^n = \frac{n!}{(n-k)!k!}$$

Esta fórmula no es muy práctica para calcular las combinaciones automáticamente, pues tanto el numerador como el denominador pueden ser números muy grandes. Muchas veces puede pasar que varios términos se cancelen y el resultado sea chico, pero que si calculamos el numerador y el denominador sin cancelar nada, tengamos que trabajar con números muy grandes. Un ejemplo es C_1^{100} , lo cual da 100, pero si usamos la fórmula sin cancelar nada, para el numerador debemos calcular $100!$.

Hay una conocida recurrencia, llamada regla de Stiffel, que para $n, k > 0$ dice lo siguiente:

$$C_k^n = C_k^{n-1} + C_{k-1}^{n-1}$$

Teniendo en cuenta que si $C_0^n = 1$ para todo n y que $C_k^0 = 0$ si $k > 0$, que son cosas que sirven como casos base, podemos hacer el siguiente algoritmo.

```
def combinaciones(n,k):
    if k == 0:
        return 1
    elif n == 0:
        return 0
    else:
        return combinaciones(n-1,k) + combinaciones(n-1, k-1)
```

Al igual que ocurre en el caso de la sucesión de Fibonacci, este algoritmo es exponencial. Se puede ver que aquí también hay casos en los que algunas cuentas se repiten. Por otra parte, si ejecutamos con $n = k$ aumentándolos, el tiempo de ejecución crece de modo exponencial.

Vamos a aplicar las mismas soluciones que para la sucesión de Fibonacci, con la diferencia que en el primer enfoque en lugar de utilizar un vector usaremos una matriz, ya que ahora tenemos dos parámetros.

Solución con matriz. La idea ahora es crear una matriz de $n + 1$ filas y $k + 1$ columnas e ir llenando de modo que cada entrada (i, j) quede con el valor C_j^i y por lo tanto, en particular, la entrada (n, k) valga C_k^n . La estrategia es la misma. Creamos la matriz, inicializamos los lugares que se deducen de los casos base y luego vamos iterando según el caso recursivo.

El punto que se vuelve un poco más complejo es el del orden para ir iterando. Lo importante es utilizar un orden en el que cuando vamos a calcular la entrada (i, j) , tanto la $(i - 1, j)$ como la $(i - 1, j - 1)$, que son las que aparecen en la recurrencia, ya se hayan calculado. Una forma de hacer esto es ir calculando las filas de arriba a abajo, cada una

de izquierda a derecha, es decir primero usar $i = 1$ y calcular toda la fila con j creciente: $(1,1), (1,2), \dots, (1,k)$; luego poner $i = 2$ y calcular toda la fila con j creciente: $(2,1), (2,2), \dots, (2,k)$; así sucesivamente hasta llegar a $i = n$ y $j = k$. Notar que se comienza con $i = 1$ y $j = 1$ porque la fórmula de Stiffel requiere que sean mayores a 0. Las entradas en los que alguno es 0 se completan al principio según el caso base.

```
def combinaciones_vector(n,k):
    if k == 0:
        return 1
    elif n == 0:
        return 0
    else:
        #creamos la matriz, que es una lista de listas
        matriz = []
        #va a tener n+1 filas
        for i in range(n+1):
            #creamos la fila iésima, que debe tener k+1 lugares
            fila = []
            for j in range(k+1):
                fila.append(0)
            #agregamos la fila a la matriz
            matriz.append(fila)
        #la matriz queda con todas las entradas valiendo 0
        #se accede a la entrada (i,j) con matriz[i][j],
        #pues es una lista de listas

        #usando el caso base, inicializamos los lugares
        #con i=0 o con j=0
        for i in range(n+1):
            # caso k==0 del algoritmo recursivo
            matriz[i][0] = 1
        for j in range(1,k+1):
            # caso n==0 del algoritmo recursivo.
            # notar que no se aplica cuando k es 0
            matriz[0][j] = 0

        #Ahora llenamos el resto de la matriz según el
        #caso recursivo, siguiendo el orden explicado
        #en el párrafo previo
        for i in range(1,n+1):
            for j in range(1,k+1):
                matriz[i][j] = matriz[i-1][j] + matriz[i-1][j-1]

        #retornamos el valor del lugar (n,k)
        return matriz[n][k]
```

Un factor importante a tener en cuenta al hacer esto es que en la matriz las filas sean

objetos lista distintos, incluso al principio tengan los mismos valores (ver sección 2.4.1). Como lo hicimos en este caso está bien, porque para cada fila de la matriz armamos una lista nueva a partir de la lista vacía. Por otra parte, si pensando en el hecho de que todas las filas de la matriz son iguales hiciéramos lo siguiente:

```
matriz = []

#creamos una única fila con k+1 ceros
fila = []
for j in range(k+1):
    fila.append(0)

#agregamos esa fila n+1 veces a la matriz
for i in range(n+1):
    matriz.append(fila)
```

entonces quedaría una matriz de $n+1$ filas y $k+1$ columnas en las que todas las entradas valen 0, pero en realidad todas las filas, además de ser iguales, serían *la misma lista*. Si modificáramos una de estas, la misma modificación ocurriría con todas las otras.

Solución con lista de valores ya calculados. Aquí la diferencia es que como la función tiene dos parámetros, las entradas de la lista de valores ya calculados serán de la forma $[[n, k], v]$. Aparte de eso, la idea es la misma.

```
def combinaciones_lista(n,k,calculados):
    for x in calculados:
        # x es [[a,b],c]
        if x[0][0] == n and x[0][1] == k:
            return [x[1], calculados]

    if k == 0:
        calculados.append([[n,k],1])
        return [1, calculados]
    elif n == 0:
        calculados.append([[n,k],0])
        return [0, calculados]
    else:
        rec1 = combinaciones_lista(n-1,k,calculados)
        valor1 = rec1[0]
        calculados = rec1[1]

        rec2 = combinaciones_lista(n-1,k-1,calculados)
        valor2 = rec2[0]
        calculados = rec2[1]

        valor = valor1 + valor2
        calculados.append([[n,k],valor])
        return [valor, calculados]
```

Capítulo 4

Modularidad

Un motivo muy importante por el que la computación ha avanzado es el enfoque de utilizar programas ya implementados, sin necesidad de tener presentes los detalles de cómo se hicieron, para programar otros más complejos. A este principio se le llama **modularidad**. Un ejemplo de esto es la definición de funciones, que luego se pueden utilizar como parte de distintos programas. El beneficio que esto da es que después de que la función está definida, se la puede usar sabiendo para qué sirve, sin necesidad de preocuparse por los detalles de cómo está implementada.

Si bien las funciones permiten reutilizar código, cuando el programa es muy complejo conviene además definir distintos **módulos**. Esto lo que permite es separar las distintas partes del programa en distintos archivos, en lugar de tener todo junto en uno solo. Esto puede ser deseable por una parte para evitar trabajar con archivos demasiado extensos y por otra para tener por separado partes del código que conceptualmente cumplen distintas funciones.

La modularidad, además de permitir que uno ordene mejor su código, abre la posibilidad de utilizar cosas hechas por otras personas en programas propios. Las **bibliotecas** son módulos programados por otras personas o por organizaciones que contienen funcionalidades con algún fin específico y que se pueden incluir en los programas propios. Hay por ejemplo bibliotecas para trabajar con datos temporales, para procesar datos audiovisuales o para hacer álgebra lineal.

En general las bibliotecas tienen cierta documentación oficial en internet, que explica las funcionalidades. En varios casos esta documentación está solamente en inglés.

Las principales bibliotecas de python son proyectos de código abierto, o *open source*. Esto significa que además de que se la pueda usar gratuitamente, el código interno se puede descargar gratuitamente y modificarlo a gusto (aunque en general hay que entender muy bien el funcionamiento para poder hacer esto último). Estos proyectos se basan en gran parte en aportes voluntarios de programadores de distintas partes del mundo.

4.1. Algunos otros elementos de python

Hay elementos de python que no se introdujeron en el capítulo 2 pero que se utilizan comúnmente. En esta sección se presentan brevemente algunos de ellos. Los otros que vayan surgiendo se irán presentando sobre la marcha. Cuando sea necesario, se recomienda complementar buscando información en internet, ya sea en la documentación oficial de

python o en otras fuentes de información que se encuentren. Por lo general la documentación oficial es lo más preciso pero difícil de leer, por lo que suele ser buena idea usar ambas cosas.

4.1.1. Tuplas y diccionarios

Las tuplas y los diccionarios son otras estructuras de datos compuestas, al igual que las listas. Con las listas se puede hacer todo, pero en algunas situaciones es más fácil o intuitivo usar estos otros tipos de datos.

Las **tuplas** representan vectores de objetos, como $(x_1, \dots x_n)$ y la notación es exactamente esa. Para hacer una tupla con dos objetos **x** e **y** se escribe **(x,y)**. Con más objetos es análogo. Se puede usar indizado para acceder a los elementos, pero no se los puede cambiar. En este sentido las tuplas son un tipo de objeto inmutable.

```
>>> x = (1,2)
>>> type(x)
<class 'tuple'>
>>> x[0]
1
>>> x[1]
2
>>> x[0] = 2 # No se puede
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

También se pueden crear tuplas sin poner los paréntesis, solamente separando con las comas.

```
>>> x = 1, 2
>>> type(x)
<class 'tuple'>
>>> x[0]
1
>>> x[1]
2
```

Un uso común de las tuplas (escritas sin paréntesis) es el de hacer asignaciones múltiples. Consideramos el siguiente ejemplo:

```
>>> x, y = 4, 5
>>> x
4
>>> y
5
```

Otro uso común de las tuplas, también omitiendo los paréntesis, es el de una función que retorne varios objetos, haciendo que en realidad retorne una tupla. En el capítulo 2 hicimos esto retornando listas, pero con tuplas queda escrito de una forma más sencilla. Por ejemplo podemos hacer una función que retorna un número y el doble:

```
def f(n):
    return n, 2*n
```

y luego para llamar la función y obtener los dos resultados se hace de la siguiente forma:

```
>>> x, y = f(5)
>>> x
5
>>> y
10
```

Los **diccionarios** son otro tipo de datos compuesto. En este caso son pares de clave y valor, donde cada valor está indizado por su clave. La sintaxis es {clave1:valor1, clave2:valor2,..., claven:valorn}. Los valores pueden ser de cualquier tipo, mientras que las claves deben ser inmutables (por ejemplo números, strings o tuplas, pero no listas).

```
>>> d = {1:"aaa", 2:"bbb", "hola":5, (2,3):9}
>>> type(d)
<class 'dict'>
>>> d[1]
'aaa'
>>> d["hola"]
5
>>> d[(2,3)]
9
>>> d[0] #Si accedemos por una clave que no está, da error
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 0
>>> d[0] = "ccc" #Pero podemos agregar algo con esa clave
>>> d[0] #Ahora sí está
'ccc'
```

Ejercicio 4.1.1. Investigar cómo se hace para iterar por un diccionario.

4.1.2. Funciones como objetos y expresiones lambda

En python las funciones se pueden utilizar como objetos. De hecho son un tipo de datos, al igual que los enteros o los strings. Si ejecutamos lo siguiente:

```
def f(n):
    return n

print(type(f))
```

la salida es <class 'function'>. Se puede asignar una función a una variable, por ejemplo:

```
def f(n):
    return n
```

```
x = f
```

Después de eso, dado un n , podemos escribir $x(n)$ y hará lo mismo que $f(n)$.

Se puede también pasar una función como parámetro a otra función. Por ejemplo, la siguiente es una función que dada otra función y un valor, retorna la otra función compuesta con ella misma aplicada al valor.

```
def comp(g,x):
    return g(g(x))
```

Notar que el primer parámetro, g , se usa como una función. En caso de que se pase otra cosa, dará un error.

Supongamos que definimos las siguientes otras dos funciones:

```
def f1(n):
    return n + 2
```

```
def f2(n):
    return 2*n
```

En caso de hacer `comp(f1,n)`, se retornará el resultado de aplicar dos veces $f1$ a n , mientras que al hacer `comp(f2,n)` ocurrirá lo mismo pero con $f2$. Por ejemplo:

```
>>> comp(f1,2)
6
>>> comp(f2,2)
8
```

Por otra parte, están las expresiones lambda que permiten definir una función en una sola línea. Se basa en el cálculo lambda (página de wikipedia). En cálculo lambda, se escribe un λ , la variables de la función, o las variables, luego un punto y luego lo que retorna la función. Por ejemplo, la función que dado x retorna $x + 1$ se escribe como $\lambda x. x + 1$. En python en lugar del λ se escribe `lambda` (que es una palabra especial, al igual que por ejemplo `if` o `for`) y en lugar del punto va un `:`. Por ejemplo, definamos en una línea una función que duplica un número y utilicémosla.

```
>>> f = lambda n: 2*n
>>> f(4)
8
>>> comp(f,4)
16
```

Si queremos definir una función con más de un parámetro, ponemos las variables separadas por comas.

```
>>> g = lambda x, y: x+y
>>> g(3,3)
6
```

Por otra parte, las expresiones lambda se pueden usar para pasar una función como parámetro, sin necesariamente guardarla antes en una variable. Por ejemplo se puede hacer:

```
>>> comp(lambda n: 2*n, 4)
16
```

Que es equivalente a lo que se hizo en el ejemplo de arriba, en el que la función que duplica se guarda en una variable `f` antes de pasarla a `comp`.

4.1.3. Manejo básico de archivos

En python se tiene la función `open()` que permite abrir un archivo y utilizarlo, ya sea para leer sus contenidos o escribirle cosas. El primer parámetro que se pasa es el nombre del archivo a abrir, incluyendo la extensión (`.txt`, `.jpg`, `.py`, etc). El segundo parámetro, que es opcional, indica si vamos a leer o escribir el archivo, entre otras posibilidades. Para leer se pone `"r"` y para escribir `"w"`. Si este parámetro no se ingresa, por defecto se abre en modo lectura. Luego de terminar de usar el archivo, hay que cerrarlo con el método (función del archivo) `close()`.

En caso de archivo abierto en modo lectura, podemos obtener el contenido con el método `read()`. En caso de escritura, podemos escribir su contenido (borrando lo que había antes) con `write(contenido)`.

Supongamos que tenemos un archivo `texto.txt`, el cual queremos abrir para leer sus contenidos e imprimirlo en la terminal. Lo podemos hacer así:

```
archivo = open("texto.txt")
# Se abre por defecto en modo lectura,
# igual que con open("texto.txt", "r")

# la función read retorna el contenido del archivo
contenido = archivo.read()

print(contenido)

# cerramos el archivo
archivo.close()
```

ahora supongamos que queremos escribir el archivo. Lo podemos hacer de la siguiente forma.

```
archivo = open("texto.txt", "w")
archivo.write("Nuevo contenido del archivo.")
archivo.close()
```

Tener en cuenta que esto sobrescribe el contenido anterior del archivo. En caso de que no exista un archivo con ese nombre, se lo crea.

Para más información, se puede consultar la documentación de python u otras fuentes en internet, como por ejemplo www.w3schools.com/python/python_file_handling.asp.

4.2. Módulos

Un módulo es en realidad un archivo de python con terminación «.py», como los que venimos escribiendo hasta ahora. Los contenidos del módulo son las cosas que se definen adentro, como por ejemplo funciones o variables.

Cada módulo tiene un nombre, que es el nombre del archivo excepto por el «.py». Para utilizar el módulo en un programa escrito en otro archivo, o desde la terminal, se utiliza la instrucción `import` seguida del nombre del módulo. Después de eso, los contenidos definidos dentro del módulo se pueden acceder agregándoles antes el nombre del módulo y un punto.

A modo de ejemplo, supongamos que tenemos el siguiente código dentro de un archivo llamado `miModulo.py`, el cual está en una carpeta llamada `Módulo`.

```
# es_par(n) retorna True si es par y False si es impar.
def es_par(n):
    return n % 2 == 0

#una variable que vale 3
aprox_pi = 3
```

Para importarlo como módulo ejecutando python desde una terminal en esa carpeta hay que escribir `import miModulo` y luego podemos acceder a la función y la variable con `miModulo.es_par` y `miModulo.aprox_pi`. Por ejemplo:

```
PS C:\Users\...\Módulo> python
>>> import miModulo
>>> miModulo.es_par(6)
True
>>> miModulo.aprox_pi
3
```

Por otra parte, dentro de la misma carpeta podemos escribir otro programa que importe el módulo. El contenido del programa podría ser el siguiente:

```
import miModulo

def no_es_par(n):
    return not miModulo.es_par(n)

print(no_es_par(miModulo.aprox_pi))
```

Si además de las definiciones hay instrucciones dentro del módulo, al importarlo estas instrucciones se ejecutan. De hecho, la definición de variables dentro del módulo se puede considerar como un caso particular de esto.

Cuando un módulo es importado se genera una carpeta `_pycache_`, en la que se guarda el módulo compilado. Esto permite ahorrar tiempo si se lo vuelve a importar.

La instrucción `import` se puede usar también de las siguientes formas.

1. `import modulo as nombre`. Esta forma permite cambiar el identificador del módulo dentro del programa, es decir, lo que se debe escribir antes del punto para acceder a los contenidos. Por ejemplo, si ejecutamos `import miModulo as miM`, luego para acceder a los contenidos escribiríamos `miM.es_par()` y `miM.aprox_pi`.
2. `from modulo import x1,...,xn`. Esta sintaxis permite importar ciertas cosas del módulo y no necesariamente todos sus contenidos. Además, los elementos importados se referencian directamente por su nombre, sin tener que poner el nombre del módulo antes. Por ejemplo, si hacemos `from miModulo import es_par`, estamos importando solamente la función, la cual se puede usar con `es_par()`, sin tener que poner antes «`miModulo.`». Por otra parte, podemos ejecutar `from miModulo import es_par, aprox_pi`, e importar las dos cosas de tal modo que se pueden usar con `es_par()` y `aprox_pi`. Para importar todos los contenidos de un módulo de este modo, se puede hacer `from modulo import *`.

4.3. Bibliotecas

Las bibliotecas permiten extender el lenguaje con más funcionalidades. Son cosas que ya fueron implementadas por otra gente u organizaciones y están instaladas en el lenguaje, o se pueden instalar. Al momento de usarlas, es muy similar al caso de los módulos. Cuando una biblioteca tiene distintas partes con distintos nombres, se les llama también módulos a cada una de estas.

Python viene con una biblioteca ya instalada, la cual se llama *biblioteca estándar*. Es muy extensa, conteniendo funcionalidades muy variadas. La documentación oficial es la siguiente: <https://docs.python.org/3.14/library/index.html>. Hay secciones para los distintos módulos de la biblioteca.

Por ejemplo está el módulo `math`, que se mencionó en la sección 2.2.3. Como ya vimos en la sección mencionada, se lo puede importar escribiendo `import math` y luego usar por ejemplo `math.sqrt()` y `math.round()`. Se encuentra una lista completa de lo que incluye en docs.python.org/3/library/math.html.

Por otra parte, hay muchas otras bibliotecas que se pueden instalar aparte y tienen otras funcionalidades. Cada una de ellas suele tener su documentación oficial disponible en internet. Por ejemplo está la biblioteca `numpy`, la cual vemos más adelante, cuya documentación se puede encontrar en el sitio numpy.org/.

Para instalar gran parte de las bibliotecas se puede utilizar el comando `pip install` con el nombre de la biblioteca desde una terminal. Por ejemplo, para instalar `numpy` basta abrir una ventana de comandos y ejecutar `pip install numpy`. Al hacer eso, automáticamente se descarga y se instala. Las bibliotecas más usadas se pueden instalar con `pip` de esta forma.

Dada una biblioteca (o módulo de biblioteca) con nombre `nombre` que está instalada (ya sea porque sea parte de la biblioteca estándar o porque la instalamos aparte), se la importa con `import nombre`, pudiendo luego acceder a las funcionalidades igual que si fuera un módulo. También se puede usar las otras formas de `import` que se presentaron al final de la sección 4.2.

A veces las bibliotecas tienen distintos módulos adentro, que se referencian poniendo el nombre de la biblioteca, luego un punto y luego el nombre del módulo. Por ejemplo,

numpy tiene un módulo de álgebra lineal llamado `numpy.linalg`.

A veces se usan términos como «biblioteca», «módulo» o «paquete» de modo más o menos equivalente.

En el resto del capítulo estudiaremos un poco algunas bibliotecas. En el resto de esta sección hay ejemplos de biblioteca estándar y de otras bibliotecas de usos variados (concretamente, procesar archivos PDF e imágenes). En las secciones posteriores, se introducen algunas bibliotecas con funcionalidades específicamente matemáticas. Hay muchas más bibliotecas que las que mencionamos aquí. Además, las que aparecen tienen más (la mayoría muchas más) cosas que lo que vemos aquí.

Sobre las documentaciones

Las documentaciones suelen tener explicaciones conceptuales del funcionamiento de la biblioteca en cuestión, explicaciones detalladas de las distintas funcionalidades y ejemplos. Para cada función disponible suele haber una sección en la que se explica precisamente lo que hace, qué parámetros recibe y que cosas retorna. Es común que haya mucha más información que lo que se necesita, por lo que en general lo mejor es a priori ver solo lo básico e ir profundizando conforme vaya siendo necesario.

Dentro de la explicación de cada función también puede haber más información de la que se requiere, por ejemplo con parámetros opcionales o con datos *extra* que se retornan. Ahí también suele ser recomendable leer lo básico, solo prestarle atención a los parámetros opcionales que se necesiten e ignorar las cosas que se retornan que no vamos a usar. Ver por ejemplo docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad, que es una función de scipy que usaremos más adelante. Esta muestra más de diez parámetros, pero solo usaremos los primeros tres. Por otra parte, presenta cuatro cosas distintas que se retornan, de las cuales en general solo usaremos la primera. Por lo tanto, puede alcanzar con leer la descripción de la función, la de los primeros tres parámetros y la del primer dato que retorna, además de ver el primero de los ejemplos de abajo.

Tanto para la biblioteca estándar como para otras, la documentación oficial en general es muy completa y precisa, pero puede ser difícil de leer. Muchas veces buscando en internet se encuentran tutoriales o guías más sencillas, que permiten comenzar a usar las funcionalidades más rápidamente.

4.3.1. Ejemplos de la biblioteca estándar

Consideremos por ejemplo la biblioteca `time`, que provee funcionalidades vinculadas con datos temporales. La documentación es <https://docs.python.org/3/library/time.html> (se puede acceder también desde la página de la documentación de la biblioteca estándar).

Una funcionalidad que suele ser útil es la función `time.time()`, que retorna un número en punto flotante que indica el momento actual, llamado *timestamp*. Este número representa el tiempo que pasó desde cierto momento en particular medido en segundos, incluyendo parte fraccionaria. Se la puede usar para medir precisamente el tiempo que tarda en ejecutarse cierto código. Por ejemplo, si queremos ver cuánto le lleva a python hacer un millón de sumas, podemos hacer lo siguiente:

```
import time
```

```

tiempoI = time.time()
cont = 0
for i in range(1000000):
    cont = cont + 1
tiempoF = time.time()
print(tiempoF - tiempoI)

```

Consideremos ahora la biblioteca `datetime`, que permite también trabajar con fechas. Como se puede ver en la documentación (docs.python.org/3/library/datetime.html), incluye varios tipos de datos, para trabajar con fechas, con horas, con fecha y hora, y con diferencias de tiempo, entre otras cosas. Un ejemplo sencillo es el de crear un objeto de tipo `date` a partir de año, mes y día, para luego utilizando la función `weekday` determinar qué día de la semana fue.

```

import datetime as dt

fecha = dt.date(2000,1,1) # 1 de enero de 2000
print(fecha.weekday())

```

Lo que se imprime es un número del 0 al 6, donde 0 es lunes y 6 es domingo. Esto se especifica en la documentación, en la parte del método `date.weekday()`. Un método es una función de un objeto de cierto tipo, en este caso de tipo `date`.

Los métodos son un concepto de programación orientada a objetos, tema en el que no ahondamos. Sí es bueno explicar algo de la terminología que aparece, ya que es muy común. Una clase es como un tipo de datos y un objeto es un elemento perteneciente a cierta clase. Un atributo es como una variable asociada a una clase o a un objeto. Un método es como una función asociada a una clase o a un objeto.

Otra función (formalmente, método de la clase `date`) útil es `date.fromtimestamp(t)`. Esto retorna una fecha a partir de un timestamp, como los que retorna la función `time()` de la biblioteca `time`. Por ejemplo si hacemos lo siguiente:

```

import datetime as dt
import time

tiempoActual = time.time()
fecha = dt.date.fromtimestamp(tiempoActual)

```

la variable `fecha` queda con la fecha actual. Hay un método `date.today()` que retorna justamente la fecha actual, pero esto de obtener una fecha a partir de un timestamp puede tener otras utilidades.

Esta biblioteca tiene también objetos de tipo `datetime`, que contienen información de fecha y hora. Esta clase también tiene un método `today`, en este caso `datetime.today()`, que retorna la fecha y hora actual.

Veamos ahora un ejemplo con la biblioteca `os`, cuyo nombre es por la sigla de *operating system* (sistema operativo en inglés). Esta biblioteca permite interactuar con el sistema operativo, permitiendo hacer varias cosas. Vamos a usarla para obtener información sobre archivos. Documentación: <https://docs.python.org/3/library/os.html>.

Con la función `os.stat(nombre)`, donde `nombre` es el nombre de un archivo, se puede obtener información sobre el archivo, incluyendo la cantidad de bytes que ocupa y la última fecha de modificación. Lo que esto retorna es un objeto de la clase `stat_result`, lo cual contiene muchos atributos que se indican en la documentación. Para acceder a un atributo hay que poner el nombre del objeto, un punto y el nombre del atributo. Uno de los atributos es `st_size`, que contiene el tamaño en cantidad de bytes. Otro de los atributos es `st_mtime`, que es el timestamp de la fecha de modificación (mismo formato que lo que retorna la función `time()` del módulo `time`). A modo de ejemplo:

```
import os
import datetime as dt

datos = os.stat('programa.py') # infor del archivo programa.py
print(datos.st_size) # esoacio que ocupa en bytes
print(datos.st_mtime) # timestamp de última modificación
fH = dt.datetime.fromtimestamp(datos.st_mtime) # fecha y hora
print(fH)
```

En la penúltima línea, utilizando el módulo `datetime` obtenemos la fecha y hora de última modificación a partir del timestamp.

Otra función que puede ser útil es `os.listdir()`, que retorna una lista con los nombres de todos los archivos en el directorio actual. Por ejemplo, el siguiente programa cuenta la cantidad de bytes ocupada por todos los archivos en la carpeta juntos.

```
import os

cont = 0
for nombre in os.listdir():
    datos = os.stat(nombre)
    cont = cont + datos.st_size
print(cont)
```

Hagamos algo un poco más difícil. Supongamos que estamos en una carpeta con imágenes y queremos cambiar los nombres de todas las imágenes por números de 1 en adelante, en orden de momento de última modificación. Supongamos que las imágenes son en formato jpg. Una forma de hacerlo es lo siguiente.

```
import os

archivos = os.listdir()

# quitamos de la lista los nombres que no sean de archivo jpg
# en particular sacamos archivo python de este programa, que está
# en la misma carpeta
for nombre in archivos:
    if nombre[-4:] != ".jpg":
        archivos.remove(nombre)
    # nombre[-4:] son los últimos 4 elementos
```

```
# ordenamos según la fecha de creación. Para esto usamos
# el parametro key del método sort de listas, el cual
# permite indicar según qué ordenamos.
# Este parámetro se pasa poniendo "key = valor", donde
# valor es la función del orden
archivos.sort(key = lambda n: os.stat(n).st_mtime)

num = 1
for nombre in archivos:
    nombreNuevo = str(num) + ".jpg"
    os.rename(nombre, nombreNuevo) #operación para renombrar
    num = num + 1
```

La biblioteca `os` tiene también funcionalidades vinculadas al directorio (carpeta) de trabajo. Con la operación `os.getcwd()` se obtiene el directorio de trabajo actual. Por otra parte, con la operación `os.chdir()` se lo puede cambiar.

La posibilidad de cambiar el directorio de trabajo puede ser útil para ordenar las cosas mejor. Por ejemplo, con el ejemplo de las imágenes, en lugar de tener el programa y las imágenes en la misma carpeta, podríamos tener las imágenes dentro de una carpeta más adentro llamada `img`. Lo que tendríamos que hacer es comenzar moviendo el directorio de trabajo a esa carpeta. Esto se puede hacer con `os.chdir(os.getcwd()+"/img")`, donde en el argumento estamos agregando `"/img"` al directorio de trabajo actual, lo cual corresponde a meterse en esa carpeta.

4.3.2. Ejemplos de otras bibliotecas

Comencemos con la biblioteca `pypdf`, que sirve para procesar archivos en formato PDF. Esta es una biblioteca de código abierto. El repositorio con el código fuente es github.com/py-pdf/pypdf. Documentación oficial: pypdf.readthedocs.io/en/stable/.

Para instalar la biblioteca hay que ejecutar `pip install pypdf` desde una consola.

Hagamos un programa para concatenar archivos PDF. Supongamos que son 5 archivos y que los llamamos `1.pdf`, `2.pdf`, hasta `5.pdf`. En esta biblioteca tenemos, entre otras cosas, objetos de tipo `PdfReader` que permiten leer archivos PDF y objetos de tipo `PdfWriter` que permiten crear nuevos archivos PDF. Podemos hacer lo siguiente:

```
from pypdf import PdfReader, PdfWriter

# lista con nombres de los pdf a concatenar
inputs = ["1.pdf", "2.pdf", "3.pdf", "4.pdf", "5.pdf"]

# objeto para crear el pdf nuevo
writer = PdfWriter()

for input in inputs:
    # leemos el pdf input
    reader = PdfReader(input)
```

```

# iteramos en las páginas del pdf
for pag in reader.pages:
    # agregamos la página al w
    writer.add_page(pag)

# creamos el nuevo PDF con nombre pdfConcatenado
writer.write("pdfConcatenado.pdf")
# cerramos al writer
writer.close()

```

Supongamos ahora que queremos agregar una página en blanco al final de un PDF. Podríamos hacer lo siguiente:

```

from pypdf import PdfReader, PdfWriter

reader = PdfReader("input.pdf")
writer = PdfWriter()

# agregamos las páginas del pdf al objeto writer
for page in reader.pages:
    writer.add_page(page)

# luego agregamos una página en blanco
writer.add_blank_page()

# guardamos con otro nombre y cerramos
writer.write("output.pdf")
writer.close()

```

Veamos ahora un poco sobre la biblioteca **Pillow**, que sirve para procesar imágenes. Documentación: pillow.readthedocs.io/en/stable/.

Hay una clase **Image**, que se usa para trabajar con datos que son justamente imágenes. Con el método **open()** podemos abrir una imagen, con el método **save** podemos guardar una imagen y tiene varios otros con los que se pueden realizar transformaciones. Por ejemplo, si queremos cambiar las dimensiones de una imagen, por ejemplo a 600×600 , podemos hacer lo siguiente.

```

from PIL import Image

image = Image.open("input.jpg")

new_size = (600, 600)

# creamos una nueva imagen con el nuevo tamaño
resized_image = image.resize(new_size)

```

```
#guardamos la nueva imagen con nombre output.jpg
resized_image.save("output.jpg")
```

Las imágenes en formato jpg se guardan con cierto nivel de compresión, para disminuir el espacio que se ocupa a coste de bajar también la calidad. En el método `save` hay un parámetro `quality` al que se le puede dar un valor entre 0 y 100, siendo 0 máxima compresión y 100 la menor. Al disminuir ese parámetro, bajan el espacio que se ocupa y la calidad de la imagen. En general no se usa más que 95. El siguiente es un ejemplo:

```
from PIL import Image

image = Image.open("input.jpg")
image.save("output.jpg", quality = 50)
```

Notar que el parámetro se pasa poniendo `quality=50` dentro de los argumentos de la función. Es una forma de pasar parámetro en funciones de python, que se usa también por ejemplo en `lista.sort(key = f)`, que apareció en uno de los ejemplos de la biblioteca `os`.

Supongamos que necesitamos que la imagen pese a lo sumo 300KB, lo cual es más o menos 300000 bytes. Para encontrar el valor de compresión óptimo que nos da ese peso, podemos hacer una iteración en la que vamos bajando dicho valor y chequeando el peso del archivo resultante con la biblioteca `os`.

```
import os
from PIL import Image

# 300 KB
pesoMax = 300*1024

image = Image.open("input.jpg")

calidad = 95

image.save("output.jpg", quality = calidad)
peso = os.stat("output.jpg").st_size
while peso > pesoMax and calidad > 0:
    calidad = calidad - 1
    image.save("output.jpg", quality = calidad)
    peso = os.stat("output.jpg").st_size
```

Se podría hacer como una estrategia de búsqueda binaria para encontrar el óptimo con menos intentos. La idea sería probar con 50. Si es más pesado, pasamos a 25 y si es más liviano pasamos a 75. Terminaríamos al encontrar un n tal que con n de menor o igual pero con $n + 1$ de mayor. Sería de la siguiente forma.

```
import os
from PIL import Image
```

```

# 300 KB
pesoMax = 300*1024
image = Image.open("input.jpg")

arriba = 100
abajo = 0
calidad = (arriba + abajo)//2

image.save("output.jpg", quality = calidad)
peso = os.stat("output.jpg").st_size
while arriba - abajo > 1:
    if peso <= pesoMax:
        abajo = calidad
    else:
        arriba = calidad
    calidad = (arriba + abajo)//2

    image.save("output.jpg", quality = calidad)
    peso = os.stat("output.jpg").st_size

# el extremos de abajo es el que es <=
image.save("output.jpg", quality = abajo)

```

4.4. numpy

La biblioteca numpy (numpy.org/) (github.com/numpy/numpy) provee de funcionalidades eficientes para trabajar con datos vectoriales y matriciales (y en general con lo que se define como arreglos N dimensionales). Es la base de la computación científica en python. Documentación: numpy.org/doc/stable/.

Python ya tiene listas, con las que se puede codificar vectores y matrices, por lo que a primera vista podría parecer que no hace falta una biblioteca específica. Lo que no estamos teniendo en cuenta es la **eficiencia**.

Las listas de python son objetos muy versátiles, que permiten entre otras cosas ir cambiando el tamaño dinámicamente e incluso que los objetos sean de distintos tipos de datos. Sin embargo, esta gran flexibilidad tiene asociado un costo en términos de eficiencia. Con una estructura de datos más rígida la computadora podría operar mucho más rápidamente.

Por otra parte, **python** es un lenguaje de muy alto nivel, lo cual en comparación con lenguajes más básicos como **c**, lo hace más fácil de usar pero también menos eficiente. En python cada instrucción puede implicar muchas cosas que se hacen de fondo (para simplificar la tarea del programador), mientras que en **c** eso no ocurre. En este lenguaje, el programador define precisamente lo que hace la computadora. Esto hace que sea más difícil programar, pero a su vez da más control sobre cómo opera la computadora, lo cual se puede traducir en mayor eficiencia, si las cosas se hacen bien. Además, los programas en **c** se compilan de una antes de ejecutarse, mientras que los de **python** se compilan línea

por línea durante la ejecución. Compilar todo el programa de una aporta a la eficiencia, porque no hay que irlo compilando después mientras se ejecuta y además el compilador aplica optimizaciones que tienen en cuenta la totalidad del programa.

La biblioteca **numpy** aplica lo mencionado en los últimos dos párrafos para mejorar la eficiencia. Por una parte, si bien es una biblioteca de **python**, las funciones están programadas y compiladas en **c**, logrando algo así como un lenguaje híbrido, en el que tenemos funciones de **python** que encapsulan cosas programadas en **c**. Por otra parte, las estructuras de datos son en el fondo **arreglos**, que a diferencia de las listas, tienen un tamaño fijo y los elementos deben ser todos del mismo tipo de datos. Esta rigidez permite que la memoria se use de modo más eficiente, logrando mucha mayor velocidad en la ejecución.

En esta sección veremos lo básico de la biblioteca y algunas aplicaciones. En la documentación y buscando en internet se pueden encontrar muchas más cosas. En particular, hay una guía para principiantes, numpy.org/doc/stable/user/absolute_beginners.html, que puede ser útil.

4.4.1. Arreglos

El tipo central de **numpy** es **numpy.array**, que representa arreglos multidimensionales con tamaño y tipo de dato fijo. Un arreglo de dimensión 1 es un vector y uno de dimensión 2 es una matriz. Se puede inicializar un arreglo a partir de una lista o de una lista de listas.

```
>>> import numpy as np # se suele importar así
>>> vector = np.array([1,2,3,4]) # creamos un vector
>>> vector
array([1, 2, 3, 4])
>>> type(vector)
<class 'numpy.ndarray'>
>>> matriz = np.array([[1,2],[3,4],[5,6]])
>>> matriz
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Para saber las dimensiones de un arreglo se puede usar el atributo **shape**.

```
>>> vector.shape # vector de 4 elementos
(4,)
>>> matriz.shape # matriz de 3x2
(3, 2)
```

Para saber el tipo de datos de los elementos de un arreglo (que se define al crear el arreglo y todos sus elementos deben ser de ese tipo), se puede usar el atributo **dtype**.

```
>>> vector.dtype
dtype('int64')
>>> matriz.dtype
```

```
dtype('int64')
>>> arreglo = np.array([1.6,2.7,0.0])
>>> arreglo.dtype
dtype('float64')
```

El nombre de cada tipo de datos indica el tipo de los objetos y la cantidad de bits que ocupan. Por ejemplo, `int64` son enteros de 64 bits y `float64` son números en punto flotante de 64 bits. En la mayoría de los casos vamos a usar punto flotante.

Cabe aclarar que los tipos de datos que se usan en los arreglos de numpy no son los mismos que los de python. Los de numpy siempre tienen fija la cantidad de bits que ocupan, cosa que a veces no se cumple con los de python. Por ejemplo, los enteros `int64` se representan por una cantidad de bits fija que es 64. Esto implica que están restringidos a números entre -2^{63} y 2^{63} . Por otra parte, los números enteros de python no tienen una cantidad fija de memoria, sino que cuanto más grande son, más memoria se usa para guardarlos. Esto nos permite hacer cuentas como 2^{1500} sin preocuparnos, pero en general reduce la eficiencia.

Hay otras funciones para crear arreglos a partir de las dimensiones. Por ejemplo la función `np.ceros()` crea un arreglo de ceros con las dimensiones que se le pasen de tipo de datos `float64`. Si se le pasa un número `n`, es un vector de n ceros. Si se le pasa una tupla `(n,m)`, es una matriz de $n \times m$ de ceros.

```
>>> vCeros = np.zeros(7)
>>> vCeros
array([0., 0., 0., 0., 0., 0., 0.])
>>> mCeros = np.zeros((4,5)) # notar los paréntesis dobles
>>> mCeros
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Notar que los ceros aparecen como «0.» . Los puntos son para indicar que son número en punto flotante y no enteros.

La función `np.ones()` hace lo mismo pero inicializando con unos. Crear un arreglo con estas funciones es más rápido que hacerlo a partir de listas de python. De todos modos, independientemente de cómo creemos el arreglo, una vez que está creado es lo mismo.

También hay funciones para crear arreglos con rangos de números. Por ejemplo la función `np.linspace(min, max, cant)` genera un vector con la cantidad indicada de puntos equiespaciados entre el mínimo y el máximo.

```
>>> arreglo = np.linspace(1,10,20)
>>> arreglo
array([ 1.          ,  1.47368421,  1.94736842,  2.42105263,  2.89473684,
        3.36842105,  3.84210526,  4.31578947,  4.78947368,  5.26315789,
        5.73684211,  6.21052632,  6.68421053,  7.15789474,  7.63157895,
        8.10526316,  8.57894737,  9.05263158,  9.52631579, 10.          ])
```

También tenemos la función `np.identity(n)` que retorna la matriz identidad de $n \times n$.

```
>>> np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

En numpy.org/doc/stable/reference/routines.array-creation.html hay más funciones de creación de arreglos en numpy.

Se puede indizar arreglos de modo análogo a con las listas de python. La diferencia es que con matrices (y arreglos de mayor dimensión) se permite poner las dos coordenadas separadas por comas, es decir `matriz[i,j]`, lo cual se siente más intuitivo que `matriz[i][j]` (y de hecho es más eficiente).

```
>>> vector
array([1, 2, 3, 4])
>>> matriz
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> vector[1]
np.int64(2)
>>> matriz[1,0]
np.int64(3)
```

El valor `np.int64(2)` es el número 2 con su tipo indicado. Ídem con `np.int64(3)`. También se puede realizar asignaciones para cambiar el valor de alguna entrada del arreglo.

```
>>> vector
array([1, 2, 3, 4])
>>> vector[0] = 5
>>> vector
array([5, 2, 3, 4])
>>> matriz
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>>> matriz[1,2] = 5.7
>>> matriz
array([[1. , 1. , 1. , 1. ],
       [1. , 1. , 5.7, 1. ],
       [1. , 1. , 1. , 1. ],
       [1. , 1. , 1. , 1. ]])
```

Dada una matriz `m`, se puede indizar con rangos para obtener una submatriz. Dada una matriz `m`, tenemos que `m[i1:i2,j1:j2]` es la submatriz con las filas dadas por el rango `i1:i2` y las columnas dadas por el rango `j1:j2`. La convención de los rangos es la misma que siempre en python: `x1:x2` son los `x` tales que `x1 ≤ x < x2`. Por otra parte, podemos hacer `m[i,:]` para obtener la fila `i`-ésima y `m[:,j]` para la columna `j`-ésima.

```

>>> m = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> m
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> m[0:2,0:2]
array([[1, 2],
       [4, 5]])
>>> m[0:2,1:3]
array([[2, 3],
       [5, 6]])
>>> m[0,:]
array([1, 2, 3])
>>> m[:,0]
array([1, 4, 7])

```

4.4.2. Operaciones con arreglos

Se puede hacer suma de vectores o matrices y producto de matrices con los símbolos + y @, respectivamente.

```

>>> v1 = np.array([1,3,5,7])
>>> v2 = np.array([4,4,4,4])
>>> v1+v2
array([ 5,  7,  9, 11])
>>> matriz = np.ones((4,4))
>>> matriz + matriz
array([[2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.]])
>>> matriz @ matriz
array([[4., 4., 4., 4.],
       [4., 4., 4., 4.],
       [4., 4., 4., 4.],
       [4., 4., 4., 4.]])
>>> v1 @ matriz
array([16., 16., 16., 16.])

```

En este punto se puede hacer un experimento para ver la diferencia de tiempo de ejecución entre arreglos de numpy y listas de python. Por ejemplo, podemos ver cuánto lleva multiplicar dos matrices de 1000×1000 en las que todas las entradas son 1. Con listas de python se puede hacer de la siguiente forma.

```

def mult(m1,m2):
    N = len(m1)
    res = []

```

```

    for i in range(N):
        fila = []
        for j in range(N):
            valor = 0
            for k in range(N):
                valor += m1[i][k] * m2[k][j]
            fila.append(valor)
        res.append(fila)
    return res

def matriz(N):
    matriz = []
    for i in range(N):
        fila = []
        for j in range(N):
            fila.append(1)
        matriz.append(fila)
    return matriz

A = matriz(1000)
B = mult(A,A)

```

Con numpy es lo siguiente.

```

import numpy as np

A = np.ones((1000,1000))
B = A@A

```

Se recomienda ejecutar ambos y ver la diferencia de tiempo que tardan.

Para transponer una matriz alcanza escribir el nombre de la matriz y «.T».

```

>>> m = np.array([[1,2],[3,4]])
>>> m
array([[1, 2],
       [3, 4]])
>>> m.T
array([[1, 3],
       [2, 4]])

```

4.4.3. Álgebra lineal

El módulo `numpy.linalg` incluye funcionalidades de álgebra lineal. Tiene su página dentro de la documentación: numpy.org/doc/stable/reference/routines.linalg.html. Veamos algunas de las funcionalidades.

Se puede calcular determinantes, invertir matrices y resolver sistemas de tipo $Ax = b$.

```

>>> a = np.array([[1,1,1],[0,1,0],[1,1,0]])
>>> a
array([[1, 1, 1],
       [0, 1, 0],
       [1, 1, 0]])
>>> np.linalg.det(a) # calcula el determinante
np.float64(-1.0)
>>> np.linalg.inv(a) # calcula la inversa
array([[ 0., -1.,  1.],
       [ 0.,  1.,  0.],
       [ 1., -0., -1.]])
>>> b = np.array([0,1,1])
>>> b
array([0, 1, 1])
>>> x = np.linalg.solve(a,b) # resuelve ax = b
>>> x
array([ 0.,  1., -1.])
>>> a@x # verificación
array([0., 1., 1.])

```

Por otra parte, con `np.linalg.eig()` se pueden determinar valores propios y vectores propios de una matriz. Esta operación retorna un objeto con dos atributos: **eigenvalues** con los valores propios y **eigenvectors** con una matriz cuyas columnas son los vectores propios asociados a los valores propios.

```

>>> m = np.array([[1,2,1],[1,2,2],[0,0,5]])
>>> m
array([[1, 2, 1],
       [1, 2, 2],
       [0, 0, 5]])
>>> res = np.linalg.eig(m)
>>> res.eigenvalues # valores propios
array([0., 3., 5.])
>>> res.eigenvectors # matriz con vectores propios en columnas
array([[ -0.89442719, -0.70710678,  0.46156633],
       [ 0.4472136 , -0.70710678,  0.59344243],
       [ 0.          ,  0.          ,  0.65938047]])
>>> vep1 = res.eigenvectors[:,0] # esto da la primera columna
>>> vep2 = res.eigenvectors[:,1]
>>> vep3 = res.eigenvectors[:,2]
>>> m @ vep1
array([0., 0., 0.])
>>> res.eigenvalues[0] * vep1 # debería dar lo mismo
array([-0.,  0.,  0.])
>>> m @ vep2
array([-2.12132034, -2.12132034,  0.          ])
>>> res.eigenvalues[1] * vep2

```

```
array([-2.12132034, -2.12132034,  0.          ])
>>> m @ vep3
array([2.30783166, 2.96721213, 3.29690237])
>>> res.eigenvalues[2] * vep3
array([2.30783166, 2.96721213, 3.29690237])
```

En caso de que sea necesario se trabaja con números complejos. De hecho numpy tiene un tipo de datos de números complejos de 128 bits, en los que la parte imaginaria se indica con j.

```
>>> m = np.array([[1,1],[-1,1]])
>>> res = np.linalg.eig(m)
>>> res.eigenvalues
array([1.+1.j, 1.-1.j])
>>> res.eigenvalues.dtype
dtype('complex128')
>>> res.eigenvectors
array([[0.70710678+0.j, 0.70710678-0.j],
       [0.          +0.70710678j, 0.          -0.70710678j]])
```

4.4.4. Mínimos cuadrados

Dada una matriz A de $m \times n$ y un vector $b \in \mathbb{R}^m$, el problema de mínimos cuadrados lineal (PMCL) consiste en hallar $x \in \mathbb{R}^n$ que minimice $\|Ax - b\|_2$. Si el sistema es compatible, la solución al PMCL es solución al sistema, ya que en ese caso se tiene que la norma es 0. En caso de que el sistema no sea compatible, el PMCL busca hallar el x que *más se aproxima a ser solución*, en el sentido de la norma euclídea.

Un caso en el que esto se aplica es el de determinar parámetros de un modelo matemático aplicado en base a datos. Puede ser por ejemplo una ecuación física con ciertos coeficientes que se cumple y queremos determinar los valores de estos. En general los modelos son simplificaciones de la realidad y además las mediciones tienen errores, por lo que no se cumplen de modo exacto y hay que aplicar algo como mínimos cuadrados para obtener la solución que mejor se ajusta a los datos. Veamos cómo se escribe un problema de este tipo en el marco de PMCL, lo cual implica definir A , b y x .

Supongamos que tenemos un conjunto de datos $\{(t_i, y_i)\}_{1 \leq i \leq n}$, que según cierto modelo (o ley) cumplen la ecuación $y = \alpha t + \beta$ para ciertos valores α y β a determinar. Las ecuaciones que resultan de aplicar el modelo a los datos son las siguientes.

$$\begin{aligned}\alpha t_1 + \beta &= y_1 \\ \alpha t_2 + \beta &= y_2 \\ &\vdots \\ \alpha t_n + \beta &= y_n\end{aligned}$$

Esto se puede traducir a un sistema $Ax = b$ con las siguientes definiciones.

$$A = \begin{pmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \vdots \\ t_n & 1 \end{pmatrix} \quad x = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad b = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

De esta forma, hallar los parámetros se corresponde con resolver el sistema $Ax = b$. En el caso ideal de que el modelo sea exacto y los datos no tengan errores, se podrá resolver el sistema de ecuaciones directamente. Sin embargo, normalmente eso no es el caso y para obtener los valores de α y β (vector x) se debe resolver el problema de mínimos cuadrados asociado.

En numpy tenemos la función `np.linalg.lstsq()` que resuelve problemas de mínimos cuadrados. Recibe dos parámetros, de los cuales el primero es la matriz A y el segundo el vector b . Retorna varias cosas, de las cuales la primera es la solución. Página de la función en la documentación: numpy.org/doc/2.0/reference/generated/numpy.linalg.lstsq.html.

Usemos esto para resolver el problema anterior con datos $\{(1, 1), (2, 3), (3, 2), (4, 4)\}$.

```
>>> t = np.array([1,2,3,4]) # t de los datos
>>> y = np.array([1,3,2,4]) # y de los datos
>>> a = np.ones((4,2)) # matriz inicializada con unos
>>> # la segunda columna ya queda como tiene que estar
>>> a[:,0] = t # en la primera columna van los valores de t
>>> a # vemos la matriz
array([[1., 1.],
       [2., 1.],
       [3., 1.],
       [4., 1.]])
>>> res = np.linalg.lstsq(a,y) # el b es igual a y
>>> x = res[0] # lsqr retorna varias cosas. La sol es lo primero
>>> x
array([0.8, 0.5])
>>> alpha = x[0]
>>> beta = x[1]
```

Concluimos que los valores de los parámetros que mejor se ajustan a estos datos (en el sentido de mínimos cuadrados) son $\alpha = 0,8$ y $\beta = 0,5$. Con la biblioteca `matplotlib` veremos formas de hacer gráficas a partir de esto.

En caso de otros modelos, como uno cuadrático con tres parámetros $y = \alpha t^2 + \beta t + \gamma$, la construcción de la matriz A y del vector b es análoga. En general si la cantidad de datos es m y la cantidad de parámetros por determinar es n , la matriz A resulta de $m \times n$ y el vector b de m coordenadas. Para un desarrollo más completo, se recomiendan las notas de teórico del curso “Métodos Numéricos” de la Facultad de Ingeniería, FIng, sección 5.1.

4.5. matplotlib

La biblioteca `matplotlib` (matplotlib.org/) (github.com/matplotlib/matplotlib) provee de funcionalidades para generar gráficas y otro tipo de visualizaciones de datos en

python. En esta sección veremos algunas de las funcionalidades básicas, las cuales usaremos para graficar distintas cosas que calculemos con otras bibliotecas. En la página hay documentación, que incluye tutoriales y ejemplos con muchas otras funcionalidades, pero también una guía básica en matplotlib.org/stable/users/explain/quick_start.html.

En general de matplotlib se utiliza el módulo `matplotlib.pyplot`, que suele importarse como:

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

Los datos que se grafican suelen ser dados por arreglos de numpy, así que lo más común es importarlo también.

Para representar las figuras y los distintos ejes de coordenadas, tenemos los objetos **Figure** y **Axes**. Cada objeto Figure representa una imagen, la cual puede contener uno o más objetos Axes, que son dentro de los cuales se realizan las gráficas. Si quisiéramos una sola figura con dos gráficas, tendríamos que crear un objeto Figure con dos objetos Axes. Una forma sencilla de crear una figura con un objeto axes es:

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

Después de hacer eso, **fig** es la figura y **ax** es el axes perteneciente. En este momento la figura consta de ejes cartesianos sin datos adentro.

Podemos guardar la figura en una imagen con `fig.savefig('nombre.png')`.

Podemos mostrar la figura en una ventana con `plt.show()`. Al hacer esto la ejecución del código se interrumpe hasta que la ventana se cierre y después de esto, internamente la figura se cierra, por lo que si volvemos a ejecutar `plt.show()`, esta vez no se muestra.

Para graficar datos se puede usar `ax.plot(x,y)`, donde **x** e **y** son arreglos de numpy unidimensionales con las coordenadas de los puntos a graficar. Por defecto lo que hace es graficar una función lineal a trozos entre los puntos dados. Por ejemplo si queremos hacer esto con los puntos (1,1), (2,4), (3,2) y (4,3) alcanza con lo siguiente:

```
import numpy as np # en caso de que no lo hayamos importado antes

x = np.array([1,2,3,4]) # las coordenadas x de los puntos
y = np.array([1,4,2,3]) # las coordenadas y de los puntos
ax.plot(x,y)
plt.show() # si queremos que lo muestre
```

En la figura 4.1 vemos la gráfica que se genera. De aquí en más, se incentiva al lector a ejecutar los códigos para generarlas por si mismo.

Se puede pasar un tercer parámetro para indicar el formato con el que se grafican los datos. Por ejemplo con "ro" los puntos se grafican con círculos rojos. La instrucción completa es `ax.plot(x,y, "ro")`. Para más información, la documentación tiene una página sobre `plot`: matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html.

Para graficar varias cosas en los mismos ejes, se puede simplemente ejecutar varias veces la instrucción `ax.plot()`.

Si queremos graficar una función que no sea lineal a trozos, lo que hacemos es dibujar una lineal a trozos con puntos muy cercanos, de modo que con la resolución de la computadora se vea como la función. Para esto es útil la función `np.linspace()`, con la que

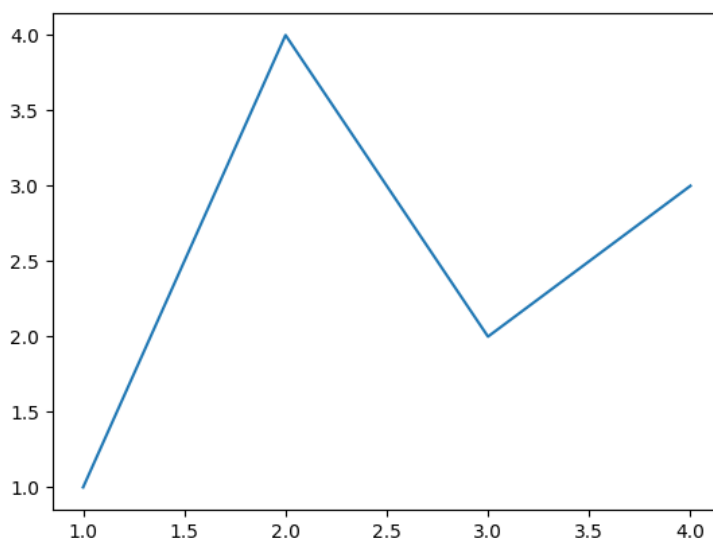


Figura 4.1: Ejemplo con matplotlib

podemos obtener un vector de muchos puntos equiespaciados. Por ejemplo para graficar la función seno podemos hacer lo siguiente.

```
fig, ax = plt.subplots()
x = np.linspace(0,2*np.pi,100) # 100 puntos equiesp entre 0 y 2pi
y = np.sin(x) # seno coordenada a coordenada
ax.plot(x,y)
plt.show()
```

Con la resolución que viene por defecto se ve como la función seno. Haciendo zoom sobre un máximo o un mínimo (se puede hacer desde la ventana del `plt.show()`) se puede ver que de hecho es lineal a trozos pero con líneas chicas.

En general la forma de graficar una función f en $[a, b]$ es genera un vector \mathbf{x} de varios puntos equiespaciados entre a y b , luego generar un vector \mathbf{y} del mismo tamaño con los valores de la función evaluada en los puntos del vector \mathbf{x} y finalmente hacer `ax.plot(x,y)`. A veces el vector \mathbf{y} se puede hacer de una con alguna función de numpy que se ejecute coordenada a coordenada, como en el ejemplo del seno, mientras que otras veces hay que calcular las entradas una a una con un `for` de python.

Dado el ejemplo de mínimos cuadrados que vimos en 4.4.4, podemos usar matplotlib para hacer una gráfica y visualizar la recta que mejor aproxima los puntos. Lo hacemos de la siguiente forma.

```
t = np.array([1,2,3,4]) # t de los datos
y = np.array([1,3,2,4]) # y de los datos
a = np.ones((4,2)) # matriz inicializada con unos
# la segunda columna ya queda como tiene que estar
a[:,0] = t # en la primera columna van los valores de t
```

```

res = np.linalg.lstsq(a,y) # el b es igual a y
x = res[0] # lsqr retorna varias cosas. La sol es lo primero
alpha = x[0]
beta = x[1]

# ahora graficamos
fig, ax = plt.subplots()
ax.plot(t, y, "ro") # graficamos los puntos con circulos rojos

# la recta es y = alpha*x + beta. La graficamos con dos puntos
rectaX = np.array([1,4])
rectaY = np.array([alpha*1 + beta, alpha*4 + beta])
ax.plot(rectaX, rectaY) # graficamos la recta arriba de los puntos
plt.show()

```

4.6. scipy

La biblioteca `scipy` (scipy.org/es/) (github.com/scipy/scipy) contiene funcionalidades para distintos problemas de matemática pura o aplicada, como integración y ecuaciones diferenciales. Aquí veremos justamente esas dos, pero en la página se pueden encontrar muchas otras cosas. Documentación: docs.scipy.org/doc/scipy/.

Las dos funcionalidades que veremos están dentro del módulo `scipy.integrate`. Hacemos la siguiente importación:

```
import scipy.integrate as integrate
```

Para calcular numéricamente integrales se usa la función `quad(f,a,b)`, a la que se le pasan como parámetros la función a integrar y los extremos de integración. La función puede estar definida con `def` o mediante una expresión `lambda`. Se retornan dos cosas: el valor de la integral y una estimación del error numérico que puede haber. La integral que se calcula es:

$$\int_a^b f(x)dx$$

Veamos algunos ejemplos:

```

>>> import scipy.integrate as integrate
>>> import numpy as np
>>> integrate.quad(lambda x: x**2, 0, 1)
(0.3333333333333337, 3.700743415417189e-15) # (valor, error)
>>> integrate.quad(lambda x: 1/x, 1, np.e)
(0.9999999999999998, 1.1102230246251562e-14)
>>> integrate.quad(lambda x: np.exp(-x**2), 0, 1)
(0.7468241328124271, 8.291413475940725e-15)
>>> integrate.quad(lambda x: np.exp(-x**2), 0, 2)
(0.8820813907624215, 9.793070696178202e-15)
>>> integrate.quad(lambda x: np.exp(-x**2), -np.inf, np.inf) # impropia
(1.7724538509055159, 1.4202636756659625e-08)

```

Pasemos ahora al problema de las ecuaciones diferenciales ordinarias. Queremos resolver un problema de valores iniciales, lo cual es:

$$\begin{cases} y'(t) = f(y(t), t) & \text{con } t \in [a, b] \\ y(a) = y_0 \end{cases}$$

Es decir, queremos hallar una función $y : [a, b] \rightarrow \mathbb{R}$ que cumpla la ecuación diferencial $y'(t) = f(y(t), t)$ y la condición inicial $y(a) = y_0$.

Para esto usamos la función `odeint(f, y0, t)`, donde `f` es la función que define la ecuación diferencial, `y0` es el valor inicial y `t` es el vector de los tiempos en los que queremos determinar el valor de la solución. Los a y b están dados por el vector `t`. El valor de a es la primera entrada del vector, mientras que el de b es la última. Se retorna un vector con los valores de la solución a la ecuación diferencial en los tiempos dados por el vector `t`.

A modo de ejemplo, podemos usar esto para resolver la ecuación diferencial $y' = \sin(y + t)$ con condición inicial $y(0) = 0$ en el intervalo $[0, 10]$ y graficarla con `matplotlib`.

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

# definimos la función de la ecuación diferencial
def pend(y,t): # el orden de los parámetros importa
    return np.sin(y+t)

y0 = 0 # valor inicial
t = np.linspace(0,10,100) # tiempos para calcular la sol
y = integrate.odeint(pend, y0, t)
# y queda con los valores de la solución en los tiempos de t

fig, ax = plt.subplots()
ax.plot(t,y)
plt.show()
```

Se puede trabajar también con sistemas de ecuaciones diferenciales, en los que $y(t)$ es un vector de funciones. La idea es que $y(t) = (y_1(t), \dots, y_n(t))$, con lo que $y'(t) = (y_1'(t), \dots, y_n'(t))$. Consideremos a modo de ejemplo el sistema con dos funciones incógnita dado por $y_1'(t) = -y_2(t)$ y $y_2'(t) = y_1(t)$ con condiciones iniciales $y_1(0) = 1$ y $y_2(0) = 0$. De modo vectorial se escribe de la siguiente forma.

$$\begin{cases} \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix}' = \begin{pmatrix} -y_2(t) \\ y_1(t) \end{pmatrix} \\ \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{cases}$$

Ahora la función f es $f((y_1, y_2), t) = (-y_2, y_1)$, una función $f : \mathbb{R}^2 \times \mathbb{R} \rightarrow \mathbb{R}^2$. Al hacerlo en python será una función `pend(y, t)`, donde `y` es una lista con dos entradas y `t`

es un número, la cual retorna una lista de dos entradas. Resolvamos en el intervalo $[0, 10]$ y grafiquemos las dos funciones juntas.

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

def pend(y,t):
    return [-y[1], y[0]]

y0 = [1,0] # ahora el valor inicial es lista de dos números
t = np.linspace(0,10,100) # tiempos para calcular la sol
y = integrate.odeint(pend, y0, t)
# y es una matriz de 2x100, tal que la primera columna son
# los valores de y1 y la segunda son los de y2
y1 = y[:,0]
y2 = y[:,1]

fig, ax = plt.subplots()
ax.plot(t,y1)
ax.plot(t,y2, "r") # esta roja
plt.show()
```

4.7. sympy

La biblioteca `sympy` (sympy.org/en/index.html) (github.com/sympy/sympy) es para hacer operaciones matemáticas de modo simbólico, en lugar de trabajar con aproximaciones. En ese sentido es muy diferente a las otras que venimos viendo. En la página hay documentación, de la cual se recomienda el tutorial introductorio.

A modo de ejemplo de computación simbólica, al trabajar con raíces cuadradas, si el resultado es exacto se lo calcula, pero en otro caso se las manipula simbólicamente, aplicando de modo automático algunas simplificaciones cuando se puede. Por ejemplo:

```
>>> sympy.sqrt(4)
2
>>> sympy.sqrt(5)
sqrt(5)
>>> sympy.sqrt(8)
2*sqrt(2)
```

Se puede introducir variables con la función `sympy.symbols()` y luego usarlas para distintas cosas. Por ejemplo, podemos trabajar con polinomios, desarrollándolos o factorizándolos.

```
>>> x, y = sympy.symbols('x y')
>>> expr = (x**2 + 2*x)*(y**2 - x**2)
>>> expr
```

```

(-x**2 + y**2)*(x**2 + 2*x)
>>> sympy.expand(expr)
-x**4 - 2*x**3 + x**2*y**2 + 2*x*y**2
>>> sympy.factor(expr)
-x*(x + 2)*(x - y)*(x + y)

```

También podemos calcular derivadas e integrales.

```

>>> f = y*x**2 - sympy.sin(x)
>>> sympy.diff(f,x) # respecto a x
2*x*y - cos(x)
>>> sympy.diff(f,y) # respecto a y
x**2
>>> sympy.diff(expr,x)
-2*x*(x**2 + 2*x) + (2*x + 2)*(-x**2 + y**2)
>>> cuad = x**2
>>> sympy.integrate(cuad,x) # primitiva respecto a x
x**3/3
>>> sympy.integrate(cuad,(x,0,1)) # integral entre 0 y 1
1/3
>>> sympy.integrate(f,(x,0,1)) # va a depender de y
y/3 - 1 + cos(1)
>>> sympy.integrate(sympy.exp(x),x)
exp(x)
>>> sympy.integrate(sympy.exp(-x**2),(x,-sympy.oo,sympy.oo)) #impropia
sqrt(pi)

```

Además, podemos trabajar con matrices, por ejemplo hallando valores propios, forma diagonal y forma de Jordan. Cabe mencionar que el último es un problema numéricamente inestable, porque dada una matriz con forma de Jordan no diagonal, si modificamos un poquito alguna de las entradas, es común que se vuelva diagonalizable. Esto hace que no sea adecuado para tratarse con métodos numéricos aproximados, por lo que no está en `numpy` ni `scipy`. En `sympy`, que en su lugar funciona simbólicamente, sí se puede hacer.

```

>>> m = sympy.Matrix([[1,1,1],[1,1,0],[0,0,1]])
>>> m
Matrix([
[1, 1, 1],
[1, 1, 0],
[0, 0, 1]])
>>> m.eigenvals()
{1: 1, 2: 1, 0: 1} # los :1 indican multiplicidad
>>> P, D = m.diagonalize()
>>> D
Matrix([
[0, 0, 0],
[0, 1, 0],
[0, 0, 2]])

```

```
>>> m = sympy.Matrix([[1,2,0],[0,1,0],[0,0,1]])
>>> P,J = m.jordan_form()
>>> J
Matrix([
[1, 1, 0],
[0, 1, 0],
[0, 0, 1]])
```

Capítulo 5

Introducción a TeX y LaTeX

Latex, escrito de modo estilizado L^AT_EX o LaTeX, es un lenguaje para generar documentos de alta calidad tipográfica, especialmente de tipo científico con presencia de fórmulas matemáticas. Aquí el sentido de “alta calidad tipográfica” es que los distintos elementos (texto, fórmulas, etc.) quedan organizados de modo fácilmente legible. No es una herramienta que destaque por la posibilidad de hacer cosas *creativas* (aunque sí que se las puede hacer si uno profundiza), sino por la facilidad de generar textos bien ordenados que transmiten la información de modo claro. Que un material se entienda requiere de dos cosas: primero, que el contenido sea claro y segundo, que esté bien organizado y prolijo. Latex es una herramienta para lograr lo segundo.

Los materiales de este curso están hechos con latex. Se puede acceder a los archivos fuente en https://github.com/fcarbballal/curso_comp.

Latex está basado en T_EX, el cual es un lenguaje más elemental (aunque muy potente) para generación de documentos que puede extenderse con elementos llamados *macros*. De hecho, LaTeX es TeX junto con un conjunto de macros que simplifica la creación de documentos.

Tex fue hecho por Donald Knuth y Latex por Leslie Lamport algunos años después. Tras hacer tex, Knuth creó un conjunto de macros llamado “plain tex”, que facilita el uso.

$$\text{T}_{\text{E}}\text{X} \rightarrow \begin{cases} \text{Plain T}_{\text{E}}\text{X} \\ \text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} \end{cases}$$

Latex (que se creó después) incluye la mayoría de los macros de plain tex, pero no todos. Aparte de eso tiene muchas más cosas y una gran cantidad de paquetes con los que se puede extender (análogamente al uso de bibliotecas en python). Los paquetes de latex suelen tener documentación en ctan.org

Respecto a materiales de referencia, considerar los siguientes:

1. *La introducción no-tan-corta a latex* [4]. Introduce los conceptos y además explica y ejemplifica los elementos que se usan comúnmente. La versión en inglés [5] es más nueva. Se trata de material disponible en internet. Ver los enlaces en la bibliografía.
2. *The Texbook* [2]. Es el manual de tex y plain tex escrito por el autor del sistema. A pesar de ser un manual, es muy didáctico y agradable de leer, al menos para quien escribe estos apuntes. Además de explicar tex, tiene muchos comentarios de tipografía, tanto en general como específica de textos con matemática.

3. *Latex, a document preparation system* [3]. Manual de Latex por el autor.

5.1. Instalación y uso de latex

Hay que instalar un compilador y opcionalmente un entorno de desarrollo integrado (un programa que facilita el uso). Respecto al compilador, se puede usar por ejemplo *MiKTeX*: miktex.org/. Respecto al entorno de desarrollo integrado, se puede usar por ejemplo *TeXstudio*: texstudio.org/.

El modo de usar latex es el siguiente. Primero se escribe *código latex* que especifica el contenido y la estructura del documento y luego con el compilador se genera el archivo pdf junto con otros archivos auxiliares. El código latex se escribe en un archivo de formato “**.tex**”. El pdf y los archivos auxiliares quedan en la misma carpeta que el código fuente (es decir, el código latex).

Los entornos integrados de desarrollo suelen permitir visualizar a la vez el código latex y el pdf generado. Para comenzar, con el entorno de desarrollo se abre un archivo tex preexistente o se crea uno nuevo. Suele haber un botón para compilar y mostrar el pdf generado.

También se puede compilar desde la terminal sin necesidad de entorno de desarrollo integrado. Teniendo miktex instalado, una forma básica de hacerlo es ejecutando el comando

pdflatex nombre

en la carpeta en que tenemos el archivo fuente **nombre.tex**. Cabe aclarar que esta forma básica de compilación a veces no genera bien ciertas cosas. Por ejemplo, para que el índice quede bien hay que compilar dos veces seguidas. Esto parece ser porque el índice referencia cosas posteriores y la ejecución es lineal; con cosas de tex que no impliquen referencias parece funcionar bien de una. Hay otras formas de compilar desde la terminal que resuelven esos problemas de modo más prolijo.

5.2. Estructura de un archivo latex

Los archivos de latex comienzan con una línea en la que se indica el tipo de documento. Algunos de los posibles tipos son **article**, **report** y **beamer** (siendo este último para hacer dispositivas). Luego viene el preámbulo, que es una parte en la que se suelen importar paquetes y definir cosas que se usarán en el resto del documento. Finalmente, va el contenido del documento entre `\begin{document}` y `\end{document}`.

`\documentclass[a4paper, 12pt]{article}` → tipo de documento

<code>\usepackage[spanish]{babel}</code> <code>\usepackage{amsmath}</code> <code>\usepackage{amssymb}</code> <code>:</code>	}	preámbulo
--	---	-----------

`\begin{document}`

`:`

→ contenidos

`\end{document}`

5.3. Escritura básica de texto

Los párrafos en lenguaje natural (español, inglés, etc.) son una parte muy importante de los textos científicos. Un documento que solamente tiene fórmulas y diagramas sin explicaciones suele ser mucho más difícil de entender que uno que incluye párrafos en los que se explican las ideas.

Para hacer un párrafo, uno escribe el texto con las palabras, signos de puntuación y espacios y luego LaTeX (de hecho TeX) se ocupa de armar el párrafo, eligiendo los puntos en los que terminan las líneas y la cantidad de espacio entre las palabras (que no es algo fijo) de modo que quede lo mejor posible.

Armar el párrafo no es trivial porque se hacen de modo que tanto el borde izquierdo como el derecho del texto sean rectos. Para ilustrar esto, se colocan líneas verticales a los lados de este párrafo. Para lograrlo minimizando la cantidad de palabras que se deben cortar por la mitad, la cantidad de espacio entre las palabras varía de una línea a otra. El algoritmo de TeX elige los puntos en los que terminar las líneas y la separación entre las palabras de modo que quede lo mejor posible (evitando tener que separar palabras con guiones y que los espacios sean demasiado apretados u holgados, entre otras cosas). En este párrafo, la segunda línea tiene espacios un poco más cortos que la tercera.

Como el que se encarga de determinar el espacio entre las palabras es TeX, en el código latex **no se tienen en cuenta espacios múltiples**, es decir, muchos espacios juntos se tratan de modo igual que un espacio solo. Por otra parte, un salto de línea solo se toma como equivalente a un espacio.

Para indicar el final de un párrafo se debe dejar una línea en blanco. La primera línea de cada párrafo tiene sangría por defecto (se la puede sacar poniendo el comando (ver sec 5.5) `\noindent` al principio).

A modo de ejemplo, lo siguiente es un código de latex a la izquierda con los párrafos generados a la derecha.

```
Este es un ejemplo de
párrafo para ilustrar
el funcionamiento de tex con
espacios y saltos de línea.
```

Este es un ejemplo de párrafo
para ilustrar el funcionamiento de
tex con espacios y saltos de línea.
Aquí tenemos otro párrafo.

Aquí tenemos otro párrafo.

Hay algunos símbolos que tienen usos especiales en TeX, que si los escribimos en el código, en lugar de agregar el símbolo al documento resultante, se toman como algún tipo de comando. Son los siguientes:

`\ { } $ & # ^ _ % ~`

Puede haber más símbolos con usos especiales que provengan de latex o de los paquetes que importamos en el preámbulo del documento.

El símbolo `~` sirve para crear lo que se llama un *espacio irrompible*. Esto es un espacio en el que se prohíbe realizar un salto de línea. Por ejemplo, si escribimos `Juan~Perez`, no se permitirá que una línea termine entre medio de “Juan” y “Perez”.

El símbolo `%` sirve para hacer comentarios, los cuales cumplen el mismo fin que los comentarios de python: dejar aclaraciones para quien lee el código. Lo que se escriba

después del símbolo es ignorado por el compilador y no afecta el documento. Cabe destacar que también se ignora el salto de línea, por lo que si se pone el % después de una palabra y en la siguiente línea se comienza con una palabra, estas dos quedarán juntas. Para evitar esto se puede por ejemplo poner un espacio antes del %.

Los otros símbolos reservados irán apareciendo a lo largo del capítulo.

Un aspecto que requiere cierta atención es el uso de comillas. Si bien muchas veces simplemente escribimos el símbolo `"`, en realidad existen símbolos distintos para comillas de apertura y de cierre: `«` `»` y `«` `»`. Esas son de hecho comillas dobles. También tenemos las comillas simples que son `‘` `’` y `‘` `’`. Las comillas simples de apertura y de cierre se escriben en el código con `‘` y `’` respectivamente (acento agudo y apóstrofe). Las dobles se escriben con los mismos símbolos dos veces, es decir `“` y `”`. En caso de que en el teclado falte alguno de los acentos (en los teclados españoles puede no estar el acento agudo), se pueden reemplazar con los comandos `\lq` y `\rq` (veremos comandos en la sección 5.5). Por otra parte, las comillas españolas `“` `»` y `“` `»` se generan con dos `<` juntos y dos `>` juntos, respectivamente.

5.4. Modos

TeX tiene distintos modos, entre los que va cambiando al procesar el código. Hay dos modos “normales” o “de texto” que son el **vertical** y el **horizontal**, junto con dos modos matemáticos, que son el **matemático** (matemático básico, digamos) y el **display matemático**. Es importante entenderlos al menos hasta cierto punto, ya que hay funcionalidades que se pueden aplicar solamente en algunos de ellos.

TeX está en modo vertical cuando organiza contenido de modo vertical (una cosa arriba de otra) y en modo horizontal cuando organiza contenido de modo horizontal. En la generación de párrafos comunes y corrientes TeX va alternando entre estos dos modos. Al generar una línea está en modo horizontal, pues está poniendo una letra al lado de otra, mientras que al armar el párrafo a partir de las líneas está en modo vertical, ya que pone una línea arriba de otra. Cuando veamos cajas (en una sección posterior), tendremos una forma de alternar flexiblemente entre estos dos modos y ordenar lo que queramos de modo horizontal o vertical.

El modo matemático es para ingresar fórmulas matemáticas en el medio del texto. Por ejemplo $x + y = z$ y $x - y = z$ son fórmulas hechas con el modo matemático. Las fórmulas se delimitan con símbolos `$`. Para generar las anteriores, en el código se escribió `$x+y=z$` y `$x-y=z$`. En modo matemático los símbolos se colocan (por defecto) con una fuente distinta. Comparar la fórmula $x + y = z$ con el texto `x+y=z`.

En modo matemático, el espaciado entre los símbolos funciona de modo distinto. Los espacios y saltos de línea que pongamos son ignorados y no se permiten líneas en blanco (da error). Por ejemplo, tanto `xy` como `$x y$` generan xy . Por otra parte, automáticamente se coloca algo de espacio entre símbolos de operación (como `+`) y más espacio entre símbolos de relación (como `=`). Observar la fórmula $xy + z = w$, que tiene los tres espaciados. Esta convención de espaciado ayuda a entender fácilmente las fórmulas. Más adelante (cuando nos enfoquemos en la escritura de fórmulas) veremos modos de alterar el espaciado entre los elementos (aunque en la mayoría de las situaciones no es necesario).

El modo display matemático es para mostrar una fórmula centrada y con margen arriba y abajo. Se hace escribiendo la fórmula delimitada con dos \$ juntos de cada lado o con \[y \]. Por ejemplo con $x+y=z$ o $\[x+y=z\]$ se genera

$$x + y = z$$

y si seguimos escribiendo después continúa el párrafo. En Latex se recomienda \[y \]; los dobles \$ son la forma de Tex y a veces no funcionan bien con ciertas cosas de Latex.

Si ponemos una línea en blanco antes de la fórmula, esta comienza un párrafo nuevo, lo cual aumenta el espacio en blanco. Por otra parte, una línea en blanco después de la fórmula hace que lo que viene después comience un párrafo nuevo.

Que tex ignore los espacios y saltos de línea en modo matemático, permite escribir fórmulas complejas con una estructura que nos resulte más fácil de manejar. Eso sí, si dejamos una línea en blanco, da error. Lo que sí podemos hacer es poner una línea con un comentario vacío, si queremos separar más las cosas. La siguiente fórmula

$$(x + y/2)z - 10w(13y + 2u) = 40i + 3j$$

es puede escribir por ejemplo como:

```

$$
(x+y/2)z
-
10w(13y+2u)
=
40i + 3j
$$

```

Conforme vamos agregando cosas más complejas, el uso de esto puede volverse muy útil. En algunas de las fórmulas posteriores de este capítulo se aplica.

5.5. Grupos y estructuras de control

Se pueden delimitar grupos con los símbolos reservados “{” y “}”. Los grupos se pueden usar para tratar su contenido como una unidad y para aislarlo del resto del documento. En seguida veremos varios usos en conjunto con las estructuras de control. Un grupo que puede ser sorprendentemente útil es el grupo vacío {}. No produce nada en el documento, pero tiene varias aplicaciones dentro del código.

Las estructuras de control (también llamadas *comandos*) son instrucciones dadas por el símbolo reservado \ seguido de algún símbolo o palabra en particular. Permiten hacer cosas que no sean simplemente agregar al documento lo que escribimos en el código. Hay estructuras de control para escribir ciertos símbolos (en especial símbolos que no están en el teclado) y otras que producen ciertas acciones o alteraciones de partes del documento. Los comandos pueden ser básicos de tex, estar definidos por latex, provenir de algún paquete importado o ser definidos por el usuario (ver la sección 5.8). Veremos algunos, pero hay muchísimos más

En caso de que el \sea seguido por un símbolo, como \{ o \\$, la estructura de control termina con el símbolo, independientemente de los que venga después. En caso de que

sea seguido por una palabra, como `\rq` o `\lq`, la estructura de control termina cuando se llega a un símbolo que no sea una letra, como por ejemplo un espacio.

Ejemplos de estructuras de control para producir símbolos que ya vimos son `\lq` y `\rq`, para las comillas. Para poner en el documento algunos de los símbolos reservados hay comandos como `\$` y `\{`. Hay estructuras de control para letras griegas para fórmulas, las cuales solo se pueden usar en modo matemático o modo display. Es una contrabarra y el nombre de la letra griega. Por ejemplo, con `\alpha` se produce α .

Al final del capítulo sobre matemática en la introducción no-tan-corta a latex [4], hay una lista de símbolos matemáticos. En ctan.org/pkg/comprehensive se encuentra otra que tiene más de 20000 símbolos (matemáticos o de uso en texto).

Hay estructuras de control para cambiar la fuente y el tamaño (las primeras de plain tex). Estas instrucciones realizan el cambio desde el punto que se escriben en adelante. Por ejemplo, si escribimos `\sl`, de ese punto en adelante se usa *fuentes de letras torcidas*. Si escribimos `\rm` se vuelve a la fuente básica. Con `\bf` se cambia a **en negrita**. Para cambiar el tamaño hay comandos como `\small`, `\normalsize` o `\large` de modo análogo.

Con las estructuras de control que cambian algo de ese punto en adelante, como las de las fuentes, si se pone adentro de un grupo, solo tiene efecto dentro del grupo. Por lo tanto, una forma de hacer que una sola palabra quede en negrita es con `{\bf palabra}`.

Para una estructura de control dada por una palabra, tex interpreta los espacios que vengan después como delimitadores de la estructura de control y no como un espacio que deba ir en el documento. Por ejemplo, si escribimos `\lq hola`, el resultado es «‘hola’». El espacio entre `\lq` y `hola` se interpreta como que está para terminar la estructura de control y no para agregar un espacio al documento.

En el caso anterior, que los espacios después de estructuras de control no vayan al documento vino bien, pero hay casos en los que quisiéramos tener un espacio. Por ejemplo, entre la comilla de cierre y la siguiente palabra queremos que haya un espacio. Si escribimos `\lq hola \rq dijo Juan`, el resultado es «‘hola’dijo Juan», que tiene el defecto de que la comilla de cierre y la palabra «dijo» están pegadas. Si agregamos más espacios entre `\rq` y `dijo` el problema no se soluciona, ya que tex trata espacios múltiples como uno solo. Una forma de resolverlo es poner un grupo vacío después del comando, el cual no agrega nada al documento, y luego un espacio. Este grupo vacío cumple la función de separar la estructura de control del espacio posterior, por lo que este se trata como un espacio normal. El código es `\lq hola \rq{} dijo Juan` y la salida es «‘hola’ dijo Juan».

Hay estructuras de control que funcionan con uno o más parámetros. Por ejemplo, `\centerline` (comando remanente de plain tex) recibe un parámetro y lo pone centrado en una línea, mientras que `\frac` recibe dos parámetros y los pone en una fracción. En caso de un solo parámetro, el valor es lo primero que viene después de la estructura de control excepto espacios. En caso de dos parámetros, los valores son las primeras dos y así sucesivamente con más parámetros. Por ejemplo, `\centerline A` produce:

A

mientras que si escribimos `$$\frac xy$$` la salida es:

$$\frac{x}{y}$$

Si queremos que la acción se realice con algo más grande que un solo símbolo, hay que usar agrupamiento, de modo que la estructura de control tome al contenido entero del grupo co-

mo lo primero que tiene después. Por ejemplo si hacemos `\centerline {Introducción}` obtenemos:

Introducción

y si hacemos `$$\frac {x+y}{2}$$` obtenemos:

$$\frac{x+y}{2}$$

Los comandos de cambio de fuente tienen versiones con parámetros que son las recomendadas en Latex (los de dos letras, como `\bf` son de plain tex). Por ejemplo, se puede poner una palabra en negrita con `\textbf{palabra}`. En el capítulo «fundiciones y tamaños» de La introducción no-tan-corta a latex [4] se presentan varias fuentes.

Hay estructuras de control que solamente se pueden usar en ciertos modos. Por ejemplo, el comando `\hrule` es para hacer una línea horizontal y solamente se puede usar en modo vertical. Se se pone ese comando dentro de modo horizontal, tex pasa a modo vertical y lo que venga después lo toma como un nuevo párrafo. Por otra parte, el comando `\vrule` es para hacer una línea vertical y solo funciona en modo horizontal. Hay también estructuras de control cuyo comportamiento cambia según el modo. Por ejemplo, `\kern` agrega o quita espacio dado por una medida que le pongamos después (por ejemplo `1cm`, `0.3cm` para agregar o `-1cm`, `-0.3cm` para quitar), pero si ese espacio es horizontal o vertical depende del modo actual. Hay muchas estructuras de control específicas para el modo matemático, de las cuales veremos algunas más adelante.

Latex introduce otro tipo de estructuras de control que son los entornos, dados por `\begin{nombre}` y `\end{nombre}`. Hay entornos para hacer cosas muy variadas. Por ejemplo, el entorno `itemize` permite hacer listas, poniendo adentro el comando `\item` para cada elemento. Por ejemplo, si escribimos lo siguiente:

```
\begin{itemize}
\item Primer item
\item Segundo item
\item Tercer item
\end{itemize}
```

la salida es:

- Primer item
- Segundo item
- Tercer item

Por otra parte, algunos comandos tienen argumentos opcionales que van entre corchetes rectos. Está por ejemplo `\includegraphics[op]{archivo}` para incluir imágenes, cuyos argumentos opcionales permiten entre otras cosas ajustar el tamaño (sección 5.9.2).

Ver la definición de un comando (técnico)

Algunos comandos son primitivos de tex y otros son definidos a partir de comandos más básicos. Con `\show` podemos ver la definición de un comando. Lo que hay que hacer es ponerlo seguido del comando cuya definición queremos ver y ejecutar desde la terminal. Dentro de la salida que latex produce en la terminal se incluye la definición del comando (ver Texbook [2] p. 10, dentro del capítulo 3).

5.6. Cajas y pegamento

Las cajas y el pegamento son conceptos de bajo nivel fundamentales en TeX. Todos los elementos visibles del archivo resultante se procesan con cajas y para determinar los espacios vacíos entre algunos de estos elementos se utiliza pegamento. En general no los manipulamos a mano, pero los comandos que utilizamos funcionan en base a estos conceptos. Por lo tanto, para entender bien el funcionamiento de TeX, hay que saber al menos lo básico sobre cajas y pegamento. Por otra parte, el uso directo de cajas puede ser útil para hacer algunas cosas muy específicas, como el diagrama de la sección 5.2. Para una buena explicación detallada, ver los capítulos “Boxes” y “Glue” del Texbook [2].

Las cajas delimitan (al menos conceptualmente) cierto elemento que va a ir en el documento. Cada caja contiene una lista de elementos, que pueden ser otras cajas. Las cajas pueden ser horizontales o verticales, según cómo se organizan sus elementos. Cada letra conforma una caja básica. Cada palabra es una caja horizontal formada por sus letras. Cada línea es una caja horizontal formada por las palabras y los espacios. Cada párrafo es una caja vertical formada por sus líneas y el espacio entre ellas. En general una página es una caja vertical formada por distintos elementos, que pueden ser líneas, fórmulas matemáticas, o espacios vacíos, entre otros.

Recordando los modos de tex que vimos en la sección 5.4, el modo vertical es en el que está cuando arma una caja vertical, mientras que el modo horizontal es en el que está cuando arma una horizontal.

Cada fórmula matemática es una caja que se forma combinando de distintos modos las cajas más chicas de las subfórmulas. Por ejemplo, un cociente será una caja vertical formada por el numerador, la barra de división y el denominador (tal vez junto con elementos para las separaciones). El numerador y el denominador son en sí otras cajas, que pueden ser horizontales formadas por símbolos. Por ejemplo en

$$\frac{x+y}{2\alpha}$$

tenemos una caja vertical formada por las subcajas horizontales de $x+y$, de 2α y por la barra de división.

En general todo lo que TeX genera se descompone en cajas de alguna forma. Saber esto puede ayudar a comprender mejor los resultados, pero en general no es necesario conocer profundamente los detalles.

Elementos básicos de cajas

Enfoquémonos en algunos de los elementos básicos de cajas, es decir, los que no son otras cajas. Existen otros elementos básicos además de los que vemos aquí.

Cada símbolo (por ejemplo una letra) es un elemento básico. Otros elementos básicos son las reglas y los espacios fijos llamados **kern**. Otro elemento básico es el pegamento, el cual veremos más adelante, pero que esencialmente es espacio variable.

Las reglas son líneas, las cuales pueden ser verticales u horizontales. Las líneas verticales se generan con `\vrule`, por ejemplo « \rangle ». Se pueden usar solamente como parte de cajas horizontales y la longitud se determina a partir de la altura de la caja a la que pertenecen (en el ejemplo anterior, la caja horizontal a la que pertenece es el renglón). Las líneas horizontales, por el contrario, se generan con `\hrule` y solamente pueden

formar parte de cajas verticales. La longitud de la línea es el ancho de la caja vertical a la que pertenece. Después de este párrafo hay una línea horizontal, que pertenece a la caja vertical que forma la página.

Hay distintas formas de poner espacios fijos en una caja. Uno de ellos es el comando `\kern` seguido por una dimensión. El resultado es la cantidad dada por la magnitud de espacio en blanco. Dentro de una caja vertical, el espacio es vertical, mientras que dentro de una caja horizontal, el espacio es horizontal. Las dimensiones son números con la parte fraccionaria separada por un punto seguidos de una unidad, la cual puede ser `cm`, `in` o `em`, entre otras (ver el capítulo “Dimensions” del Texbook [2]). Si el número es `0.d`, se puede omitir el cero, por ejemplo en `\kern .3em`.

La dimensión puede tener signo negativo, lo cual hace que se quite espacio. En otras palabras, acerca lo que tiene antes y lo que tiene después, pudiendo llegar a hacer que se superpongan. Un ejemplo es `\kern -.3em`.

A modo de ejemplo, `{Este \kern 1em es\kern -3em un ejemplo.}` produce

Este esin ejemplo.

Por supuesto que en general no tiene mucho sentido usar estas cosas en medio de texto. Para el diagrama de la sección 5.2 fueron útiles y tal vez aparezca en ejemplos posteriores. Cabe aclarar que normalmente los comandos de alto nivel de latex están hechos de modo que no haga falta ajustar los espacios de esta forma.

Hay otros comandos de espaciado, de los que hablaremos más adelante. Por ejemplo, `\quad` es uno muy común para producir cierta cantidad fija de espacio horizontal.

Creación de cajas compuestas

Para crear explícitamente una caja horizontal se usa el comando `\hbox` y para crear una vertical `\vbox`, los cuales arman una caja con lo que venga después (que en general es un grupo, delimitado por corchetes). Dentro del contenido de una caja se pueden poner otras cajas. Por ejemplo, al crear una caja vertical, es común que adentro cajas horizontales, las cuales quedarán una arriba de la otra. Considerar el siguiente ejemplo (recordar que el código fuente está disponible en el git del curso, cuya página se indica al principio de este capítulo).

lo de arriba
y lo de abajo

Se pueden poner cajas en el modo matemático y el modo display. Al hacer eso, dentro de la caja se pasa a modo vertical si es vertical y a modo horizontal si es horizontal. En cualquiera de los dos casos, se escribe texto de modo usual. Esto es una forma sencilla de poner texto dentro de una fórmula matemática. Por otra parte, dentro del modo display puede servir para poner texto centrado en medio de la página, como se hizo en el ejemplo de arriba y en el siguiente.

Ejemplo de texto centrado con `hbox` en modo display.

Las cajas verticales hechas con `\vbox` se alinean según el elemento de más abajo. Para hacer una caja que se alinea según el de arriba se usa `\vtop` y para una que se alinie por

el centro, `\vcenter`, siendo que este último solo se puede usar en modo matemático. En el siguiente ejemplo (no muy bello) usamos los tres tipos.

El supremo ínfimo es el mínimo máximo de las cotas superiores inferiores.

Si ponemos texto suelto dentro de una caja vertical funciona como un párrafo. Para hacer una caja vertical que sea un párrafo más chico, dentro de esa caja se puede cambiar el `\hsize`, que indica el ancho del texto. Por ejemplo, podemos hacer `\hsize=.4\hsize` para que sea el 40 % (por el funcionamiento de los grupos, fuera de la caja se mantiene el valor original). En el siguiente ejemplo hacemos eso, junto con reglas para meter el párrafo en una caja visible.

Este es un párrafo de ejemplo que queda metido dentro de una caja. Seguramente se puede hacer con algún paquete, pero ¿quién nos quita la diversión?

Ejercicio: modificarlo para que los bordes laterales también queden dos puntos separados del párrafo. Si solamente se agregan los espacios a los lados, las esquinas de la caja no quedan bien cerradas.

Se puede fijar el ancho de una caja horizontal con `\hbox to ancho`. Con una caja vertical se puede definir la altura del mismo modo. Si ponemos un ancho mayor al tamaño natural de los contenidos, `tex` intenta espaciarlos para que quede completo, de acuerdo al funcionamiento del pegamento (que veremos a continuación). Por ejemplo. `\hbox to 2in {Ejemplo de texto.}` da:

Ejemplo de texto.

Puede ser útil en conjunto con pegamentos infinitos, cosa que veremos a continuación.

Pegamento

El pegamento son espacios variables que suelen formar parte de cajas. Si el espacio es horizontal o vertical depende del tipo de caja. Cada pegamento tiene un tamaño natural, una compresibilidad y una extensibilidad. La compresibilidad dice qué tanto se puede achicar, mientras que la extensibilidad qué tanto se puede agrandar.

Un ejemplo son los espacios entre las palabras de cada línea de un párrafo. Tex separa el párrafo en líneas, que son cajas horizontales y en cada una de esas líneas, los espacios se ajustan de acuerdo a la extensibilidad y compresibilidad para que el largo total sea adecuado. En caso de que no haya ninguna solución que respete la compresibilidad y extensibilidad de los pegamentos, tex genera una caja horizontal que se pasa del largo estipulado y genera una advertencia (*warning*) de *overfull hbox*.

Quando ocurre un overfull hbox, lo mejor es escribir las cosas de modo distinto para evitar el problema. Por otra parte, se puede aumentar la tolerancia de tex, haciendo que permita estirar los pegamentos más que la extensibilidad (lo que nunca se hace es achicarlos más que lo que permite la compresibilidad). Esto hace que las palabras puedan

separarse más, lo cual puede solucionar el problema a costo de que el párrafo se vea más feo, al tener las palabras demasiado sueltas. Para cambiar la tolerancia se hace `\tolerante=n`. La tolerancia por defecto es 200.

Si el contenido de una caja horizontal no alcanza para completar el tamaño estipulado, se estiran los pegamentos por encima de la extensibilidad y se produce un warning de *underfull hbox*. Esto es lo que pasa por ejemplo si hacemos `\hbox to ancho {contenido}` con un contenido que no alcanza para el ancho dado. En caso de que en el contenido no haya pegamento, no se estira nada e igual se produce el warning.

En las fórmulas también puede haber pegamento horizontal. Por ejemplo, $xy + z = w$ tiene pegamento a los lados del símbolo de operación $+$ y del símbolo de relación $=$.

También hay pegamento vertical. No se utiliza para las líneas de texto, ya que estas se tratan de modo distinto, pues en general se desea que estén equiespaciadas, pero sí se utiliza al rededor de los títulos y de las fórmulas matemáticas en modo display, entre otras cosas.

Los comandos `\smallbreak`, `\medbreak` y `\bigbreak` producen pegamento vertical, de tres tamaños distintos como sugieren los nombres. El chico es de $3\text{pt}\pm 1$, el mediano es el doble y el grande es el doble del mediano. Entre este párrafo y el siguiente hay un `\smallbreak`.

Un ejemplo conceptualmente interesante y además útil es el de pegamento infinito. Se trata de pegamento con extensibilidad infinita. La versión horizontal es `\hfill` y la vertical es `\vfill`. Si colocamos pegamento infinito en una caja, los otros pegamentos quedan todos en su tamaño natural y el pegamento infinito se extiende lo que haga falta. En caso de que hayan varios pegamentos infinitos, el espacio se reparte equitativamente entre ellos.

Una aplicación común es la de correr a la izquierda, correr a la derecha o centrar algo. Para correr a la izquierda ponemos un `\hfill` a la derecha. Para correr a la derecha ponemos el `\hfill` a la izquierda. Para centrar, ponemos un `\hfill` a la izquierda y otro a la derecha. Considerar los siguientes ejemplos.

$$\left| \begin{array}{c} \text{Un texto} \\ \text{Un texto más grande} \end{array} \right| \quad \frac{1}{x+y} \quad \frac{1}{x+y}$$

Usos de registros de cajas (técnico)

Tex tiene registros en los que se pueden guardar cajas para ciertos usos. Son como variables de un lenguaje de programación. En el capítulo “How TeX Makes Lines into pages” del Texbook [2] se explican los registros de modo detallado. Los registros de cajas permiten ver cómo es la estructura interna de la caja y acceder a sus dimensiones.

Para ingresar una caja en el registro 0 alcanza `\setebox0 = caja`. Luego de hacer esto, podemos por ejemplo acceder a su ancho con `\wd0`. Para ver la estructura de la caja, hay que poner primero el comando `\showoutput`, luego `\showbox0` y ejecutar desde la terminal. En el texto de salida de la terminal se incluye la estructura interna de la caja. En el capítulo “Boxes” del Texbook [2] hay un ejemplo.

Conocer el ancho de una caja puede ser útil para hacer que otra caja tenga exactamente el mismo. Considerar el siguiente ejemplo.

Línea
Línea más grande

Comentario final sobre cajas y pegamento

Lo que acabamos de introducir en esta sección son elementos básicos del funcionamiento de Tex. Las instrucciones más complejas se definen a partir de conceptos como estos. Haciendo una analogía con los lenguajes de programación, las cajas y el pegamento son como instrucciones básicas del procesador. Si bien saber armar cajas explícitamente puede ser útil para hacer ciertas cosas muy específicas, en general es preferible utilizar las instrucciones de más alto nivel, ya que lo que hay que escribir es menos y el código queda más fácil de entender.

5.7. Escritura matemática

En esta sección veremos funcionalidades del modo matemático y el modo display. Lo que explicaremos son funcionalidades básicas de tex, cosas de latex y cosas agregadas por paquetes, en particular por `amsmath`, que es un paquete muy común para escritura matemática (documentación: ctan.org/pkg/amsmath).

Los capítulos “Typing Math Formulas”, “More about Math”, “Fine Points of Mathematical Typing” y “Displayed Equations” del Texbook [2] tienen lo central que se presenta aquí con explicaciones menos escuetas, varios ejemplos y ejercicios (cuyas soluciones están en un apéndice), por lo que puede aportar mucho. Un detalle a tener en cuenta es que algunas cosas, como las matrices y las ecuaciones alineadas, en latex con el paquete `amsmath` se escriben distinto, pero la idea de cómo funcionan es análoga.

Fuentes y texto en fórmulas

Dentro de fórmulas se pueden cambiar las fuentes igual que en texto. Por ejemplo, para obtener la fórmula $\cos(x)$ se puede escribir `\rm cos(x)`. También puede ser útil usar fuente en negrita, por ejemplo en $\alpha \mathbf{x}^T \mathbf{T} \mathbf{x}$ (ejemplo que también usa fuente `rm`). La fuente por defecto en matemática es `\mit`. No es lo mismo que la itálica `\it`, en particular por el espaciamiento de las letras (comparar *Hom* con *Hom*). Para nombres de funciones es muy común que se use la fuente `rm`, por lo que hay comandos específicos. Por ejemplo, otra forma de escribir el coseno de x es `\cos(x)`.

Hay fuentes que no hemos mencionado que suelen ser útiles en fórmulas. Una de ellas es `\cal`, la cual funciona con letras mayúsculas, dando resultados como \mathcal{R} . El paquete `amsfonts` agrega fuentes dadas por los comandos `\mathbb` y `\mathfrak`. Un ejemplo de la primera es \mathbb{R} y uno de la segunda es \mathfrak{s} . La primera es solo para mayúsculas. Estos comandos funcionan con parámetros, es decir, aplican esa fuente solamente a lo que venga después. Para aplicarlos a varios símbolos seguidos hay que usar grupos, pero justo con estas fuentes no es común.

En matemática es común que se coloquen distintos símbolos sobre letras, como acen-tos. En general son con un comando antes de la letra. Por ejemplo `\dot x` da \dot{x} y `\vec x` da \vec{x} . Ver la lista de símbolos matemáticos de la introducción no-tan-corta a latex [4], al final del capítulo sobre fórmulas matemáticas. Por otra parte, el símbolo $'$ en modo matemático pone *prima* a lo anterior. Por ejemplo `x'` da x' y `x''` da x'' . También se puede hacer con `x^\prime`.

Para cambiar a modo texto en una fórmula se puede usar `\hbox`, como ya vimos. Hay otro comando que es `\text`, que tiene en cuenta ciertos aspectos del tamaño de fuente (si lo ponemos en un subíndice, el texto queda chico, por ejemplo). Escribir texto adentro de una fórmula no es lo mismo que solamente cambiar la fuente. En el texto los espacios se tienen en cuenta mientras que en las fórmulas no, independientemente de la fuente. Por ejemplo, `\$x{\rm tal que }P\$` produce $x\text{talque}P$ mientras que `\$x\text{\textit{tal que }}P\$` produce x tal que P .

Subíndices y supraíndices

Para poner subíndices y supraíndices se usan los símbolos reservados `«_»` y `«^»`. Solo se permiten en modo matemático. El símbolo `«_»` pone lo que viene después como subíndice de lo que tiene antes y `«^»` funciona de modo análogo. El resultado de `x_2` es x_2 , mientras que el de `x^2` es x^2 . No obstante, cuando hay subíndices y supraíndices juntos, el comportamiento es distinto. Al ingresar `x_1_x2^x3` o `x_1^x3_x2`, se aplican x_2 como subíndice y x_3 como supraíndice, ambos a x_1 . Esto es así para facilitar el ingreso de símbolos con subíndices y supraíndices a la vez. Por ejemplo, `x_0^2` genera x_0^2 .

Usando agrupamiento se puede poner estructuras más complejas como subíndices o supraíndices. Lo más básico es poner más de un símbolo, pero también se puede poner por ejemplo un subíndice que tenga otro subíndice. Ver los siguientes ejemplos.

$$2^{10} \quad x_{i_0} \quad f^{t*} \quad e^{\frac{(x-\mu)^2}{2\sigma}}$$

Por otra parte, se puede usar agrupamiento en la base. Esto puede cambiar la posición de los sub/supraíndices. Considerar las siguientes expresiones

$$((x^2)^3)^4 \quad ((x^2)^3)^4$$

En la primera expresión, los supraíndices 3 y 4 se aplican a símbolos `«)»`, por lo que quedan en esa altura. Por otra parte, en la segunda, se ponen grupos para que los supraíndices se coloquen teniendo en cuenta la altura de todo el grupo correspondiente.

Se pueden aplicar sub/supraíndices a grupos vacíos. Esto se puede usar por ejemplo para poner sub/supraíndices a la izquierda de un símbolo, entre otras cosas. Las siguientes expresiones usan sub/supraíndices aplicados a grupos vacíos.

$${}_4^3x_1^2 \quad \Gamma_i^{jk}{}_l$$

Algunos símbolos de `tex` son *operadores grandes*. Un ejemplo es la sumatoria \sum , que se escribe con `\sum` (no es lo mismo que `\Sigma`, la letra sigma mayúscula). Lo que caracteriza a los operadores grandes es que en modo display los subíndices y supraíndices van exactamente abajo y arriba del símbolo. La idea es usarlos para indicar extremos o dominios. Para escribir la sumatoria de i desde 0 hasta n , se escribe `\sum_{i=0}^ni`. En modo matemático de texto queda $\sum_{i=0}^n i$, mientras que en modo display queda

$$\sum_{i=0}^n i.$$

Otro ejemplo:

$$\bigcup_{x \in A} x$$

raíces, barras y corchetes con comentarios

Para hacer una raíz cuadrada se usa el comando `\sqrt`, que recibe un parámetro que es lo que va adentro. Por ejemplo:

$$\sqrt{1 + \sqrt{x}}.$$

Notar que el tamaño del símbolo de raíz se ajusta a lo que hay adentro.

Para la raíz enésima, se puede usar `\root n \of x`. Por ejemplo, $\sqrt[3]{x+y}$.

Para poner una barra arriba o abajo de algo se puede usar `\overline` o `\underline`, con un parámetro que es a lo que queremos ponerle la barra. Por ejemplo, $\overline{x+y}$.

Podemos también sacar corchetes para arriba o abajo de cierta parte de una fórmula y agregar explicaciones. Esto se hace con los comandos `\overbrace` y `\underbrace`. Estos comandos por sí solos solamente ponen el corchete arriba o abajo. Lo que los hace fáciles de usar para poner comentarios es que son operadores grandes, por lo que los subíndices van abajo y los supraíndices arriba. Lo siguiente son dos ejemplos.

$$\begin{array}{c} \text{par} \\ \overbrace{x+y} = z \\ \\ g^n * g^m = \underbrace{\overbrace{g * g * \dots * g}^{n \text{ veces}} * \overbrace{g * g * \dots * g}^{m \text{ veces}}}_{n+m \text{ veces}} = g^{n+m} \end{array}$$

Veamos un detalle técnico sobre las raíces. Cuando el contenido tiene profundidad (algo que va por debajo de la línea de base, como la cola de «g»), el símbolo de raíz baja (al menos un poco). Observar $\sqrt{x}\sqrt{g}$. Para que se ignore la profundidad, se puede usar el comando `\smash[b]`, que es un agregado de `amsmath` sobre el comando `smash` de `tex`. Con esto el resultado es $\sqrt{x}\sqrt{g}$.

Fracciones, combinaciones y ordenamiento vertical

Como ya vimos, las fracciones se hacen con `\frac{}{}`. También está la posibilidad de escribirlas con una barra, como x/y , que en algunos casos puede quedar mejor.

Por otra parte, el comando `\binom` es para poner una cosa arriba de otra entre paréntesis, como en las combinaciones. Por ejemplo $\binom{n}{k}$.

El comando `\atop` de `plain tex` sirve para poner una cosa arriba de otra sin nada más, como una fracción sin la barra. También se puede hacer con `\genfrac`, de `amsmath`, pero es un poco más rebuscado. El funcionamiento de `atop` es que pone todo lo que viene antes arriba y lo que viene después abajo. Para que no se aplique a toda la fórmula se puede usar agrupamiento. Por ejemplo:

$$\begin{array}{c} y \\ x \\ z \end{array}$$

Usar esto tira un warning (advertencia) de `amsmath` porque al parecer es mejor usar `genfrac`. En la sección 5.8 veremos cómo inventar un comando más fácil a partir de `genfrac` que haga lo mismo que `atop`.

Estilos (y tamaño)

En modo matemático hay cuatro estilos: **display**, **texto**, **script** y **scriptscript**. El estilo **display** es el de modo matemático **display**, el estilo **texto** es del des modo matemático usual, el estilo **script** es para cosas chicas como los subíndices y el **scriptscript** es para cosas aún más chicas, como subíndices de subíndices. En realidad hay otros 4 más con diferencias sutiles (ver el capítulo “More about math” del Texbook [2]).

En una misma fórmula, automáticamente se va cambiando entre los distintos estilos en distintas partes. Los subíndices, supraíndices y las fracciones cambian a estilos menores (salvo que ya se esté en estilo **scriptscript**). En el capítulo “More about math” del Texbook [2] se explica bien cómo son las reglas automáticas de cambio de estilo.

Cuando el estilo decidido automáticamente no es el que deseamos, podemos fijarlo con los comandos `\displaystyle`, `\textstyle`, `\scriptstyle` y `\scriptscriptstyle`. Cada uno de esos comandos hacer que se cambie al estilo correspondiente de ese punto en adelante en el grupo actual. Un ejemplo artificial: $x + y + z$ (notar que el cambio de estilo también afecta el tamaño del «+»).

Veamos algunos ejemplos en los que el cambio de estilo es útil. Supongamos que queremos poner una fracción de varios niveles. Por defecto queda

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4 + 1}}}}$$

pero podríamos querer que todos los niveles tengan tamaño normal. Lo podemos hacer poniendo varios `\displaystyle`.

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4 + 1}}}}$$

Los comandos `\strut` que hay en la fórmula son para ajustar el espacio vertical (sino los 1 quedan muy cerca de las barras), lo veremos en la sección de espaciado.

El siguiente ejemplo es una aplicación de `\atop` para un índice en tres líneas. Por defecto el estilo se vuelve **scriptscript**, pero podría decirse que queda mejor con **script**. A la izquierda ponemos como queda automáticamente y a la derecha cambiando el estilo.

$$\sum_{\substack{1 \leq i \leq p \\ 1 \leq j \leq q \\ 1 \leq k \leq r}} a_{ij} b_{jk} c_{ki} \qquad \sum_{\substack{1 \leq i \leq p \\ 1 \leq j \leq q \\ 1 \leq k \leq r}} a_{ij} b_{jk} c_{ki}$$

De modo *artesanal*, se puede hacer con cajas.

$$\sum_{\substack{1 \leq i \leq p \\ 1 \leq j \leq q \\ 1 \leq k \leq r}} a_{ij} b_{jk} c_{ki}$$

Aquí ajustamos el espacio entre las líneas del subíndice con `\kern` negativo. Agregando lo siguiente, hacemos también que las desigualdades estén bien alineadas.

$$\sum_{\substack{1 \leq i \leq p \\ 1 \leq j \leq q \\ 1 \leq k \leq r}} a_{ij} b_{jk} c_{ki}$$

Delimitadores

Con «delimitadores» nos referimos a símbolos que se usan para delimitar partes de una fórmula. Algunos son «(», «)», «|», «{» y «}»; en la lista de símbolos matemáticos de la introducción no-tan-corta a latex [4] (al final del capítulo sobre fórmulas matemáticas) hay un cuadro con todos los delimitadores que vienen por defecto.

Un buen uso de delimitadores ayuda a que la estructura de una fórmula sea clara a primera vista. En tex la clave está en el uso de **distintos tamaños**. Tenemos comandos `\big`, `\Big`, `\bigg` y `\Bigg` que permiten aumentar el tamaño del delimitador en distinta medida. Se escribe el comando seguido del delimitador. Lo siguiente son «(» y «|» con los distintos tamaños.

$$\left(\left(\left(\left(\left(\left| \right| \right| \right| \right| \right| \right| \right| \right| \right| \right|$$

Hay versiones de los comandos en las que indicamos si el delimitador es de izquierda, derecha o medio, las cuales se escriben igual pero con una `l`, `r` o `m` al final. Los delimitadores del medio automáticamente agregan espacio a ambos lados.

Por otra parte, están los comandos `\left` y `\right`, que automáticamente ajustan los delimitadores al tamaño de lo que tienen entre medio. Nuevamente los delimitadores a usar se escriben después de los comandos. Es importante que los `\left` y `\right` estén emparejados (o sea, que cada `\left` tenga su `\right`), sino no queda definido el contenido al que se deben ajustar los delimitadores. Más generalmente, deben estar *balanceados* respecto a otros «entornos». Algo como `\left(\cdots\right)\cdots\right` da error. Sí podríamos poner `\left(\cdots\right)\cdots\right` o `\left(\cdots\left\{\cdots\right\}\cdots\right)`. Si no queda claro, investigar sobre «paréntesis balanceados».

Veamos algunas aplicaciones.

Un primer caso es el de paréntesis anidados. Tomemos como ejemplo $((x+y)z+5)w$. Es bueno de alguna forma diferenciar los paréntesis exteriores de los interiores, de modo que la estructura sea clara a primera vista. Alcanza con usar `\big`: $((x+y)z+5)w$. La diferencia es poca, de modo que sigue entrando bien en la línea, pero salta a la vista. Está también la posibilidad de escribir $[(x+y)z+5]w$, pero en general es mejor resolverlo con tamaño de paréntesis (así podemos dejar los corchetes rectos reservados para otras cosas, como intervalos). El mismo principio puede servir también con valores absolutos. Comparar $||x|-|y||$ con $|\left|x\right|-\left|y\right||$. Notar que si usamos `\left` y `\right`, los delimitadores de afuera quedan de igual tamaño que los de adentro.

Otro caso es cuando contenido ocupa más espacio vertical. Comparar las siguientes escrituras de la misma expresión.

$$\left(x+\frac{y}{2}\right)z \qquad \left(x+\frac{y}{2}\right)z$$

Usamos `\left` y `\right` para que tex determine por sí solo el tamaño adecuado.

En algunas situaciones como la de conjuntos definidos por comprensión o probabilidad condicional, tenemos delimitadores con un separador entre ellos. En estos casos conviene usar las versiones «izquierda», «medio» y «derecha» del comando de algún tamaño, como `\bigl`, `\bigm` y `\bigr`.

$$A = \{x \in \mathbb{R} \mid x^2 < 3(x+4)\} \quad \mathbb{P}(A \cup B \mid C)$$

Notar que se agrega espacio al rededor de los delimitadores del medio. Para el conjunto, se podría considerar también usar también el tamaño `\Big`.

Como ya se mencionó, los `\left` y `\right` deben estar emparejados. Sin embargo, no hay restricciones sobre los delimitadores que se ponen, es solo para tener un contenido definido al que ajustarlos. Se puede poner algo como `\left)\cdots\right|`. Además, se puede poner delimitador de un solo lado usando como delimitador del otro lado un punto, lo cual `tex` interpreta como delimitador vacío. Por ejemplo, con `\left\{\cdots\right.` podemos hacer lo siguiente.

$$\begin{cases} x + 2y = 1 \\ x - y = 0 \end{cases}$$

Cabe mencionar que en sistemas como este en general se hace que las igualdades queden alineadas. Una forma de lograrlo fácilmente (sin hacer un trabajo artesanal con tamaños de cajas y `\kerns`) es con el entorno `aligned`, que veremos más adelante.

Espaciamento

Tenemos funcionalidades que permiten ajustar el espaciamento, tanto horizontalmente como verticalmente. Comenzamos con consideraciones sobre espaciamento horizontal dentro de una fórmula.

Al procesar una fórmula, `tex` automáticamente coloca distintos espacios entre distintos símbolos (recordar que los espacios y saltos de línea del código son ignorados). Por ejemplo, si detecta que un símbolo es de operación, le agrega espacio a ambos lados y si detecta que es de relación, le agrega más espacio a ambos lados. Considerar $xy + z = w$. Cabe destacar que la forma como se trata un símbolo depende del contexto (lo que tenga a los lados). Notar que en el siguiente caso no se agrega espacios a los lados del símbolo de suma: $f(x+)$. Por otra parte, las comas agregan un poco de espacio después pero no antes y los puntos no agregan ningún espacio: $f(x, y, a.b)$.

En general el espaciamento automático es bueno, pero hay situaciones en las que no es el ideal, por lo que se puede cambiarlo. Para evitar que se agregue espacio al rededor de algún símbolo alcanza encerrarlo en un grupo. Al hacer esto, pasa a ser tratado como un símbolo normal. Por ejemplo, `\mathbf{x}+\mathbf{y}` da $x+y$. Por otra parte, hay comandos específicos para aumentar o disminuir el espaciamento. Para insertar espacio, de menor a mayor, tenemos los comandos `\,`, `\,`, `\:` y `\;`. El primero es el espacio automático después de una coma, el segundo (que también puede ser `\>`) es el de una operación y el tercero es el de una relación. Por otra parte, para sustraer espacio tenemos `\!`. En las siguientes fórmulas se usan (mirar el código fuente).

$$\int_0^x f(t) \, dt \quad \iint_D dx \, dy$$

También se puede usar `\kern` con una dimensión positiva o negativa para elegir el espaciamiento con más flexibilidad, pero en general conviene usar los comandos anteriores, ya que son como espacios “estandarizados”.

Para separar distintas fórmulas dentro de una línea, se suelen usar `\quad` o `\qquad`, siendo el segundo el doble que el primero. Las dos integrales de arriba están separadas por un `\qquad`. Todos los comandos de espaciamiento que mencionamos se pueden usar también fuera del modo matemático (específicamente, en modo horizontal, porque el espaciamiento es horizontal).

Hay un comando bastante particular que afecta tanto al espaciamiento horizontal como vertical. Se trata de `\phantom`, el cual recibe un parámetro y lo que hace es colocar la caja de ese parámetro, pero sin mostrarlo. A efectos de espaciamiento, tanto horizontal como vertical, es como si estuviera, pero no se ve. Sirve cuando queremos poner exactamente el espacio de cierta cosa.

Pasemos ahora a centrarnos en el espaciamiento vertical. Por ejemplo en las fracciones, a veces ocurre que cierto símbolo queda muy cerca de una barra, por ejemplo:

$$\frac{1}{2 + \frac{3}{4}}.$$

Una forma sencilla de arreglarlo, es haciéndole creer a tex que cierta parte de la fórmula es más alta de lo que realmente es. En este caso lo haríamos con el 3. Una forma de lograrlo sería poner un `\phantom` de un elemento más alto, como un «(», pero esto también agregaría espacio horizontal. Podemos hacer que el fantasma solamente aporte al espacio vertical con `\vphantom`. Si ponemos un `\vphantom(` al lado del 3, el resultado es

$$\frac{1}{2 + \frac{3}{4}}.$$

El comando `\strut` que usamos en la parte sobre estilos es esencialmente `\vphantom(`.

Por otra parte, para quitar espaciamiento vertical tenemos el comando `\smash`, el cual recibe un parámetro y hace que este tenga altura y profundidad nula, sin cambiar cómo se ve. En conjunto con `\vphantom`, permite ingresar cualquier cosa con la altura y profundidad de cualquier otra cosa. Amsmath agrega `\smash[b]`, lo cual en lugar de anular la altura y la profundidad, solamente lo hace con la profundidad. En la sección de raíces vimos una aplicación.

Matrices

Para crear matrices tenemos el entorno `matrix`. En plain tex es un comando, que se usa como `\matrix{}`, pero en latex con amsmath pasó a ser un entorno, que se usa con `\begin{matrix}` y `\end{matrix}`. Dentro del entorno se escribe el contenido de la matriz fila por fila. Es decir, primero se escribe la primera fila, luego la segunda y así sucesivamente hasta terminar. Para cambiar de entrada en una misma fila se usa el carácter de alineación `&` (uno de los caracteres reservados de tex), mientras que para pasar

a la siguiente fila se usa `\\` (o `\cr`). Por ejemplo:

```
\begin{matrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{matrix}
```

Notar que para la última fila no hace falta poner `\\`. Por otra parte, el código está escrito de modo que se vea la estructura de la matriz, pero no es obligatorio. Si se quiere se puede poner en una línea como `\begin{matrix}1&2&3\\4&5&6\\7&8&9\end{matrix}`.

El entorno no pone delimitadores a la matriz porque con `\left` y `\right` podemos fácilmente poner los que queramos.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \left| \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \right|$$

En el tercer ejemplo se agregaron espacios chicos entre las barras y el contenido porque sin ellos se veía algo apretado. Para el caso de paréntesis se puede usar el entorno `pmatrix` que los agrega solo.

La matriz puede tener una única fila o una única columna, quedando como un vector. Por ejemplo:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \quad (1 \quad 2 \quad 3 \quad 4)$$

Si hay más de una fila, todas deben tener la misma cantidad de entradas, o equivalentemente, misma cantidad de `&` en el código. Si en una entrada no queremos poner nada, en su lugar del código no ponemos nada, o bien espacios.

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

Para hacer matrices o vectores genéricos, están los comandos `\ldots`, `\vdots` y `\ddots`, que producen puntos suspensivos en distintas direcciones. El primero es para horizontales bajos, si se quiere que estén centrados, está `\cdots`.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

El tamaño de las columnas se ajusta automáticamente y las entradas se centran.

$$\begin{pmatrix} 1 & x+y & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Si se quiere que algunas columnas se justifiquen a la izquierda o derecha, en lugar de estar centradas, se puede usar el entorno `array`. En la introducción no-tan-corta a latex [4] se lo presenta, junto con una aplicación para la definición de funciones por casos.

Alineamiento

A veces se requiere separar una fórmula en más de una línea, por ejemplo cuando se debe mostrar una cuenta larga. En estos casos lo estándar es alinear según las igualdades o desigualdades, como en el siguiente ejemplo de cálculo.

$$\begin{aligned} \frac{d}{dx} \left(\int_{\sin(x)}^{x^2+1} h(t) dt \right) &= h(x^2+1) \frac{d}{dx}(x^2+1) - h(\sin(x)) \frac{d}{dx}(\sin(x)) \\ &= 2x h(x^2+1) - \cos(x) h(\sin(x)) \end{aligned}$$

Esto se logra con el entorno `align` o `align*`, siendo la diferencia entre los dos que el primero numera las líneas y el segundo no (en general en latex se usan asteriscos con el sentido “no enumerar”). Sirve en general para alinear fórmulas en puntos deseados. La sintaxis es la misma que para las matrices, con `&` y `\\`, pero el sentido de los símbolos es distinto. Los `&` se usan para marcar puntos de alineación y los `\\` para pasar a la siguiente línea. Para alinear por igualdades o desigualdades, en cada línea se coloca un `&` antes del símbolo. En todas las líneas se deben colocar la misma cantidad de `&`, pero no es obligatorio que hayan cosas entre ellos (al igual que con las matrices). Por otra parte, cabe mencionar que si se pone un `\\` al final, se genera una línea vacía (a diferencia de con las matrices, en las que no cambia el resultado).

Cuando una expresión es muy larga, lo usual es partirla en un símbolo de operación. Se suele hacer de modo que quede más adelante que los símbolos de relación, por ejemplo con `&\qquad`, lo cual agrega espacio después del punto de alineamiento. Complejizemos un poco la cuenta anterior para hacer un ejemplo.

$$\begin{aligned} \frac{d}{dx} \left(\int_{\sin(x)}^{x^2+1} h(t) dt + e^{x^2} \right) &= h(x^2+1) \frac{d}{dx}(x^2+1) - h(\sin(x)) \frac{d}{dx}(\sin(x)) \\ &\quad + 2xe^{x^2} \\ &= 2x h(x^2+1) - \cos(x) h(\sin(x)) + 2xe^{x^2} \end{aligned}$$

En la segunda línea se usó `\smash` para que la altura del exponente no haga que quede muy separada de la primera. Es un poco discutible si en este caso es necesario separar o si quedaba bien igual con dos líneas; está a modo de ejemplo.

El uso de más de un caracter de alineamiento por línea es para tener más de una cosa alineada por línea, como es de esperarse, pero no es tan directo como que se alineen todos los puntos con caracteres de alineación, sino que los `&` alternan entre alineación y separación. En el caso de tres caracteres de alineación, el primero y el último se alinean, mientras que el del medio indica en dónde separar las cosas. Esto permite poner varias ecuaciones en la misma línea, como en el siguiente ejemplo, en el que se puso un caracter de alineamiento antes de cada igualdad y otro separando las distintas ecuaciones de cada

línea.

$$\begin{array}{ll} X + Y = Z & A + B = C \\ X' + Y' = Z' & A' + B' = C' \end{array}$$

Una aplicación interesante es la de agregar explicaciones a ciertas líneas.

$$\begin{array}{ll} X + Y = Z & \text{por cierto motivo} \\ = Z' & \text{por cierto otro motivo} \end{array}$$

Notar que se deben poner dos caracteres de alineamiento antes de cada texto, uno para la separación con las ecuaciones y otro para que los alinee. En fórmulas con más `&`, también alternan entre alineamiento y separación.

Este entorno no es para usarse como componente de una fórmula más grande. Para esto está el entorno `aligned`. Un ejemplo de aplicación es el de sistemas de ecuaciones con igualdades alineadas, lo cual es parte de una fórmula más grande, ya que como mínimo tiene el corchete a la izquierda.

$$\left\{ \begin{array}{l} x + y + z = 1 \\ x - y + 2z = 2 \end{array} \right.$$

Usando múltiples caracteres de alineación, se puede alinear también las variables. Hay que pensar bien dónde poner los `&`, que alternan entre alineación y separamiento.

$$\left\{ \begin{array}{llll} x & +y & +z & = 1 \\ x & -y & +2z & = 2 \end{array} \right.$$

También se podría optar por alinear los símbolos de operación o hacerlo a la vez con ambas cosas, con distintas cantidades de caracteres de alineación. Ejercicio: hacer que los símbolos de operación también queden alineados agregando solamente un `\phantom`.

5.8. Macros

Tex provee de una funcionalidad de definir nuevos comandos, llamados *macros*. Tanto latex como los distintos paquetes se hacen en base a macros. Por otra parte, cualquier usuario de latex puede definir sus propios comandos y usarlos en sus códigos.

El primer beneficio claro de definir comandos propios es el de simplificar el código. Si hay algo que va a aparecer muchas veces, puede valer la pena definir un comando específico que sea más fácil de escribir.

La forma de definir un nuevo comando en tex es con el comando especial `\def`. Lo más sencillo es escribir un nuevo comando que no tenga parámetros. Para esto se escribe `\def`, luego el nombre del nuevo comando y luego un grupo con la definición del comando. El nuevo comando sirve como una abreviación del contenido del grupo.

Un ejemplo clásico es definir un comando para el conjunto de los números reales. Para generar \mathbb{R} hay que escribir `\mathbb{R}`, lo cual requiere nueve caracteres además de la contrabarra. Podemos crear un comando `\R` que de el mismo resultado mediante:

```
\def\R{\mathbb{R}}.
```

Desde ese punto en adelante, podemos usar el nuevo comando `\R` con el mismo resultado que `\mathbb{R}`.

También se puede crear comandos con parámetros. Aquí aparece el último símbolo reservado de tex que nos queda por ver: `#`. Entre el nombre del comando nuevo y el grupo que lo define se indica la cantidad de parámetros. Si no ponemos nada entre medio, como en el caso de `\R`, no hay parámetros. Para poner un parámetro, en ese lugar se escribe `#1`; para poner dos se escribe `#1#2` y así sucesivamente. Es importante no poner ningún espacio ni nada más entre medio (tex lo interpreta de cierta forma especial; ver el capítulo sobre macros del Texbook [2]). En resumen:

```
\def\comSinParam{...} \def\comUnParam#1{...} \def\comDosParam#1#2{...}
```

Dentro del grupo que define al comando, podemos usar `#1` para el primer parámetro, `#2` para el segundo y así sucesivamente (siempre y cuando estén dentro de los parámetros que se indican para el comando). Veamos algunos ejemplos.

Supongamos que queremos un comando `\vecg` con dos parámetros de modo que por ejemplo `\vecg xn` de (x_1, \dots, x_n) y `\vecg ym` de (y_1, \dots, y_m) . Lo hacemos de la siguiente forma.

```
\def\vecg#1#2{(#1_1,\ldots,#1_{#2})}
```

Si queremos que algún parámetro tenga más de un símbolo podemos usar grupos, por ejemplo `\vecg x{n+1}` da (x_1, \dots, x_{n+1}) . El grupo al rededor del `#2` en la definición del comando es para que funcione bien con subíndices de más de un símbolo. Sin eso, el ejemplo anterior daría $(x_1, \dots, x_n + 1)$.

Podemos hacer también un comando de tres parámetros para matrices genéricas.

```
\def\matg#1#2#3
{
  \begin{pmatrix}
    #1_{11}&#1_{12}&\ldots&#1_{1#3}\\
    #1_{21}&#1_{22}&\ldots&#1_{2#3}\\
    \vdots&\vdots&\ddots&\vdots\\
    #1_{#21}&#1_{#22}&\ldots&#1_{#2#3}
  \end{pmatrix}
}
```

Con `\matg xmn` y `\matg ykt` obtenemos lo siguiente.

$$\begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix} \quad \begin{pmatrix} y_{11} & y_{12} & \dots & y_{1t} \\ y_{21} & y_{22} & \dots & y_{2t} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k1} & y_{k2} & \dots & y_{kt} \end{pmatrix}$$

Habiendo visto algunos ejemplos podemos analizar un poco más en qué casos es útil definir comandos y qué ventajas tiene.

Respecto a cuándo definir comandos, depende del documento en cuestión. En un material de álgebra lineal, el comando anterior probablemente sería útil. Por otra parte, en un documento en el que aparece una sola matriz, no valdría la pena definirlo solo para

ese caso. En general, tiene sentido definir comandos para cosas que se usan varias veces. Al escribir algo largo, es una buena práctica analizar qué cosas se usarán muchas veces para hacerles comandos.

Hay varias ventajas de definir comandos para cosas que aparecerán varias veces. Una es que agiliza la escritura del código. Otra es que sirve para prevenir errores de tipeo, al ahorrarnos tareas tediosas. Por ejemplo, supongamos que debemos escribir quince matrices genéricas. Si no usamos un comando es más fácil ingresar algo mal en alguna.

Otra ventaja es la uniformización del documento. Siguiendo con el ejemplo de las matrices genéricas, si usamos el comando, todas tendrán la misma estructura de mostrar cuatro elementos arriba a la izquierda, dos arriba a la derecha, dos abajo a la izquierda y uno abajo a la derecha. Si cada vez ingresamos la matriz con el entorno, capaz alguna vez cambiamos un poco la estructura, como en el siguiente ejemplo.

$$\begin{pmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mn} \end{pmatrix}$$

En general, conviene no hacer cambios de notación con cosas que tienen el mismo sentido.

Una ventaja más, relacionada con la anterior, es la posibilidad de cambiar una notación en todo el documento modificando una sola cosa. Por ejemplo, si usamos matrices genéricas en todo el documento con la estructura del comando dado pero queremos pasar a que sean como la del párrafo anterior, alcanza con modificar el comando al siguiente.

```
\def\matg#1#2#3
{
  \begin{pmatrix}
    #1_{11}&\ldots&#1_{1#3}\\
    \vdots&\ddots&\vdots\\
    #1_{#21}&\ldots&#1_{#2#3}
  \end{pmatrix}
}
```

Lo anterior es particularmente recomendable cuando manipulamos un concepto para el podrían usarse distintas notaciones. Supongamos que estamos inventando la *cotransformada hiperbólica cuántica* de una función. Al principio, dada una función f usamos la notación $\mathcal{H}(f)$, por lo que definimos el comando `\def\cothc#1{\mathcal{H}(#1)}`. Sin embargo, más adelante optamos por cambiar a $f^{\mathcal{H}}$. Lo único que debemos hacer es reemplazar la definición del comando por `\def\cothc#1{#1^{\mathcal{H}}}`.

Si definimos un comando con el nombre de otro anterior, es lo sobrescribe. Por otra parte, si definimos un comando dentro de un grupo, la definición solamente tiene efecto dentro de ese grupo. Si el comando tenía otra definición previa, fuera del grupo vuelve a esa definición. Por lo tanto, si se quiere definir un comando concreto para algo específico, se lo puede hacer dentro de un grupo, sin preocuparse por que afecte al resto del documento.

Latex provee de otro comando para definir comandos, el cual es `\newcommand`. En este caso la cantidad de parámetros se indica con un número entre corchetes rectos. Por ejemplo el vector genérico sería `\newcommand\vecg[2]{(#1_1,\ldots,#1_{#2})}`. Otra diferencia es que si ya hay un comando con ese nombre, da un error en lugar de sobrescribirlo (tiene sentido, ya que dice *new command*).

Cuando vimos el comando `\atop`, se mencionó que en esta sección veríamos una forma de hacer un comando que hace lo mismo pero a partir de `\genfrac` de `amsmath`. La situación es que este segundo comando tiene varios parámetros, para permitir cambiar muchas cosas. Lo que hacemos aquí es dejar varios parámetros vacíos y al que indica el grosor de la línea, le ponemos `0pt`. La definición del comando es la siguiente.

```
\newcommand\amsatop[2]{\genfrac{}{}{0pt}{}{#1}{#2}}
```

La forma de uso de este comando es como la de `\frac`, con un parámetro para lo de arriba y otro para lo de abajo. En ese sentido se diferencia de `\atop`. Lo probamos para ver si funciona bien.

$$x \overset{y}{\underset{z}{\sum_{\substack{1 \leq i \leq p \\ 1 \leq j \leq q \\ 1 \leq k \leq r}}} a_{ij} b_{jk} c_{ki}}$$

Parece que efectivamente funciona bien. Si se quisiera se podría usar `\def` y ponerle como nombre `\atop`, remplazando al otro (cosa que con `\newcommand` no se puede).

Cabe mencionar que con `\def` se pueden definir comandos con comportamientos mucho más variados. Por ejemplo, se puede hacer que los parámetros se pasen de muchas formas distintas, o incluir comandos `\if` para que el comportamiento varíe según condiciones. No se suele usar, salvo que se esté trabajando en un paquete, pero es bueno saber que es posible para no sorprenderse al encontrar comandos con funcionamiento complejo o distinto al usual. Para quienes les interese, ver el capítulo sobre macros del *TeXbook* [2].

5.9. Funcionalidades de latex

En esta sección veremos algunas funcionales de latex muy útiles que no están en `tex` ni `plain tex`. Se trata del estructuramiento del documento en secciones, el agregado de imágenes y las referencias, tanto internas como a bibliografía.

5.9.1. Secciones

Latex tiene implementado un sistema de secciones que se suele usar para estructurar el documento. Se trata de los comandos `\section`, `\subsection` y `\subsubsection`, cada uno de los cuales recibe un parámetro que es el título. Dependiendo del tipo de documento (primera línea del archivo), puede usarse también el comando `\chapter` para indicar capítulos. En documentos de tipo `report` o `book` se ponen capítulos, mientras que en los de tipo `article` no.

Se puede crear un índice automáticamente con el comando `\tableofcontents`. A veces puede hacer falta compilar dos veces para que quede bien, sobre todo si se hace con `pdflatex` desde la terminal.

Se puede hacer que una sección (subsección, etc.) no se enumere ni se incluya en el índice poniendo un asterisco entre el comando y el título, es decir `\section*{}`. Estas secciones, además de no enumerarse, no afectan la numeración de las otras. Es bastante estándar en latex que un asterisco tenga el sentido “no enumerar”.

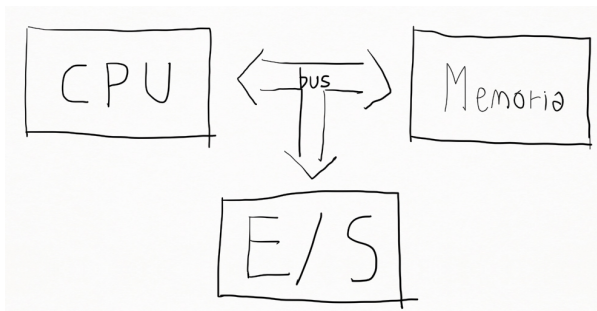
5.9.2. Imágenes


Las funcionalidades de incluir imágenes vienen de latex y paquetes. Cuando se creó tex (tiempo en el que no existía el formato pdf y tal vez lo más común era imprimir) no se contempló el uso de imágenes. El paquete que vamos a usar es `graphicx`, cuya documentación está en ctan.org/pkg/graphicx (por algún motivo está junta la documentación del paquete `color`, para cambiar colores).

El comando para incluir una figura en un documento es `\includegraphics{archivo}`, donde *archivo* es el nombre completo del archivo (por ejemplo “diagrama.png”). Este comando viene del paquete `graphicx`. Se pueden poner parámetros opcionales entre corchetes rectos. Algunos son `scale` para escalar la imagen y `width` para fijar el ancho. En general puede ser útil usar `width` en relación al ancho del texto, el cual es `\textwidth` (o `\hsize` de TeX). Si queremos que el ancho de la imagen sea por ejemplo la mitad del del texto, podemos usar `\includegraphics[width=.5\textwidth]{archivo}`.

Las imágenes se buscan en la carpeta donde está el `.tex` (tal vez en el fondo por ser la carpeta donde se ejecuta el compilador). Se puede entrar a otra carpeta con su nombre y una barra. Por ejemplo, para este documento tenemos las imágenes en una carpeta llamada `fig`, por lo que escribimos `fig/nombre.png` en los `includegraphics`. En general, conviene tener una carpeta para las imágenes, de modo que el proyecto quede ordenado mejor.

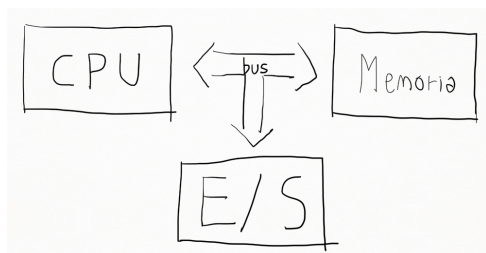
El resultado de `\includegraphics` es una caja con la imagen, la cual se puede usar como parte de otras cosas. Por ejemplo, podemos poner una imagen dentro de un modo display para que la muestre centrada y con márgenes arriba y abajo.



Tampoco nada nos impide poner una imagen con el texto, por ejemplo , pero en general no es de mucha utilidad. Parece que la imagen se comporta como contenido de modo horizontal, así como una letra. De hecho, si fuera de un párrafo ponemos una imagen al comienzo de una línea, se toma como que comienza un nuevo párrafo y se le pone la sangría correspondiente antes.



este párrafo es comenzado por la imagen.



Como otro ejemplo un poco más técnico, se puede poner un párrafo al lado de una imagen usando cajas. Por otra parte, lograr que se alineen por arriba fue MUY técnico.

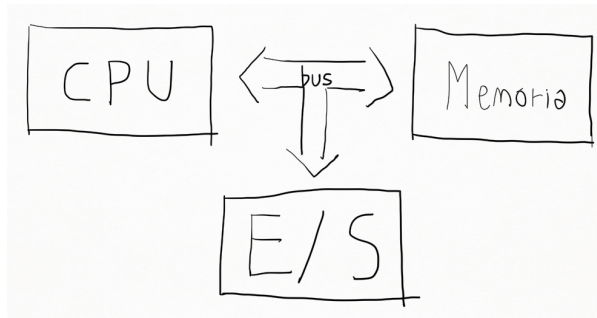


Figura 5.1: Descripción de la figura.

En resumen, con `\includegraphics` se puede insertar una imagen en cualquier parte del documento, con parámetros opcionales que permiten por ejemplo cambiar el tamaño. Sin embargo, lo más usual es usar el entorno `figure` de latex. Haciendo esto, latex se encarga de elegir en dónde poner la imagen de modo que quede bien con el resto del documento. Además, tenemos funcionalidades para ponerle una descripción por debajo y para referenciar la imagen desde el texto.

El modo más usual de usar este entorno es el siguiente

```
\begin{figure}
  \centering
  \includegraphics[width=.5\hsize]{fig/ArqVN.png}
  \caption{Descripción de la figura.}
  \label{fig-ArquiVonNeu}
\end{figure}
```

El primer comando indica que el contenido se centra. El segundo es para insertar la imagen. El tercero es para generar una descripción y el cuarto es para poder referenciarlo desde el texto (el label debe ir luego del caption). Ponemos el entorno dentro del código.

La posición de la figura no tiene por qué ser la misma que la del código. En general se suele preferir que queden en la parte de arriba o de abajo de las hojas. Para hablar sobre la imagen desde el texto, se referencia por el número con frases como “en la figura 5.1 hay tres cajas”. No hace falta saber el número exacto de la imagen, sino que se usa el `label` definido en el entorno. Al escribir `\ref{fig-ArquiVonNeu}`, automáticamente se coloca el número de la figura con un enlace que si se hace click lleva a ese lugar. Si luego ponemos otra figura antes y el número cambia, las referencias se acomodan solas, lo cual puede ser muy práctico.

Notar que esto de que la imagen se referencie en el texto por su número, sin necesariamente estar en ese mismo lugar, es lo más común en textos científicos y técnicos.

Cabe aclarar que en el entorno `figure` se puede poner cualquier cosa y de hecho no tiene por qué haber una imagen. Observar la figura 5.2. En particular, si se quiere se puede poner más de una imagen, como en la figura 5.3.

aaaaa

Figura 5.2: Esto también es una figura.

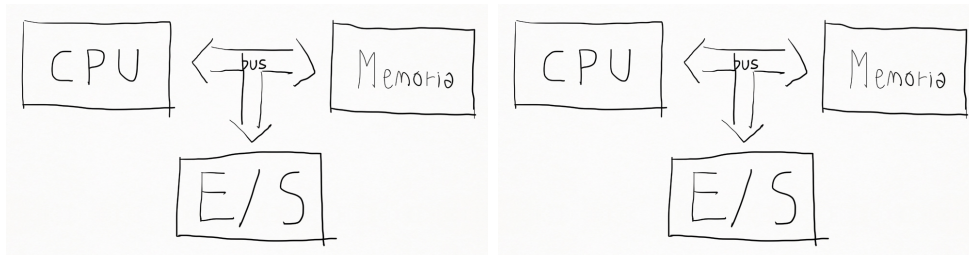


Figura 5.3: Una figura con dos imágenes. Si fueran distintas, capaz convendría fijar la altura en los parámetros opcionales con `height=x`. El salto de línea y la tabulación entre una imagen y la otra se interpretan como un espacio, por lo que están un poco separadas.

5.9.3. Referencias

Latex provee de funcionalidades muy prácticas para referenciar cosas dentro del documento. Se trata de los comandos `\label` y `\ref`, que ya vimos aplicados a referenciar figuras en la sección 5.9.2. Estos comandos también se pueden usar para referenciar secciones y ecuaciones, entre otras cosas. La ventaja que tiene esto es que si agregamos cosas antes y la enumeración se modifica, las referencias se ajustan automáticamente. El funcionamiento siempre es que con `\label` definimos una etiqueta asociada a algo, la cual conviene que tenga alguna lógica, y luego podemos referenciar esa cosa con `\ref` pasando la etiqueta como parámetro. Esto funciona tanto si el `\ref` está después de la cosa como si está antes.

Para referenciar una sección (o capítulo, o subsección, etc.) se pone un `\label` con alguna etiqueta para la sección y luego usando `\ref` con esa etiqueta queda la referencia. La etiqueta puede estar después de la definición de la sección. Por ejemplo, al lado de la definición de la subsección, escribimos `\label{sec-referencias}`, lo cual parece una elección clara de etiqueta. Luego podemos referirnos a la sección con `\ref{sec-referencias}`, en cualquier parte del documento, ya sea antes o después. Por ejemplo, podemos afirmar que estamos en la subsección 5.9.3.

También se pueden poner etiquetas a ecuaciones. Para eso, hay que usar el entorno `equation`, el cual activa el modo matemático display (al igual que `$$`) y además genera un número para la ecuación, que se muestra a la derecha. Dentro de ese entorno podemos poner un `\label` y luego referenciar con `\ref`. Por ejemplo, creamos la ecuación

$$E = mc^2 \tag{5.1}$$

con la etiqueta `\label{ec-Eins}` y luego la referenciamos con `\ref{ec-Eins}`, diciendo por ejemplo que la ecuación 5.1 se debe a Einstein. Si luego agregáramos otra ecuación antes y el número de esta pasara a ser (5.2), la referencia se ajustaría sola. Por otra parte, con el entorno `align` (sin asterisco), se asigna un número a cada fila y para referenciarlo hay que poner un label en la misma fila. Por ejemplo, aquí adelante tenemos las ecuaciones 5.2 y 5.3 (notar que las referencias pueden ir antes en el documento).

$$x + y = 2 \tag{5.2}$$

$$x - y = 0 \tag{5.3}$$

Otros elementos de latex que se pueden referenciar son entornos de teorema, lema, definición, ejercicio, etc. y tablas. En este documento tenemos entornos de ejercicio. Po-

demostramos por ejemplo referenciar el ejercicio 2.6.1. No entramos en detalle sobre los entornos tipo teorema, pero sí cabe mencionar que el entorno de ejercicios usado está definido en el preámbulo del documento con las líneas:

```
\theoremstyle{definition}
\newtheorem{ejercicio}{Ejercicio}[section]
```

donde la primera hace que el estilo sea como de definición (esto afecta la fuente que se usa) y la segunda define el nombre del entorno junto con cosas respecto a la numeración. Ver la sección “Lemas, teoremas, corolarios, ...” de la introducción no-tan-corta a latex [4].

Respecto a las tablas, en este documento no aparece ninguna y tampoco entraremos en detalle. Ver la sección 2.11.6 “Tablas” de [4], en la que se habla del entorno `tabular`.

Vemos ahora cómo hacer una bibliografía (referencias a otros documentos). Hay varias formas. La más básica es con el entorno `thebibliography`. Ver la sección 4.2 “Bibliografía” de [4]. Por otra parte, está la herramienta `bibtex`, que permite poner los datos de la bibliografía en otro archivo, de formato `.bib`. Finalmente, está `biblatex`: una versión más actual de `bibtex` con funcionamiento similar pero varias mejoras. Para la bibliografía de este documento usamos esta última opción. Aquí veremos un poco como funciona, ejemplificando con este mismo documento, pero está explicado más detalladamente en el capítulo “Bibliographies” de [5].

En el archivo `.bib` se escriben entradas para las distintos documentos que queremos referenciar. Para este documento usamos el archivo `biblio.bib`. Algunas de sus entradas son las siguientes dos.

```
@book{texbook,
  author = {Donald E. Knuth},
  year = {1986},
  title = {The {\TeX} Book},
  publisher = {Addison-Wesley Professional}
}
@book{Oetiker2025,
  author = {Oetiker, Tobias and ... and Schlegl, Elisabeth},
  title = {The Not So Short Introduction to LaTeX},
  year = {2025},
  url = {https://tobi.oetiker.ch/lshort/lshort.pdf},
  note = {Version 7.0}
}
```

Cada entrada comienza con un arroba y una palabra que indica el tipo de documento. Como ambos son libros, las entradas comienzan con `@book`. Si hubiera también un artículo, empezaría con `@article`. Luego entre corchetes hay un identificador (que podemos elegir libremente), con el que referenciamos al documento en cuestión, seguido de distintos datos. Dentro de estos datos hay campos obligatorios y opcionales, que dependen del tipo de documento. Por ejemplo para libros los obligatorios son el autor, el título y la fecha.

Para usar `biblatex`, en el preámbulo se incluye el paquete `biblatex` y se ejecuta el comando `\addbibresource` para indicar el archivo `.bib` a usar. En nuestro caso son las siguientes dos líneas.

```
\usepackage{biblatex}
\addbibresource{biblio.bib}
```

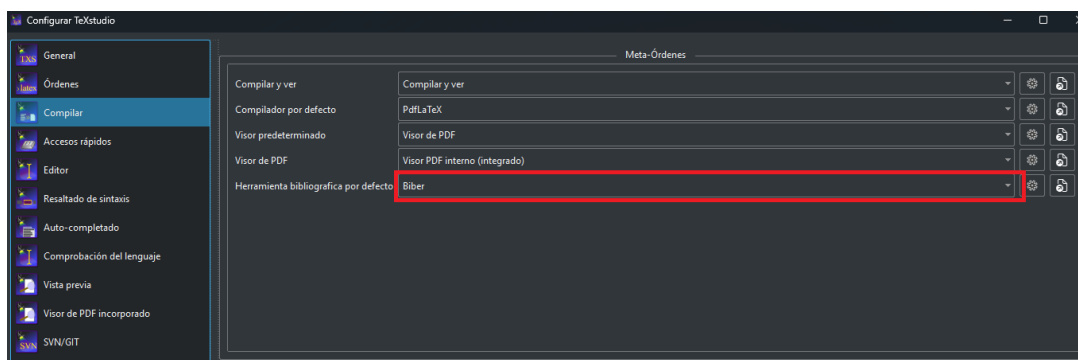


Figura 5.4: Configuración de biiber como herramienta por defecto.

Después de eso, en el documento se puede hacer referencias bibliográficas con el comando `\cite`, el cual recibe el identificador como parámetro. Para que funcione bien, el identificador que ponemos debe coincidir con el de una de las entradas en el archivo `.bib`. Por ejemplo, con `\cite{texbook}` podemos citar al Texbook [2].

Finalmente, hay que indicar en qué parte del documento queremos que quede la bibliografía. Eso se hace con el comando `\printbibliography`, el cual ponemos antes del `\end{document}` para que quede al final, como es usual.

Un detalle importante para el uso de `biblatex`. Para armar la bibliografía se usa un programa llamado `biber`, el cual viene dentro de MikTeX. Para que funcione bien, es importante que se use ese programa en lugar del de `bibtex`. Al usar un entorno de desarrollo, como TeXstudio, puede haber que elegir se use `biber` en la configuración (menú “Opciones” y “Configurar TeXstudio”). Ver la figura 5.4.

Cabe mencionar que si hay bibliografía es más complejo compilar desde la terminal. Hay que ejecutar una vez el compilador de latex, luego una vez `biber` y finalmente dos veces más el compilador. Para este archivo el orden de instrucciones sería: `pdflatex notasComp.tex`, `biber notasComp`, `pdflatex notasComp.tex`, `pdflatex notasComp.tex`.

5.10. Cosas que no tratamos y comentarios finales

Algunas cosas particularmente útiles que no vimos (o solo mencionamos) son los entornos de teoremas y las tablas. Ambos se tratan en [4].

Por otra parte, hay un paquete muy potente para generar gráficos: `tikz`. Se recomienda aprender a usarlo. Entendiendo los principios de tex y latex, se puede encarar leyendo documentación, viendo ejemplos y practicando. Documentación: tikz.dev/ o ctan.org/pkg/pgf.

También está la posibilidad de hacer diapositivas con latex, usando el tipo de documento `beamer`. En la sección 5.4 “Creating Presentations” de [5] hay una introducción.

Cabe resaltar que este capítulo no trata solo sobre Latex, sino sobre Tex y Latex. Algunas de las cosas que vimos se desalienta usarlas en latex, ya sea porque se considere que son más difíciles de entender, porque no sigan la filosofía de Latex de que el contenido y el estilo se traten por separado, porque no interactúen bien con otras funcionalidades nuevas, o directamente porque hayan nuevas alternativas mejores (en algún sentido). Por

ejemplo, el uso de `$$` para modo display se desalienta, recomendándose remplazarlo por `\[` y `\]` o el entorno `equation*`. Por otra parte, los cambios de fuente con los comandos de dos letras, como `\tt` y `\bf`, también se desalientan, recomendando usar en su lugar comandos como `\texttt` y `\textbf`. De todos modos, tras enterarse de tales situaciones, en general es muy sencillo cambiar por la alternativa recomendada (al menos en las situaciones donde afecta). Cabe mencionar que las consideradas “buenas prácticas” de Latex, en general buscan facilitar la escritura del documento, fomentando el uso de elementos intuitivos de alto nivel. Por otra parte, alguien perfectamente puede ponerse a trabajar con cosas técnicas (como cajas de Tex) si así lo desea (aunque se reduce bastante la cantidad de personas que entendería el código). Se espera que lo visto aquí aporte a un entendimiento más o menos sólido del funcionamiento de Tex y Latex, lo cual a lo mejor puede ser interesante de por sí y junto con práctica ayudar a llegar a un muy buen manejo.

Para aprender cosas nuevas, al igual que con conceptos de programación, se recomienda usar tanto libros y documentación como cosas que se encuentren en internet o textos generados por modelos del lenguaje. Cada cosa tiene sus ventajas. Los sitios de preguntas y respuestas como *Stack Exchange* o *Stack Overflow* pueden ser de particular utilidad.

En internet hay disponible material (en inglés) sobre un curso de escritura matemática dado por Donald Knuth (el autor de Tex), titulado *mathematical writing* [1]. Se recomienda mirarlo, pues tiene consideraciones que pueden ayudar a mejorar la calidad de escritura.

Por último, se recomienda aprender a usar `git`, que es una herramienta muy buena para gestión de versiones. Tanto el repositorio de este material como los de las bibliotecas de python vistas funcionan en base a git. Material para aprender desde cero (cómo usarlo pero también qué es): git-scm.com/learn.

Bibliografía

- [1] Tracy Larrabee Donald E. Knuth y Paul M. Roberts. *Mathematical Writing*. 1987. URL: https://jmlr.csail.mit.edu/reviewing-papers/knuth_mathematical_writing.pdf.
- [2] Donald E. Knuth. *The T_EX Book*. Addison-Wesley Professional, 1986.
- [3] Leslie Lamport. *L^AT_EX: a Document Preparation System*. 2.^a ed. Massachusetts: Addison Wesley, 1994.
- [4] Tobias Oetiker et al. *La introducción no-tan-corta a latex*. Version 5.03. 2014. URL: <https://ctan.org/pkg/lshort-spanish>.
- [5] Tobias Oetiker et al. *The Not So Short Introduction to LaTeX*. Version 7.0. 2025. URL: <https://tobi.oetiker.ch/lshort/lshort.pdf>.