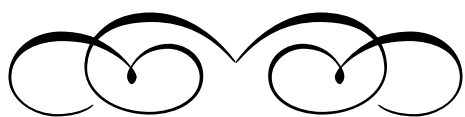


# **Matemáticas y programación *con Python***

*para ingenierías*

N. Aguilera

9 de octubre de 2019



# Contenidos

<b>1. Preliminares</b>	<b>1</b>
1.1. ¿Qué es esto? . . . . .	1
1.2. Organización y convenciones que usamos . . . . .	2
1.3. ¿Por qué Python? . . . . .	3
1.4. Censura, censura, censura . . . . .	5
1.5. De la paciencia, o cómo estudiar . . . . .	6
1.6. Comentarios . . . . .	7
1.7. Agradecimientos . . . . .	8

## Parte I : Elementos **9**

<b>2. El primer contacto</b>	<b>11</b>
2.1. Funcionamiento de la computadora . . . . .	11
2.2. Bits y bytes . . . . .	13
2.3. Programas y lenguajes de programación . . . . .	14
2.4. Python y IDLE . . . . .	16
<b>3. Python como calculadora</b>	<b>18</b>
3.1. Operaciones con números . . . . .	18
3.2. El módulo <i>math</i> . . . . .	24
3.3. Comentarios . . . . .	28

<b>4. Tipos de datos básicos</b>	<b>29</b>
4.1. ¿Por qué hay distintos tipos de datos? . . . . .	29
4.2. Tipo lógico . . . . .	30
4.3. Cadenas de caracteres . . . . .	34
4.4. print (imprimir) . . . . .	36
4.5. Comentarios . . . . .	38
<b>5. Variables y asignaciones</b>	<b>39</b>
5.1. Asignaciones en Python . . . . .	39
5.2. None . . . . .	45
5.3. Comentarios . . . . .	47
<b>6. Módulos</b>	<b>48</b>
6.1. Módulos propios . . . . .	49
6.2. Ingreso interactivo . . . . .	51
6.3. Documentación y comentarios en el código . . . . .	53
6.4. Usando import . . . . .	54
<b>7. Funciones</b>	<b>58</b>
7.1. Ejemplos simples . . . . .	59
7.2. Funciones numéricas . . . . .	63
7.3. Variables globales y locales . . . . .	64
7.4. Comentarios . . . . .	71
<b>8. Tomando control</b>	<b>72</b>
8.1. if (si) . . . . .	72
8.2. while (mientras) . . . . .	77
8.3. El algoritmo de Euclides . . . . .	83
<b>9. Sucesiones</b>	<b>88</b>
9.1. Índices y secciones . . . . .	89
9.2. tuple (tupla) . . . . .	90
9.3. list (lista) . . . . .	93
9.4. range (rango) . . . . .	100

9.5. Operaciones comunes . . . . .	101
9.6. Comentarios . . . . .	103
<b>10. Recorriendo sucesiones</b>	<b>104</b>
10.1. for (para) . . . . .	104
10.2. Un alto en el camino . . . . .	116
10.3. Listas por comprensión . . . . .	118
10.4. Filtros . . . . .	121
10.5. Comentarios . . . . .	123
<b>11. Formatos y archivos de texto</b>	<b>124</b>
11.1. Formatos . . . . .	124
11.2. Archivos de texto . . . . .	128
11.3. Comunicación con otras aplicaciones . . . . .	136
11.4. Comentarios . . . . .	137
<b>12. Clasificación y búsqueda</b>	<b>138</b>
12.1. El módulo <i>random</i> . . . . .	139
12.2. Clasificación . . . . .	140
12.3. Eliminando elementos repetidos de una lista . . . . .	145
12.4. Búsqueda binaria . . . . .	150
12.5. Comentarios . . . . .	153

## **Parte II : *Popurrí*** **155**

<b>13. El largo y serpenteante recorrido</b>	<b>157</b>
13.1. Números de Fibonacci . . . . .	157
13.2. Sucesiones que se entrelazan . . . . .	160
13.2.1. Álgebra Lineal . . . . .	161
13.2.2. Relojeando . . . . .	162
13.2.3. Plazos fijos y préstamos . . . . .	163
13.3. Polinomios . . . . .	167
13.4. Ecuaciones diofánticas y la técnica de barrido . . . . .	170

13.5.	Cribas . . . . .	175
13.6.	El problema de Flavio Josefo . . . . .	177
13.7.	Ejercicios adicionales . . . . .	179
13.8.	Comentarios . . . . .	182
<b>14.</b>	<b>Recursión</b>	<b>184</b>
14.1.	Introducción . . . . .	184
14.2.	Funciones definidas recursivamente . . . . .	185
14.3.	Ventajas y desventajas de la recursión . . . . .	187
14.4.	Los Grandes Clásicos de la Recursión . . . . .	190
14.5.	Ejercicios adicionales . . . . .	195
14.6.	Comentarios . . . . .	195
<b>15.</b>	<b>Grafos</b>	<b>196</b>
15.1.	Ensalada de definiciones . . . . .	196
15.2.	Representación de grafos . . . . .	200
15.3.	Recorriendo un grafo . . . . .	206
15.4.	Ejercicios adicionales . . . . .	215
15.5.	Comentarios . . . . .	216
<b>16.</b>	<b>Cálculo numérico elemental</b>	<b>217</b>
16.1.	La codificación de decimales . . . . .	217
16.2.	Errores numéricos . . . . .	225
16.3.	Métodos iterativos: puntos fijos . . . . .	230
16.4.	El método de Newton y Raphson . . . . .	236
16.5.	El método de la bisección . . . . .	242
16.6.	Comentarios . . . . .	248
<b>17.</b>	<b>Objetos combinatorios</b>	<b>251</b>
17.1.	Contando objetos combinatorios . . . . .	251
17.2.	Las grandes listas . . . . .	254
17.3.	yield . . . . .	256
17.4.	Generando objetos combinatorios . . . . .	260
17.5.	Ejercicios adicionales . . . . .	264

17.6. Comentarios . . . . .	265
-----------------------------	-----

## **Parte III : Apéndices** **267**

### **Apéndice A. Módulos y archivos mencionados** **269**

A.1. Módulos Python . . . . .	269
<i>dearchivoatterminal</i> . . . . .	269
<i>decimales</i> . . . . .	270
<i>eratostenes</i> . . . . .	272
<i>euclides2</i> . . . . .	272
<i>fargumento</i> . . . . .	274
<i>flocal</i> . . . . .	275
<i>grafos</i> . . . . .	275
<i>holamundo</i> . . . . .	277
<i>holapepe</i> . . . . .	277
<i>ifwhile</i> . . . . .	277
<i>numerico</i> . . . . .	279
<i>recursion1</i> . . . . .	282
<i>recursion2</i> . . . . .	284
<i>sumardos</i> . . . . .	285
<i>tablaseno</i> . . . . .	286
A.2. Archivos de texto . . . . .	286
<i>santosvega.txt</i> . . . . .	286
<i>unilang8.txt</i> . . . . .	286

### **Apéndice B. Notaciones y símbolos** **288**

B.1. Lógica . . . . .	288
B.2. Conjuntos . . . . .	289
B.3. Números: conjuntos, relaciones, funciones . . . . .	289
B.4. Números importantes en programación . . . . .	291
B.5. En los apuntes . . . . .	291
B.6. Generales . . . . .	292



<b>Parte IV : Índices</b>	<b>293</b>
<b>Comandos de Python que usamos</b>	<b>295</b>
<b>Índice de figuras y cuadros</b>	<b>298</b>
<b>Bibliografía</b>	<b>300</b>
<b>Índice alfabético</b>	<b>302</b>

# Capítulo 1

## Preliminares

### 1.1. ¿Qué es esto?

Estos son apuntes de un curso introductorio a la resolución de problemas matemáticos usando la computadora que se dicta en el [Departamento de Matemática](#) de la Facultad de Ingeniería Química de la Universidad Nacional del Litoral (Argentina), y disponibles gratuitamente en lo que cariñosamente llamamos [página del «libro»](#).<sup>(1)</sup>

La primera parte, *Elementos*, es el núcleo del curso y la mayoría de sus contenidos son comunes a otros cursos iniciales de programación. Comenzamos con tipos de datos, funciones y estructuras de control, pasamos luego al manejo de secuencias, vemos manejo básico de archivos de texto y terminamos con rudimentos de búsqueda y clasificación.

La segunda parte y como su nombre, *Popurrí*, lo indica, tiene una variedad de temas más directamente relacionados con matemáticas y las carreras a las que está destinado el curso: después de aplicaciones de **while** y **for**, vemos recursión, elementos de grafos, cálculo numérico y generación de objetos combinatorios.

---

<sup>(1)</sup> <http://oma.org.ar/invydoc/libro-prog.html>

Todos los enlaces de internet mencionados en estas notas eran válidos en febrero de 2018.

A diferencia de los cursos tradicionales de programación, vemos muy poco de aplicaciones informáticas como bases de datos, interfaces gráficas o internet: el énfasis es en matemáticas y algoritmos, intentando ir más allá de «apretar botones».

Por otro lado, tampoco vemos temas de complejidad o corrección de programas. En ese sentido, no se presentan los algoritmos más rápidos, tratando de hacer un equilibrio entre sencillez y eficiencia.

En fin, vemos muy poca teoría, que se habrá visto o se verá en otros cursos: lo esencial aquí son los ejercicios.

Con la convicción de que usar *seudocódigo* en un curso introductorio es como enseñar geometría sin axiomas (uno nunca sabe qué está permitido y qué no), usamos un lenguaje concreto de programación, en este caso Python, tratando de no apartarnos demasiado de las estructuras disponibles en otros lenguajes.

## 1.2. Organización y convenciones que usamos

En los distintos capítulos se presentan los temas y ejercicios, agrupados en secciones y a veces subsecciones. Secciones y ejercicios están numerados comenzando con 1 en cada capítulo, de modo que la «sección 3.2» se refiere a la sección 2 del capítulo 3, y el «ejercicio 4.5» se refiere al ejercicio 5 del capítulo 4.

La [versión electrónica](#) de estos apuntes ofrece la ventaja de vínculos (*links*) remarcados en azul. Está diseñada para que pueda leerse en «tabletas», mientras que con una pantalla más grande es posible ocupar una mitad con el apunte y otra mitad con Python.

Lo que escribiremos en la computadora y sus respuestas se indican con **otro tipo de letra y color**, y fragmentos más extensos se ponen en párrafos con una raya vertical a la izquierda:

| como éste

A veces incluimos los símbolos `>>>` para destacar la diferencia entre lo que escribimos (en el renglón aparecen estos símbolos) y lo que responde la computadora (renglón sin símbolos).

Muchas veces usaremos indistintamente la notación de Python como **12** o **f(x)**, y la notación matemática como 12 o  $f(x)$ , esperando que no de lugar a confusión.

Siguiendo la tradición norteamericana, la computadora expresa los números poniendo un «punto» decimal en vez de la «coma», y para no confundirnos seguimos esa práctica. Así, 1.589 es un número entre 1 y 2, mientras que 1589 es un número entero, mayor que mil. A veces en el texto dejamos pequeños espacios entre las cifras para leer mejor los números, como en 123 456.789.

En el [apéndice A](#) están varios módulos o funciones que son propios de estos apuntes y no están incluidos en la distribución de Python. Todos los módulos no estándar que usamos están en la [página del «libro»](#).

En el [apéndice B](#) hay una síntesis de notaciones, convenciones o abreviaturas. En la [sección B.5](#) se explican los «garabatos» como  $\oint$ ,  $\nabla$ ,... ¡y varios más!

Al final se incluyen índices que tal vez no sean necesarios en la versión electrónica, pero esperamos que sean útiles en la impresa.

Terminamos destacando una convención muy importante.

Un párrafo con el símbolo  $\bullet$  contiene construcciones de Python que, aunque tal vez válidas, hay que evitar a toda costa. Son construcciones confusas, o absurdas en matemáticas, o extremadamente ineficientes para la computadora, o no se usan en otros lenguajes, etc.:

*Las construcciones señaladas con  $\bullet$  no deben usarse en este curso.*

## 1.3. ¿Por qué Python?

Por muchos años usamos Pascal como lenguaje para los apuntes. Pascal fue ideado por N. Wirth hacia 1970 para la enseñanza de la programación y fue un lenguaje popular por varias décadas, pero ha caído

en desuso y es difícil conseguir versiones recientes para las distintas plataformas y que respeten el estándar.

Para reemplazar a Pascal hemos elegido a Python (pronunciado *páizon*), creado por G. van Rossum hacia 1990, por varios motivos:

- Lenguajes como Java o C quizás son más populares,<sup>(2)</sup> pero son más «duros» como primer lenguaje. En cambio, es fácil hacer los primeros programas con Python, y se puede comenzar con un modo interactivo escribiendo en una ventana tipo terminal, como *Mathematica*, *Matlab* o *Maple*.
- Es posible recorrer una lista usando índices y también es posible usar filtros, como con, por ejemplo, *Mathematica*.
- Las funciones pueden ser argumentos de otras funciones.
- No tiene límite para el tamaño de enteros.
- Es gratis, está disponible para las principales plataformas (Linux, MS-Windows, Mac OS y otras), y las nuevas versiones son lanzadas simultáneamente.
- Tiene un entorno integrado para el desarrollo (IDLE).
- La distribución oficial incluye una amplia variedad de extensiones (módulos), entre los que nos interesan *math* y *random*, relacionados con matemáticas.
- No es necesario un estándar: hay un único desarrollador.

Pero también tiene sus desventajas para la enseñanza:

- No tiene un conjunto reducido de instrucciones y es posible hacer una misma cosa de varias formas distintas, lo que confunde y distrae.
- La sintaxis no es consistente. Por ejemplo:
  - **print** es una función, pero **return** no lo es,
  - **sort** y **reverse** modifican una lista pero **sorted** da una nueva lista y **reversed** da un iterador.

---

<sup>(2)</sup> La popularidad es muy cambiante. Ver por ejemplo <http://www.tiobe.com/index.php/content/paperinfo/tpci/>.

- El resultado de `int(str(1))` es 1 como uno esperaría, pero `bool(str(False))` da `True`.
- Expresiones que no tienen sentido en matemáticas son válidas en Python, como `3 + False < 5 and True`.
  - ✍ Admitiremos algunas expresiones comunes a otros lenguajes de programación como los «cortocircuitos» lógicos.

Esto hace que, a diferencia del curso con Pascal, dediquemos una buena parte del tiempo a estudiar el lenguaje, tratando de entender su idiosincrasia.

Aunque será inevitable mencionar algunas particularidades de Python, es conveniente tener siempre presente que:

*Este es un curso de resolución de problemas matemáticos con la computadora y no de Python.*

## 1.4. Censura, censura, censura

Para organizarnos mejor y no enloquecernos vamos a poner algunas restricciones para el uso de Python:

- De los más de 300 módulos que trae la distribución, sólo permitiremos el uso explícito de *math* y *random*.
  - ✍ Varios módulos se usan implícitamente al iniciar IDLE y al trabajar con gráficos.
- No estudiaremos diccionarios (`dict`) o conjuntos (`set`).
- No veremos cómo definir clases, ni programación orientada a objetos, ni construcciones asociadas como «decoradores».
  - ✍ Usaremos algunos métodos ya existentes de Python, por ejemplo para listas.

- No veremos manejo de errores o excepciones y sentencias relacionadas (`try`, `assert`, `debug`, `with`, etc.).
- No veremos cómo definir argumentos opcionales, si bien los usaremos en funciones predefinidas como `round`, `print` o `sort`.
- No veremos construcciones de programación funcional como `filter`, `map` o `zip`, aún cuando son comunes a otros lenguajes.
- Evitaremos el uso de instrucciones como `del`, `remove`, `join` o `extend` (entre otras), tratando de reducir la cantidad de instrucciones disponibles.

En la [página 295](#) hay un resumen de los comandos de Python que usaremos, señalando la primera página donde aparecen. Es posible que falten algunos, pero en general son los únicos que admitiremos.

*Si decimos que no veremos alguna instrucción o módulo de Python en el curso, está prohibido usarla en la resolución de los ejercicios.*

## 1.5. De la paciencia, o cómo estudiar

Los primeros ejercicios del «libro» parecen rutinarios, pero sirven para familiarizarse con el lenguaje.

Así, al comenzar el curso muchos estudiantes copian mecánicamente los renglones del «libro» sin leer lo que se pide, observan la respuesta de la computadora y pasan al próximo enunciado, sin detenerse a reflexionar qué ha pasado.

Esto da una falsa sensación de seguridad, que todo es fácil y rápido, y en definitiva que no hay mucho para aprender. Algunos llegan a la conclusión de que es mejor dedicar el tiempo a otra cosa.

Sin embargo, a poco de avanzar los «alfileres» se desprenden y lo que para muchos resulta ser un tsunami ya está encima.<sup>(3)</sup>

Una primera recomendación es:

☞ *Leer con cuidado los enunciados antes de ponerse a escribir maquinalmente en la computadora.*

Por ejemplo, si un enunciado es «conjeturar el resultado de las instrucciones y luego verificarlas con Python », lo sano es pensar cuál sería la respuesta de Python *antes* de escribir en la computadora.

La otra recomendación es:

☞ *No dejarse estar y tratar de estudiar en forma continua.*

Dejar pasar varios días sin trabajar en el curso es invocar al tsunami.

Pero ya es tiempo de dejar los consejos, advertencias y campaña del miedo.

## 1.6. Comentarios

- A medida que pasan los años, los contenidos del «libro» se van modificando. Hay temas que cambian, otros se eliminan, otros se agregan, y otros... ¡reaparecen!

Esta versión es mayormente un reordenamiento de versiones anteriores, se corrigieron algunos errores tipográficos, y seguramente aparecieron otros.

- Los temas y formas de abordarlos están inspirados en [Wirth \(1987\)](#) y [Kernighan y Ritchie \(1991\)](#) en lo referido a programación, y en [Gentile \(1991\)](#) y [Engel \(1993\)](#) en cuanto a basar el curso en resolución de problemas (y varios de los temas considerados).
- Los primeros capítulos siguen ideas de los libros de *Mathematica* ([Wolfram, 1988](#), y siguientes ediciones), y el *tutorial de Python*.
- Hemos tratado de incluir referencias bibliográficas sobre temas y problemas particulares, pero muchos pertenecen al «folklore»

---

<sup>(3)</sup> O sea, los exámenes.



y es difícil determinar el origen.

- La mayoría de las referencias históricas están tomadas de [Mac-Tutor History of Mathematics](#)<sup>(4)</sup> y [Wikipedia](#).<sup>(5)</sup>
- Para profundizar o aprender nuevos temas de programación, sugerimos los ya mencionados libros de [Wirth](#) (en Modula-2, una variante de Pascal), y de [Kernighan y Ritchie](#) (en C).

En inglés hay muchas opciones y mencionamos sólo algunas:

- [Litvin y Litvin](#) (2010) trabajan con Python con una filosofía similar a la nuestra, pero a nivel de secundaria.
- [Sedgewick y Wayne](#) (2011) trabajan con Java.
- [Cormen y otros](#) (2009) usan *seudocódigo*.
- Y por supuesto, los clásicos de [Knuth](#) (1997a; 1997b; 1998).
- A quienes les haya «picado el bichito» les sugerimos tomar problemas o participar en las competencias estudiantiles de la ACM ([Association for Computing Machinery](#)).<sup>(6)</sup>

## 1.7. Agradecimientos

A lo largo de tantos años de existencia, son muchos los muchos docentes y alumnos que influyeron en los cambios y detectaron errores (tipográficos o conceptuales) de estas notas. Especialmente quiero agradecer a Luis Bianculli, María de los Ángeles Chara, Jorge D'Elía, María Fernanda Golobisky, Conrado Gómez, Egle Haye, Alberto Marchi, Martín Marques y Marcela Morvidone, con quienes he tenido el placer de trabajar.



---

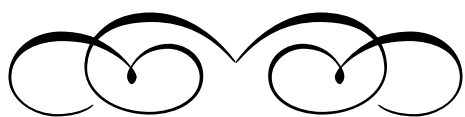
<sup>(4)</sup> <http://www-history.mcs.st-and.ac.uk/>

<sup>(5)</sup> <http://www.wikipedia.org/>

<sup>(6)</sup> <http://cm.baylor.edu/welcome.icpc>

**Parte I**

***Elementos***



## Capítulo 2

# El primer contacto

En este capítulo damos breves descripciones del funcionamiento de la computadora, de los programas y los lenguajes de programación, para terminar con nuestro primer encuentro directo con Python.

### 2.1. Un poco (muy poco) sobre cómo funciona la computadora

La computadora es una máquina que toma *datos de entrada*, los procesa y devuelve resultados, también llamados *datos de salida*, como esquematizamos en la [figura 2.1](#). Los datos, ya sean de entrada o salida, pueden ser simples como números o letras, o mucho más complicados como una matriz, una base de datos o una película.

En el modelo de computación que usaremos, el procesamiento lo realiza una *unidad central de procesamiento* o CPU (Central Processing Unit), que recibe y envía datos a un lugar llamado *memoria* de la

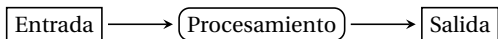


Figura 2.1: Esquema de entrada, procesamiento y salida.

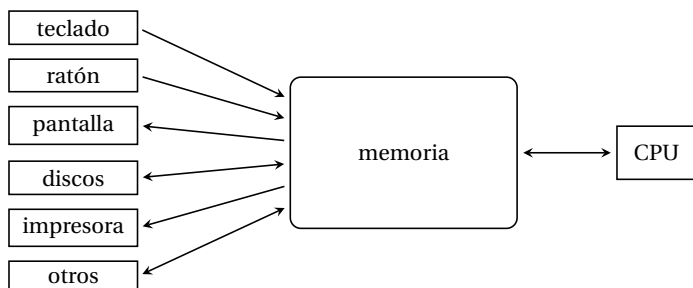


Figura 2.2: Esquema de transferencia de datos en la computadora.

computadora. Aunque las computadoras modernas suelen tener más de una CPU, en nuestro modelo consideraremos que hay una sola.<sup>(1)</sup>

Así, imaginamos que lo que escribimos en el teclado no es procesado directamente por la CPU, sino que es traducido adecuadamente y alojado en la memoria previamente. Tenemos entonces un esquema como el de la [figura 2.2](#). Los elementos a la izquierda permiten que la computadora intercambie datos con el exterior (como el teclado o la pantalla) o los conserve para uso posterior (como el disco), y la «verdadera acción» está entre la memoria y la CPU.

La CPU toma datos de la memoria, realiza alguna acción con ellos, y pone en la memoria el resultado de esta acción. Curiosamente, las instrucciones de la acción a realizar forman parte de los mismos datos.

Aunque las CPU modernas permiten el procesamiento simultáneo, nosotros pensaremos que las instrucciones se realizan *secuencialmente*, es decir, de a una y en un orden preestablecido.

Resumiendo —y muy informalmente— nuestro modelo tiene las siguientes características:

- hay una única CPU,
- que sólo intercambia datos con la memoria,
- las instrucciones que recibe la CPU forman parte de los datos

---

<sup>(1)</sup> ¡y será más que suficiente!

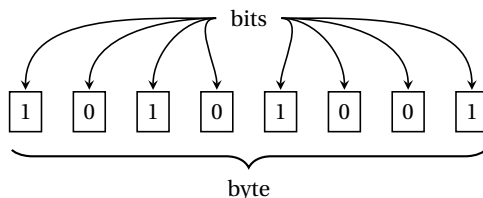


Figura 2.3: Un byte de 8 bits.

en la memoria,

- la CPU realiza las operaciones secuencialmente.

✎ El modelo, con pocos cambios, se debe a John von Neumann (1903–1957).

## 2.2. Bits y bytes

Podemos pensar que la memoria, en donde se almacenan los datos, está constituida por muchas cajitas llamadas *bits* (binary digit o dígito binario), en cada una de las cuales sólo se puede guardar un 0 o un 1. Puesto que esta caja es demasiado pequeña para guardar información más complicada que «sí/no» o «blanco/negro», los bits se agrupan en cajas un poco más grandes llamadas *bytes*, generalmente de 8 bits,<sup>(2)</sup> en los que pensamos que los bits están alineados y ordenados, puesto que queremos que 00001111 sea distinto de 11110000. Ver el esquema en la [figura 2.3](#).

**E 2.1.** Suponiendo que un byte tenga 8 bits:

- a) ¿Cuántas «ristras» distintas de 0 y 1 puede tener?

*Sugerencia:* hacer la cuenta primero para un byte de 1 bit, luego para un byte de 2 bits, luego para un byte de 3 bits,...

<sup>(2)</sup> Históricamente la cantidad de bits en un byte dependía de la computadora, pero más recientemente se estandarizó en 8 bits (por ejemplo, IEC 80000-13 de 2008).

- b) Si no importara el orden de los bits que forman el byte, y entonces 00001111, 11110000, 10100101 fueran indistinguibles entre sí, ¿cuántos elementos distintos podría contener un byte?

*Sugerencia:* si el byte tiene 8 bits puede ser que haya 8 ceros y ningún uno, o 7 ceros y 1 uno, o...



Para las computadoras más recientes, estas unidades de 8 bits resultan demasiado pequeñas para alimentar a la CPU, por lo que los bits se agrupan en cajas de, por ejemplo, 32, 64 o 128 bits (usualmente potencias de 2), siempre conceptualmente alineados y ordenados.

- ✎ La palabra *dígito* viene del latín *digitus* = dedo, y originalmente se refería a los números entre 1 y 10 (como la cantidad de dedos en las manos). El significado fue cambiando con el tiempo y actualmente según la RAE: *un número dígito es aquél que puede expresarse con un solo guarismo*. Así, en la numeración decimal los dígitos son los enteros entre cero y nueve (ambos incluidos), mientras que en base 2 los dígitos son 0 y 1.

Más recientemente, entendemos que algo es digital (como las computadoras o las cámaras fotográficas) cuando trabaja internamente con representaciones binarias, y tiene poco que ver con la cantidad de dedos en las manos.

## 2.3. Programas y lenguajes de programación

Un *programa* es un conjunto de instrucciones para que la computadora realice determinada tarea. En particular, el *sistema operativo* de la computadora es un programa que alimenta constantemente a la CPU y le indica qué hacer en cada momento. Entre otras cosas le va a indicar que *ejecute* o *corra* nuestro programa, leyendo y ejecutando las instrucciones que contiene.

Los lenguajes de programación son abstracciones que nos permiten escribir las instrucciones de un programa de modo que sean más sencillas de entender que ristra de ceros y unos. Las instrucciones para la máquina se escriben como sentencias —de acuerdo a las reglas

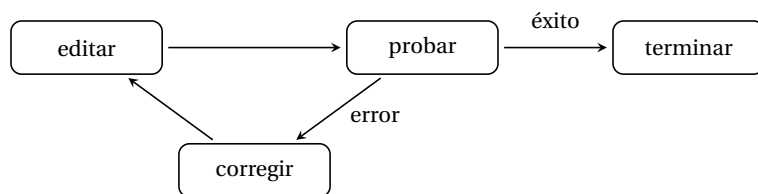


Figura 2.4: Esquema del desarrollo de un programa.

del lenguaje— que luego serán traducidas a algo que la CPU pueda entender, es decir, las famosas ristras de ceros y unos. Las sentencias que escribimos (en lenguaje humano) forman el programa *fuentes* o *código fuente* o simplemente *código*, para distinguirlo del *programa ejecutable* o *aplicación* que es el que tiene los ceros y unos que entiende la computadora.

Cuando trabajamos con el *lenguaje* Python, el programa llamado (casualmente) *python* hace la traducción de humano a binario e indica a la CPU que realice la tarea.

En la mayoría de los casos —aún para gente experimentada— habrá problemas, por ejemplo por errores de sintaxis (no seguimos las reglas del lenguaje), o porque al ejecutar el programa los resultados no son los esperados. Esto da lugar a un ciclo de trabajo esquematizado en la [figura 2.4](#): editamos, es decir, escribimos las instrucciones del programa, probamos si funciona, y si hay errores —como será la mayoría de las veces— habrá que corregirlos y volver a escribir las instrucciones.

A medida que los programas se van haciendo más largos (ponemos mayor cantidad de instrucciones) es conveniente tener un mecanismo que nos ahorre volver a escribir una y otra vez lo mismo. En todos los lenguajes de programación está la posibilidad de que el programa traductor (en nuestro caso Python) tome las instrucciones de un archivo que sea fácil de modificar. Generalmente, este archivo se escribe y modifica con la ayuda de un programa que se llama *editor de textos*.

Para los que están haciendo las primeras incursiones en progra-



mación es más sencillo tener un entorno que integre el editor de texto y la terminal. Afortunadamente, la distribución de Python incluye uno de estos entornos llamado *IDLE* (pronunciado *áidl*), y en el curso trabajaremos exclusivamente con éste. Así, salvo indicación contraria, cuando hablemos de *la terminal* nos estaremos refiriendo a *la terminal de IDLE*, y no la de Unix (aunque todo lo que hacemos con IDLE lo podemos hacer desde la terminal de Unix).

En resumen, en el curso no vamos a trabajar con Python directamente, sino a través de IDLE.

## 2.4. Python y IDLE

Usaremos la última versión «estable» de Python (3.7 al escribir estas notas), que puede obtenerse del [sitio oficial de Python](https://www.python.org/download/),<sup>(3)</sup> donde hay instalaciones disponibles para los principales sistemas operativos.

La forma de iniciar IDLE depende del sistema operativo y la instalación, pero finalmente debe aparecer la terminal de IDLE con un cartel similar al siguiente (recortado para no salirse de los márgenes):


```
Python 3.7.2 ((v3.7.2:9a3ffc0492, Dec 24 2018, 02
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license"
>>>
```

*Es importante verificar que aparezca anunciado Python 3.7 (o al menos 3.4) en alguna parte: versiones menores como 2.7 o 2.6 nos darán dolores de cabeza.*

Los signos `>>>` en la terminal indican que Python está a la espera de que ingresemos alguna orden. Por ejemplo, ponemos `2 + 2`, quedando

<sup>(3)</sup> <http://www.python.org/download/>

```
>>> 2 + 2
```

y ahora apretamos la tecla «retorno» (o «intro» o «return» o «enter» o con un dibujo parecido a , dependiendo del teclado) para obtener

```
4
>>>
```

quedando IDLE a la espera de que ingresemos nuevos comandos. En un raptó de audacia, ingresamos `2 + 3` para ver qué sucede, y seguimos de este modo ingresando operaciones como en una calculadora.

Si queremos repetir o modificar alguna entrada anterior, podemos movernos con `alt-p` (`p`revious o *p*revio) o `alt-n` (`n`ext o *s*iguiente) donde «alt» indica la tecla modificadora *alterna* marcada con «alt», o bien —dependiendo del sistema y la instalación— con `ctrl-p` y `ctrl-n`, donde «ctrl» es la tecla modificadora *control*. En todo caso se pueden mirar las preferencias en el menú de IDLE para ver qué otros atajos hay o modificarlos a gusto.

Con `ctrl-d` (fin de *d*atos) cerramos la ventana de la terminal, o bien podemos salir de IDLE usando el menú correspondiente.

*A partir de este momento, suponemos que sabemos cómo iniciar y salir de IDLE e ingresar comandos en su terminal.*



## Capítulo 3

# Python como calculadora

En este capítulo usamos Python como una calculadora sencilla, observamos que números enteros y decimales son muy diferentes para Python, y terminamos ampliando la calculadora básica a una científica mediante el módulo *math*.

### 3.1. Operaciones con números

**E 3.1.** En la terminal de IDLE ingresar **123 + 45**, y comprobar que el resultado es entero (no tiene punto decimal).

Repetir en cada uno de los siguientes, reemplazando **+** (adición) por el operador indicado y ver el tipo de resultado que se obtiene (si tiene o no coma decimal).

- a) **-** (resta, es decir, calcular **123 - 45**),
- b) **\*** (multiplicación, en matemáticas « $123 \times 45$ »),
- c) **\*\*** (exponenciación, en matemáticas « $123^{45}$  »)
- d) **/** (división),
- e) **//** (división con cociente entero),
- f) **%** (resto de la división con cociente entero, ¡no confundir con porcentaje!).

- La *guía de estilos de Python* sugiere dejar un espacio en blanco alrededor de los operadores, como en  $1 + 2$ . No es estrictamente necesario seguir esta regla, y a veces pondremos  $1+2$  sin espacios alrededor del operador. Lo que definitivamente no es estéticamente agradable es poner  $1 + 2$ , dejando un espacio de un lado pero no del otro,... ¡sobre gustos hay *tanto* escrito!

Observar que, excepto el caso de la división  $123 / 45$ , todos los resultados son enteros (no tienen coma decimal).

**E 3.2.** Repetir los apartados del ejercicio anterior considerando 12.3 y 4.5 (en vez de, respectivamente, 123 y 45), y ver si los resultados tienen o no coma decimal.

Observar que el resultado de  $12.3 // 4.5$  es 2.0 y no 2.

**E 3.3.** ¿Qué pasa si ponemos cualquiera de las siguientes?

- a)  $12 / 0$       b)  $34.5 / 0$       c)  $67 // 0$

**E 3.4.** ¿Cuánto es  $0^0$  (cero a la cero) según las matemáticas? ¿Y según Python?

**E 3.5.** Si tenemos más de una operación, podemos agrupar con paréntesis como hacemos en matemáticas. Ejecutar las siguientes instrucciones, comprobando que los resultados son los esperados.

- a)  $4 - (3 + 2)$       b)  $(4 - 3) + 2$

Cuando no usamos paréntesis, tenemos que tener cuidado con la *precedencia* (cuál se aplica primero) de los operadores. Por ejemplo, si ponemos  $2 + 3 \times 4$ , sabemos que tenemos que calcular primero  $3 \times 4$  y a eso agregarle 2, o sea, « $\times$ » tiene mayor precedencia que « $+$ ».

No es claro cómo evaluar  $2 - 3 + 4 - 5$ . Parte del problema es que el signo « $-$ » se usa tanto para indicar la resta como para indicar el inverso aditivo de un número. También ayuda a la confusión el que la suma es conmutativa ( $2 + 3 = 3 + 2$ ), pero la resta no ( $2 - 3 \neq 3 - 2$ ).

Algo similar pasa con la expresión  $3/4 \times 5$ : no queda claro si nos referimos a  $(3/4) \times 5$  o a  $3/(4 \times 5)$ .

Ante la ausencia de paréntesis, Python evalúa expresiones como  $2 - 3 + 4 - 5$  y  $3 / 4 * 5$  de izquierda a derecha, esto es, como si hiciera los reemplazos

$$a - b \rightarrow a + (-b) \quad y \quad a/b \rightarrow a \times b^{-1}$$

antes de la evaluación.

**E 3.6.** Conjeturar el valor y luego verificarlo con Python:

a)  $2 - 3 + 4 - 5$       b)  $3 / 4 * 5 / 6$  


**E 3.7.** La expresión de Python  $-3**-4$ , ¿es equivalente en matemáticas a  $(-3)^{-4}$  o a  $-3^{-4}$ ?

¿Cuáles son las precedencias de Python en este caso? 

En el curso trataremos de evitar construcciones como la de los ejercicios anteriores, agregando paréntesis aunque sean redundantes: MVQSYNQF.<sup>(1)</sup>


**E 3.8.** Desde las matemáticas, ¿es  $\sqrt{3}$  entero?, ¿y  $\sqrt{4}$ ?

¿De qué tipo son  $3**(1/2)$  y  $4**(1/2)$ ? 

**E 3.9.** En un triángulo rectángulo, un cateto mide 12 y el otro 5. Calcular cuánto mide la hipotenusa usando Python. 

**E 3.10.** Compré 4 botellas a \$ 100 cada una y 7 a \$ 60 cada una.

Resolver con lápiz y papel (o mentalmente) y luego hacer las cuentas con Python:

- a) ¿Cuántas botellas compré?
- b) ¿Cuánto gasté en total?
- c) En promedio, ¿cuál fue el costo por botella? 

**E 3.11 (Inflación).** a) Si la inflación el primer año fue del 22 %, y el segundo año fue del 25 %, ¿cuánto cuesta ahora un artículo que dos años atrás costaba \$ 100?

<sup>(1)</sup> Más vale que sobre y no que falte.

*Aclaración:* hablamos de un artículo genérico, seguramente habrá artículos que aumentaron más y otros menos que la inflación indicada.

- b) En el apartado anterior, ¿cuál dirías que fue la inflación anual *promedio* de estos dos años?

*Aclaración:* buscamos un valor  $r$  tal que si la inflación anual hubiese sido constantemente  $r$ , en dos años habríamos obtenido la misma inflación acumulada que teniendo 22 % y 25 %.

✎ La *media geométrica* de los números positivos  $x_1, \dots, x_n$  es  $\sqrt[n]{x_1 \cdots x_n}$ . Se demuestra que

$$\sqrt[n]{x_1 \cdots x_n} \leq \frac{x_1 + \cdots + x_n}{n},$$

con igualdad si y sólo si  $x_1 = x_2 = \cdots = x_n$ .

- c) El kilo de pan hoy me cuesta \$ 60 y (exactamente) dos años atrás me costaba \$ 30. ¿Qué estimación podrías dar sobre el promedio de inflación en cada uno de estos dos años?
- d) En determinado año la inflación interanual fue de 36 %, ¿cuál fue, en promedio, la inflación mensual (ese año)?
- e) Si la inflación mensual es de 3 % durante los 12 meses del año, ¿cuál será la inflación anual? ¶

**E 3.12.** Además de las operaciones entre números, podemos usar algunas funciones como el valor absoluto de  $x$ ,  $|x|$ , que se escribe **abs(x)** en Python, o el redondeo de decimal a entero, **round(x)**, que nos da el entero más próximo a  $x$ .

- a) Ver qué hace **abs** poniendo **help(abs)** y luego repetir para **round**.

✎ Cuando escribimos comandos como **help** o **round** en IDLE, es posible poner unas pocas letras y completar el comando —o al menos obtener una lista de posibilidades— introduciendo una tabulación (tecla «tab» o similar).

b) Conjeturar y evaluar el resultado:

- i) `abs(12)`                      ii) `round(12.3)`
- iii) `round(-5.67)`              iv) `abs(-3.21)`

c) Evaluar:

- i) `abs`                      ii) `round`

y observar que al poner una función (como `abs`) sin argumento, Python responde diciendo que es una función (en este caso, propia).

d) Python diferencia entre mayúsculas y minúsculas. Probar con los siguientes, viendo que da error:

- i) `Abs(2)`                      ii) `ABS(2)`



**E 3.13.** `round` admite un argumento opcional indicando la cantidad de decimales deseados para el redondeo.

a) Evaluar:

- i) `round(123.4567, 3)`                      ii) `round(765.4321, 2)`

b) Escribir el promedio calculado en el [ejercicio 3.10.c](#)) redondeado al centavo.

c) El precio de cierto auto 0 km es de \$ 239 400: expresarlo redondeado a unidades de 1000 (mil) y de 10 000 (diez mil) usando `round`.



**E 3.14.** Cuando  $x$  es un número, `type(x)` nos dice si  $x$  es entero (`int`) o decimal (`float`) según Python.

Ver los resultados de las siguientes instrucciones:

- a) `type(12)`                      b) `type(12.3)`
- c) `round(12.3)`                      d) `type(round(12.3))`
- e) `98 // 76`                      f) `type(98 // 76)`
- g) `98.0 // 76.0`                      h) `type(98.0 // 76.0)`

🔗 `int` viene de integer, o entero. `float` viene de floating point (punto flotante), el nombre de la codificación para números decimales (estudiaremos esta codificación en el [capítulo 16](#)).



Es posible pasar de uno a otro tipo de número usando `int` (para pasar de `float` a `int`) o `float` (para pasar de `int` a `float`). Claro que al pasar de decimal a entero perdemos los decimales después de la coma.

E 3.15. Analizar los resultados de:

- a) `int(12.3)`                      b) `type(int(12.3))`  
 c) `float(-45)`                      d) `type(float(-45))`



E 3.16. Explorar la diferencia entre `int` y `round` cuando aplicados a números decimales. Por ejemplo, evaluar estas funciones en 1.2, 1.7, -1.2, -1.7, 1.5, 2.5.

También poner `help(round)` (que ya vimos) y `help(int)`.

- ☞ *`int` toma el primer entero hacia el 0, `round` toma el entero más cercano, con una regla especial cuando el decimal termina en .5.*



E 3.17. Python puede trabajar con enteros de cualquier tamaño, pero los tamaños de los decimales es limitado.

- a) Calcular `12**34` y luego `12.0**34`, y observar que Python representa al decimal con una variante de *notación científica* incorporando la letra `e` para indicar una potencia de 10:

$$4.922235242952027\text{e}+36 \leftrightarrow 4.922235242952027 \times 10^{36}.$$

⚠ ¡No debe confundirse este `e` con el número  $e = 2.71828\dots$ !

- b) Del mismo modo, poner `1 / 12**34` y ver cómo se representa en Python.
- c) Escribir en la notación de Python:  $12.3 \times 10^{45}$  y  $12.3 \times 10^{-45}$  (sin usar `*` ni `**` sino `e`), observando cómo los representa Python.
- d) Poner `123**456` viendo que obtenemos un número muy grande.
- e) Poner `123.0**456.0` viendo que obtenemos un error (el mensaje de error indica `Result too large` o *resultado demasiado grande*.)



Matemática:	$\pi$	$e$	$\sqrt{x}$	sen	cos	tan	$e^x$	log
Python:	pi	e	sqrt(x)	sin	cos	tan	exp(x)	log

Cuadro 3.1: Traducciones entre matemáticas y el módulo `math`.

f) ¿Qué pasará si ponemos `float(123)**float(456)`?

g) Probar también con `123.0**456` y `123**456.0`.

✎ Python puede representar cualquier número entero (sujeto a la capacidad de la memoria), pero el máximo número decimal que puede representar es aproximadamente  $1.7977 \times 10^{308}$ . El número  $123^{456} \approx 9.925 \times 10^{952}$  es demasiado grande y Python no puede representarlo como número decimal.

En el [capítulo 16](#) estudiaremos estos temas con más cuidado.



## 3.2. El módulo `math`

Python tiene un conjunto relativamente pequeño de operaciones y funciones matemáticas predeterminadas. Para agregar nuevas posibilidades —de matemáticas y de muchas otras cosas— Python cuenta con el mecanismo de *módulos*, cuyos nombres indicaremos *con estas letras*.

Así, para agregar funciones y constantes matemáticas apelamos al módulo `math`, que agrega las funciones trigonométricas como seno, coseno y tangente, la exponencial y su inversa el logaritmo, y las constantes relacionadas  $\pi = 3.14159 \dots$  y  $e = 2.71828 \dots$ .

Para usar `math` ponemos

```
import math
```

A partir de ese momento podemos emplear las nuevas posibilidades usando las traducciones de matemática a Python indicadas en el [cuadro 3.1](#), recordando que en las instrucciones de Python hay que anteponer «`math.`». Por ejemplo,  $\pi$  se escribe `math.pi` en Python.

Aunque hay otras maneras de «importar» módulos,

*nosotros sólo usaremos la forma*

```
| import nombre_del_módulo
```

*importando siempre todo el contenido del módulo.*

**E 3.18.** Poner `import math` en la terminal, y luego realizar los siguientes apartados:

- Poner `help(math)`, para ver las nuevas funciones disponibles.
- Evaluar `math.sqrt(2)`. ¿Qué pasa si ponemos sólo `sqrt(2)` (sin `math`)?
- ¿Hay diferencias entre `math.sqrt(2)` y `2**(1/2)`? Probar con otros valores (en vez de 2).

**E 3.19.** Usando el módulo `math`:

- Ver qué hace `math.trunc` usando `help(math.trunc)`.
- ¿Cuál es la diferencia entre `round` y `math.trunc`? Encontrar valores del argumento donde se aprecie esta diferencia.
- ¿Cuál es la diferencia entre `int(1.23)` y `math.trunc(1.23)`? Probar con otros valores decimales, incluyendo negativos (en vez de 12.34).

☞ *Los dos ejercicios anteriores muestran que a veces podemos hacer el mismo cálculo de distintas formas con Python.*

**E 3.20 (funciones trigonométricas).** En Python, los argumentos de las funciones trigonométricas deben estar dados en radianes. En todo caso podemos pasar de grados a radianes multiplicando por  $\pi/180$ . Así,  $45^\circ = 45^\circ \times \pi/180^\circ = \pi/4$  (radianes).

*Python tiene las funciones `math.radians` y `math.degrees` que no usaremos.*



Poniendo `import math` en la terminal de IDLE, realizar los siguientes apartados:

- Evaluar `math.pi`.
- Sin usar calculadora o compu, ¿cuáles son los valores de  $\cos 0$  y  $\sin 0$ ? Comparar con la respuesta de Python a `math.cos(0)` y `math.sin(0)`.
- Calcular  $\sin \pi/4$  usando Python.
- Calcular  $\tan 60^\circ$  usando `math.tan` con argumento  $60 \times \pi/180$ .
- Calcular seno, coseno y tangente de  $30^\circ$  usando Python (las respuestas, por supuesto, deberían ser aproximadamente  $1/2$ ,  $\sqrt{3}/2$  y  $\sqrt{3}/3$ ).
- ¿Cuánto es  $\tan 90^\circ$ ?, ¿y según Python?

Python no puede evaluar  $\tan 90^\circ$  exactamente: `math.pi` y `math.tan` son sólo aproximaciones a  $\pi$  y  $\tan$ .

Con  $\log_a b$  denotamos el logaritmo en base  $a$  de  $b$ , recordando que  $\log_a b = c \Leftrightarrow a^c = b$  (si  $a$  y  $b$  son números reales positivos,  $a \neq 1$ ). Es usual poner  $\ln x = \log_e x$ , aunque siguiendo la notación de Python acá entenderemos que  $\log x$  (sin base explícita) es  $\log_e x = \ln x$ .


**E 3.21 (exponenciales y logaritmos).** En la terminal de IDLE, poner `import math` y realizar los siguientes apartados:

- Calcular `math.e`.
- Usando `help`, averiguar qué hace la función `math.log` y calcular `math.log(math.e)`. ¿Es razonable el valor obtenido?
- Para encontrar el logaritmo en base 10 de un número, podemos usar `math.log(número, 10)` pero también podemos poner directamente `math.log10(número)`: ver qué hace `math.log10` usando `help`, y calcular  $\log_{10} 123$  de las dos formas. ¿Hay diferencias?, ¿debería haberlas?
- Calcular  $\log_{10} 5$  aproximadamente, y verificar el resultado calculando  $10^{\log_{10} 5}$  (¿cuánto debería ser?).

- e) Otro logaritmo que usaremos es el de base 2,  $\log_2$ , que en Python podemos poner como `math.log2`.

Calcular  $\log_2 10$  y  $\log_{10} 2$  con Python. ¿Debería haber alguna relación entre estos valores?

- f) Ver qué hace `math.exp`. Desde la teoría, ¿qué diferencia hay entre `math.exp(x)` y `(math.e)**x`?

Calcular ambas funciones y su diferencia con Python para distintos valores de  $x$  (e. g.,  $\pm 1$ ,  $\pm 10$ ,  $\pm 100$ ). 


**E 3.22.** Recordemos que para  $x \in \mathbb{R}$ , la función *piso* se define como

$$\lfloor x \rfloor = \max \{n \in \mathbb{Z} : n \leq x\},$$

y la función *techo* se define como


$$\lceil x \rceil = \min \{n \in \mathbb{Z} : n \geq x\}.$$

En el módulo `math`, estas funciones se representan respectivamente como `math.floor` (*piso*) y `math.ceil` (*techo*). Ver el funcionamiento de estas funciones calculando el piso y el techo de  $\pm 1$ ,  $\pm 2.3$  y  $\pm 5.6$ .

Comparar con los resultados de `int` y `round` en esos valores. 

**E 3.23 (cifras I).** La cantidad de cifras (en base 10) para  $n \in \mathbb{N}$  puede encontrarse usando  $\log_{10}$  (el logaritmo en base 10), ya que  $n$  tiene  $k$  cifras si y sólo si  $10^{k-1} \leq n < 10^k$ , es decir, si y sólo si  $k-1 \leq \log_{10} n < k$ , o sea si y sólo si  $k = 1 + \lfloor \log_{10} n \rfloor$ .

- a) Usar estas ideas para encontrar la cantidad de cifras (en base 10) de  $123^{456}$ .

 Recordar el [ejercicio 3.17](#) y la nota al final de éste.

- b) Encontrar la cantidad de cifras en base 2 de  $2^{64}$  y de 1023, usando lápiz y papel y luego con Python. 

### 3.3. Comentarios

- La presentación como calculadora es típica en sistemas interactivos como *Matlab* o *Mathematica*. En particular, aparece en el [tutorial de Python](#).
- Es posible hacer que Python se comporte como una calculadora gráfica haciendo, por ejemplo, gráficos en 2 y 3 dimensiones. En los apuntes no trabajaremos directamente con estas posibilidades.



## Capítulo 4

# Tipos de datos básicos

En este capítulo vemos que no sólo podemos trabajar con números enteros o decimales, sino que también podemos hacer preguntas esperando respuestas de «verdadero» o «falso», y que podemos poner carteles en las respuestas.

### 4.1. ¿Por qué hay distintos tipos de datos?

Uno se pregunta por qué distinguir entre entero y decimal. Después de todo, las calculadoras comúnmente no hacen esta distinción, trabajando exclusivamente con decimales, y en matemáticas 2 y 2.0 son distintas representaciones de un mismo entero.

Parte de la respuesta es que todos los objetos se guardan como «ristras» de ceros y unos en la memoria, y ante algo como 10010110 la computadora debe saber si es un número, o parte de él, o una letra o alguna otra cosa. Una calculadora sencilla, en cambio, siempre trabaja con el mismo tipo de objetos: números decimales.

La variedad de objetos con los que trabaja la compu se ve reflejada en los lenguajes de programación. Por ejemplo, los lenguajes Pascal y C trabajan con enteros, decimales, caracteres y otros objetos contruidos a partir de ellos. De modo similar, entre los tipos básicos de Python

tenemos los dos que hemos visto, **int** o entero (como 123) y **float** o decimal (como 45.67) y ahora veremos dos más:

- **bool** o *lógico*, siendo los únicos valores posibles **True** (verdadero) y **False** (falso).
  - ✎ Poniendo **help(bool)** vemos que, a diferencia de otros lenguajes, para Python **bool** es una subclase de **int**, lo que tiene sus ventajas pero trae aparejada muchas inconsistencias.
  - ✎ **bool** es una abreviación de Boolean, que podemos traducir como *booleanos* y pronunciar *buleanos*. Los valores lógicos reciben también ese nombre en honor a G. Boole (1815–1864).
- **str** o *cadena de caracteres*, como 'Mateo Adolfo'.
  - ✎ **str** es una abreviación del inglés string, literalmente *cuerda*, que traducimos como *cadena*, en este caso de caracteres (*character string*).

## 4.2. Tipo lógico

Repasemos un poco las operaciones lógicas, recordando que en matemáticas representamos con  $\wedge$  a la conjunción «y», con  $\vee$  a la disyunción «o» y con  $\neg$  a la negación «no».

**E 4.1.** En cada caso, decidir si la expresión matemática es verdadera o falsa:

- a)  $1 = 2$     b)  $1 > 2$     c)  $1 \leq 2$     d)  $1 \neq 2$   
 e)  $\frac{3}{5} < \frac{8}{13}$     f)  $1 < 2 < 3$     g)  $1 < 2 < 0$     h)  $(1 < 2) \vee (2 < 0)$  ¶

En Python también tenemos expresiones que dan valores **True** (*verdadero*), o **False** (*falso*). Por ejemplo:

```
>>> 4 < 5
True
>>> 4 > 5
False
```

Matemáticas:	=	≠	>	≥	<	≤	∧ (y)	∨ (o)	¬ (no)
Python:	==	!=	>	>=	<	<=	and	or	not

Cuadro 4.1: Traducciones entre matemáticas y Python.


En el [cuadro 4.1](#) vemos cómo convertir algunas expresiones entre matemática y Python, recordando que «and», «or» y «not» son los términos en inglés para *y*, *o* y *no*, respectivamente.

Así, para preguntar si  $4 = 5$  ponemos `4 == 5`, y para preguntar si  $4 \neq 5$  ponemos `4 != 5`:


```
>>> 4 == 5
False
>>> 4 != 5
True
```

*En Python el significado de « = » no es el mismo que en matemáticas, lo que da lugar a numerosos errores:*

- la igualdad « = » en matemáticas se representa con « == » en Python,
- la instrucción « = » en Python es la asignación que estudiaremos con más cuidado en el [capítulo 5](#).

**E 4.2.** Traducir a Python las expresiones del [ejercicio 4.1](#), observando que en Python (como en matemáticas) la expresión `a < b < c` es equivalente a `(a < b) and (b < c)`. 

**E 4.3.** Conjeturar y verificar en Python:


- |                               |                                      |
|-------------------------------|--------------------------------------|
| a) <code>not True</code>      | b) <code>True and False</code>       |
| c) <code>True or False</code> | d) <code>False and (not True)</code> |
- 




E 4.4. Usando `type(algo)`, ver que las expresiones

- a) `4 < 5`      b) `4 != 5`      c) `4 == 5`

son de tipo `bool`. 

E 4.5. ¿Cuál es el resultado de `1 > 2 + 3`?, ¿cuál es la precedencia entre `>` y `+`? ¿Y entre `>` y otras operaciones aritméticas (como `*`, `/`, `**`)? 


E 4.6. Python considera que `1` y `1.0` son de distinto tipo, pero que sus valores son iguales. Evaluar:

- a) `type(1) == type(1.0)`      b) `1 == 1.0` 

E 4.7 (errores numéricos). Usando `==`, `<=` y `<`, comparar

- a) `123` con `123.0`,  
b) `123` con `123 + 1.0e-10`,  
c) `123` con `123 + 1.0e-20`.

¿Alguna sorpresa?

🔎 Estudiaremos problemas de este tipo con más profundidad en el capítulo 16. 

En general, los lenguajes de programación tienen reglas un tanto distintas a las usadas en matemáticas para las evaluaciones lógicas. Por ejemplo, la evaluación de expresiones en las que aparecen conjunciones (« y ») o disyunciones (« o ») se hace de izquierda a derecha y dejando la evaluación en cuanto se obtiene una expresión falsa al usar « y » o una expresión verdadera al usar « o ». Esto hace que, a diferencia de matemáticas, `and` y `or` no sean operadores conmutativos.

🔎 Estamos considerando que sólo aparece « y » o sólo aparece « o ». Cuando aparecen ambos entremezclados, hay que tener en cuenta la precedencia de `or` y `and` que estudiamos en el ejercicio 4.10.

E 4.8 (cortocircuitos en lógica). Evaluar y comparar:

- a) `1 < 1/0`

- b) `False` and `(1 < 1/0)`
- c) `(1 < 1/0)` and `False`
- d) `True` or `(1 < 1/0)`

☞ Aunque incorrectas desde las matemáticas aceptaremos el uso de «cortocircuitos» porque su uso está muy difundido en programación.



E 4.9. Evaluar y estudiar los resultados:

- a) `type(1)`
- b) `type(1) == type(2)`
- c) `type(1) == int`
- d) `type(1) == float`
- e) `type(1.1) == int`
- f) `type(1.1) == float`



E 4.10 (precedencias).

- a) Conjeturar el valor de `True or False and False`.
- b) Evaluar la expresión anterior en Python y decidir si corresponde a `(True or False) and False` o a `True or (False and False)`.
- c) Decidir si en Python hay precedencias entre `and` y `or` o si se evalúan de izquierda a derecha (como `3 / 4 * 5`).



Otra diferencia importante con las matemáticas es que Python considera que todo objeto tiene un valor verdadero o falso. Esto es muy confuso y no sucede en otros lenguajes de programación:

*En el curso están prohibidas las construcciones que mezclan valores u operaciones numéricas con lógicas como:*

- `1 and 2 or 3`,
- `False + True`,
- `-1 < False`,

*válidas en Python pero carentes de sentido en matemáticas.*



### 4.3. Cadenas de caracteres

Los *caracteres* son las letras, signos de puntuación, dígitos, y otros símbolos que usamos para la escritura. Las *cadenas de caracteres* son sucesiones de estos caracteres que en Python van encerradas entre comillas, ya sean sencillas, `'`, como en `'Ana Luisa'`, o dobles, `"`, como en `"Ana Luisa"`.

A diferencia de los tipos que acabamos de ver (entero, decimal y lógico), las cadenas de caracteres son objetos «compuestos», constituidos por objetos más sencillos (los caracteres).

- 🔗 Python no tiene el tipo carácter y esta descripción no es completamente cierta. Para representar *un* carácter en Python, simplemente consideramos *una cadena de un elemento*. Por ejemplo, la letra «a» se puede poner como la cadena `'a'`.

**E 4.11.** Ingresar en la terminal `'Ana Luisa'` y averiguar su tipo poniendo `type('Ana Luisa')`. 🔗

**E 4.12.** Ver que para Python `'Ana Luisa'` y `"Ana Luisa"` son lo mismo usando `==`. 🔗

**E 4.13.** Las comillas simples `'` no se pueden intercambiar con las dobles `"`, pero se pueden combinar. Ver los resultados de:

- a) `'mi"` b) `'mi" mama"` c) `"mi' mama"` d) `'mi" "mama' 🔗`

**E 4.14.** Como vimos en el caso de `int` y `float` (ejercicio 3.15), podemos cambiar el tipo de un objeto, aunque cuando no se trata de números pueden surgir dificultades.

- a) Cuando pasamos un objeto de tipo `int`, `float` o `bool` a `str`, básicamente obtenemos lo que se imprimiría en la terminal, sólo que entre comillas. Evaluar:

- i) `str(1)` ii) `str(1.)` iii) `str(False)`



- b) Hay cosas que no tienen sentido:

- i) `int('pepe')` ii) `float('pepe')` iii) `bool('pepe')`

c) A veces, pero no siempre, podemos volver al objeto original:

i) `int(str(1))`      ii) `float(str(1.2))`

✎ Recordar el [ejercicio 3.15](#).

d) A veces el resultado puede llevar a error:

| `bool(str(False))`



**E 4.15 (longitud de cadena).** Con `len` podemos encontrar la longitud, es decir, la cantidad de caracteres de una cadena. Por ejemplo, `'mama'` tiene cuatro caracteres (contamos las repeticiones), y por lo tanto `len('mama')` da 4.

Conjeturar el valor y luego verificarlo con Python en los siguientes casos:

a) `len('me mima')`      b) `len('mi mama me mima')`



**E 4.16 (concatenación).** *Concatenar* es poner una cadena a continuación de otra, para lo cual Python usa «+», el mismo símbolo que para la suma de números.

✎ Para evitar confusiones, en el texto a veces ponemos `_` para indicar un espacio en blanco.

a) Evaluar:

i) `'mi' + 'mama'`      ii) `'mi' + '_' + 'mama'`

b) A pesar del signo «+», la concatenación no es conmutativa (como sí lo es la suma de números). Ver el resultado de

| `1 + 2 == 2 + 1`  
| `'pa' + 'ta' == 'ta' + 'pa'`

c) Evaluar `'mi mama' + 'me mima'`.

¿Cómo podría modificarse la segunda cadena para que el resultado de la concatenación sea `'mi mama me mima'`?

d) Conjeturar los valores de

i) `len('mi mama') + len('me mima')`,

ii) `len('mi mama me mima')`

y luego verificar la validez de las conjeturas. ¶

**E 4.17 (cadena vacía).** La *cadena vacía* es la cadena `' '`, o la equivalente `''`, sin caracteres y tiene longitud 0. Es similar a la noción de conjunto vacío en el contexto de conjuntos o el cero en el de números.

- a) No hay que confundir `' '` (comilla-comilla) con `'_ '` (comilla-espacio-comilla). Conjeturar el resultado y luego evaluar:

i) `len('')`      ii) `len(' _ ')`

- b) ¿Cuál será el resultado de `' ' + 'mi mama'`? Verificarlo con Python. ¶

**E 4.18.**

- a) «`+`» puede usarse tanto para sumar números como para concatenar cadenas de caracteres, pero no podemos mezclar números con cadenas: ver qué resultado da `2 + 'mi'`.

- a) ¿Podría usarse `-` con cadenas como en `'mi' - 'mama'`?

- b) Cambiando ahora «`+`» por «`*`», verificar si los siguientes son válidos en Python, y en caso afirmativo cuál es el efecto:

i) `2 * 'ma'`      ii) `'2' * 'ma'`      iii) `'ma' * 2` ¶

Con `help(str)` podemos ver las muchas operaciones que tiene Python para cadenas. Nosotros veremos sólo unas pocas en el curso: las que ya vimos, algunas más dentro del contexto general de sucesiones en el [capítulo 9](#), y cuando trabajemos con formatos y archivos de texto en el [capítulo 11](#).

## 4.4. print (imprimir)

**E 4.19 (print).** `print` nos permite imprimir en la terminal expresiones de distintos tipos.

- a) Evaluar y observar las diferencias:

i) `print(123)`      ii) `print(12.3)`  
 iii) `print('mi mama')`      iv) `print('123')`

- b) `print` puede tener más de un argumento, no necesariamente del mismo tipo, en cuyo caso se separan con un espacio al imprimir. Evaluar

```
print('Según María,', 1.23,
      'lo que dice Pepe es', 1 > 2)
```

- c) Evaluar y observar las diferencias entre:

i) `print(123456)` y `print(123, 456)`.

ii) `print('Hola mundo')` y `print('Hola', 'mundo')`.

- d) `print` puede no tener argumentos explícitos. Evaluar y observar las diferencias entre:

i) `print`

ii) `print()`

iii) `print('')`

iv) `print('\n')`

- e) ¿Hay alguna diferencia entre `False` y `print('False')`?, ¿cómo lo podríamos determinar?

☞ *Que dos objetos tengan la misma representación visual, no quiere decir que los objetos sean iguales.*



**E 4.20.** Aunque matemáticamente no tiene mucho sentido, la multiplicación de un entero por una cadena que vimos en el [ejercicio 4.18](#) es útil para imprimir.

Evaluar:

a) `print(70 * '-')`

b) `print(2 * '-' + 3 * ' ' + 4 * '*')`



**E 4.21.** Observar las diferencias entre los resultados de los siguientes apartados, y determinar el efecto de agregar `\n`:

a) `print('mi', 'mama', 'me', 'mima')`

b) `print('mi\n', 'mama', 'me\n\n', 'mima')`

c) `print('mi', '\n', 'mama', 'me', '\n\n', 'mima')`



**E 4.22.** Cuando la cadena es muy larga y no cabe en un renglón, podemos usar `\` (barra invertida) para dividirla. Por ejemplo, evaluar:

```
print('Este texto es muy largo, no entra en \
un renglón y tengo que ponerlo en más de uno.')
```

¿Cómo podría usarse «+» (concatenación) para obtener resultados similares?

*Ayuda:* + puede ponerse al principio o final de un renglón, siempre que esté dentro de un paréntesis.

✎ Aunque se puede agrandar la ventana de IDLE, es sumamente recomendable no escribir renglones con más de 80 caracteres. ¶

**E 4.23.** Podemos usar `\` para indicar que el renglón continúa, pero si queremos imprimir «\» tenemos que poner `\\` dentro del argumento de `print`.

- Comparar `print('\\')` con `print('\\')`.
- Conjeturar y luego verificar los resultado de `print('/\')` y de `print('//\')`.
- Imprimir una «casita» usando `print`:

```
  /\
 /__\
|    |
|    |
|__|
```

¶

## 4.5. Comentarios

- El [ejercicio 4.23](#) está tomado de [Litvin y Litvin \(2010\)](#).



# Capítulo 5

## Variables y asignaciones

Frecuentemente queremos usar un mismo valor varias veces en los cálculos, o guardar un valor intermedio para usarlo más adelante, o simplemente ponerle un nombre sencillo a una expresión complicada para referirnos a ella en adelante. A estos efectos, muchas calculadoras tienen memorias para conservar números, y no es demasiada sorpresa que los lenguajes de programación tengan previsto un mecanismo similar: la *asignación*.

### 5.1. Asignaciones en Python

Cuando escribimos alguna expresión como `123` o `-5.67` o `'mi mama'`, Python guarda estos valores como *objetos* en la memoria, lo que esquematizamos en la [figura 5.1.a](#)).

En Python podemos hacer referencia posterior a estos objetos, poniendo un nombre, llamado *identificador*, y luego una asignación, indicada por «`=`», que relaciona el identificador con el objeto. Así, el conjunto de instrucciones

```
a = 123
suma = -5.67
```



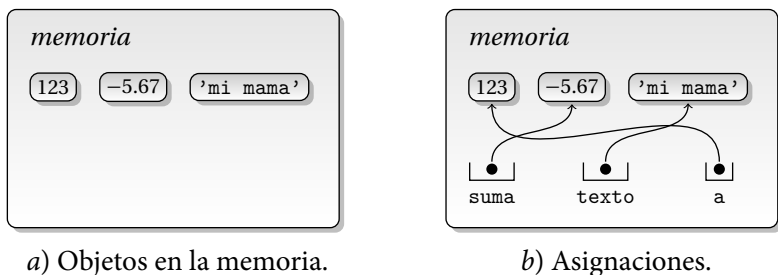


Figura 5.1: Objetos en la memoria.

```
texto = 'mi mama'
```

hace que se relacionen los identificadores a la izquierda (**a**, **suma** y **texto**) con los objetos a la derecha (**123**, **-5.67** y **'mi mama'**), como esquematizamos en la [figura 5.1.b](#)).

- 🔗 Recordemos que los datos, incluyendo instrucciones, se guardan en la memoria de la computadora como ristas de ceros y unos. En la [figura 5.1](#) ponemos los valores «humanos» para entender de qué estamos hablando.
- 🔗 También recordemos que la igualdad en matemática, «=», se indica por «==» en Python.

A fin de conservar una nomenclatura parecida a la de otros lenguajes de programación, decimos que **a**, **suma** y **texto** son *variables*, aunque en realidad el concepto es distinto en Python, ya que son una *referencia*, similar al *vínculo* o *enlace* (*link*) en una página de internet.

**E 5.1.** a) Poner **a = 123**, y comprobar que el valor de **a** no se muestra.

- 🔗 Al hacer la asignación, **a** es una variable con identificador «a» y valor «123».

- b) Poner simplemente **a** y verificar que aparece su valor (123).  
 c) Poner **type(a)**.

🔗 *El tipo de una variable es el tipo del objeto al cual hace referencia.*

d) Poner **b** (al cual no se le ha asignado valor) y ver qué pasa.

⇒ En Python, el valor de una variable y la misma variable, no existen si no se ha hecho una asignación a ella. ¶

Python trabaja con objetos, cada uno de los cuales tiene una identidad, un tipo y un valor.

✎ La *identidad* es el lugar (dirección) de memoria que ocupa el objeto. No todos los lenguajes de programación permiten encontrarla, pero sí Python. Nosotros no estudiaremos esa propiedad.

Podemos pensar que la asignación **a = algo** consiste en:

- Evaluar el miembro derecho **algo**, realizando las operaciones indicadas si las hubiere. Si **algo** involucra variables, las operaciones se hacen con los valores correspondientes. El valor obtenido se guarda en algún lugar de la memoria.

Recordar que si **algo** es sólo el identificador de una variable (sin otras operaciones), el valor es el del objeto al cual referencia la variable.

- En **a** se guarda (esencialmente) la dirección en la memoria del valor obtenido.

Por ejemplo:

<b>a = 1</b>	→	<b>a</b> es una referencia a 1 (figura 5.2, izquierda)
<b>b = a</b>	→	<b>b</b> también es una referencia a 1 (figura 5.2, centro)
<b>a = 2</b>	→	ahora <b>a</b> es una referencia a 2 (figura 5.2, derecha)
<b>a</b>	→	el valor de <b>a</b> es 2
<b>b</b>	→	<b>b</b> sigue siendo una referencia a 1

**E 5.2.** En cada caso, predecir el resultado del grupo de instrucciones y luego verificarlo en la terminal de Python, recordando que primero se evalúa el miembro derecho y luego se hace la asignación:

a) <b>a = 1</b> <b>a</b>	b) <b>a = 1</b> <b>a = a + 1</b> <b>a</b>	c) <b>a = 1</b> <b>a = a + a</b> <b>a</b>
-----------------------------	---	---

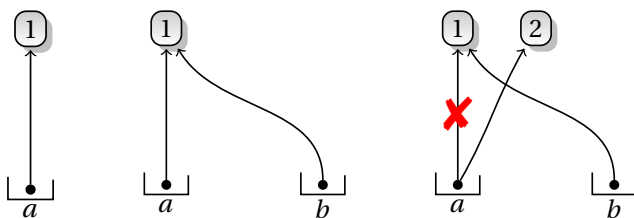


Figura 5.2: Ilustración de asignaciones.

d) $a = 3$ $b = 2$ $c = a + b$ $d = a - b$ $e = d / c$ $e$	e) $a = 3$ $b = 2$ $a = a + b$ $b = a - b$ $a$ $b$
---	---



Los identificadores (los nombres de las variables) pueden tener cualquier longitud (cantidad de caracteres), pero no se pueden usar todos los caracteres. Por ejemplo, no pueden tener espacios, ni signos de puntuación como « , » o « . », ni signos como « + » o « - » para no confundir con operaciones, y no deben empezar con un número. No veremos las reglas precisas, que son un tanto complicadas y están en el [manual de referencia](#).

Aunque se permiten, es conveniente que los identificadores no empiecen con guión bajo « \_ », dejando esta posibilidad para identificadores de Python. También es conveniente no poner tildes, como en « á », « ñ » o « ü ».

Finalmente, observamos que identificadores con mayúsculas y minúsculas son diferentes (recordar el [ejercicio 3.12](#)).

**E 5.3.** a) Decidir cuáles de los siguientes son identificadores válidos en Python, y comprobarlo haciendo una asignación:

- |                               |                              |                               |
|-------------------------------|------------------------------|-------------------------------|
| i) <code>PepeGrillo</code>    | ii) <code>Pepe_Grillo</code> | iii) <code>Pepe_Grillo</code> |
| iv) <code>Pepe-Grillo</code>  | v) <code>Pepe\Grillo</code>  | vi) <code>Pepe/Grillo</code>  |
| vii) <code>Grillo,Pepe</code> | viii) <code>Pepe12</code>    | ix) <code>34Pepe</code>       |

- b) ¿Cómo podría detectarse que `PepeGrillo` y `pepegriilo` son identificadores distintos?

*Sugerencia:* asignarles valores distintos.



**E 5.4 (palabras reservadas).** Python tiene algunas palabras *reservadas* que no pueden usarse como identificadores.

- Para encontrarlas, ponemos en la terminal `help()` y a continuación `keywords` (*keyword* = palabra clave).
- Esto quiere decir que no podemos poner `and = 5`: verificarlo.
- Por otro lado, podemos usar —por ejemplo— `float` como identificador, a costa de que después no podamos usar la función correspondiente. Para verificarlo, en la terminal poner sucesivamente:

```
type(7.8)
float
float(123)
float = 456
float
float(123)
type(7.8)
```

☞ *Es mejor no usar nombres de expresiones de Python como identificadores.*

✎ No es necesario memorizar las palabras reservadas. El propósito del ejercicio es destacar que hay expresiones que no pueden usarse como identificadores.

✎ IDLE pone con colores distintos las palabras claves como `and` y otras instrucciones como `float`.



**E 5.5.** Dados los enteros  $a$  y  $b$ ,  $b > 0$ , el *algoritmo de la división* encuentra enteros  $q$  y  $r$ ,  $0 \leq r < b$  tales que  $a = qb + r$ , aún cuando  $a$  no

sea positivo, y esto se traduce en Python poniendo

$$q = a // b \quad \text{y} \quad r = a \% b. \quad (5.1)$$

✎ Más adelante veremos la función `divmod` que hace una tarea similar con una única instrucción.

Para ver el comportamiento de Python en estas operaciones, evaluamos `q` y `r` para distintos valores de `a` y `b`: positivos, negativos, enteros o decimales.

- ¿Qué pasa si `b` es 0?, ¿y si `a` y `b` son ambos 0?
- ¿Qué hace Python cuando `b` es entero pero negativo? (o sea: ¿qué valores de `q` y `r` se obtienen?).
- Si `b` es positivo pero decimal, al realizar las operaciones en (5.1), Python pone `q` decimal y `r` decimal, pero esencialmente `q` es entero pues `q == int(q)`.

Verificar esto tomando distintos valores decimales de `a` y `b` (`b` positivo).

- Si `a` es decimal y `b = 1`, determinar si el valor de `q` coincide con `int(a)` o `round(a)` o ninguno de los dos.
- Dar ejemplos «de la vida real» donde tenga sentido considerar:
  - `a` decimal y `b` entero positivo,
  - `b` decimal y positivo.

*Ayuda:* hay varias posibilidades, por ejemplo pensar en horas, en radianes y  $b = 2\pi$ , y en general en cosas cíclicas.

- ¿Qué hace Python cuando `b` es negativo y decimal? 

**E 5.6 (cifras II).** En el [ejercicio 3.23](#) vimos una forma de determinar la cantidad de cifras de un entero  $n$  (cuando escrito en base 10) usando  $\log_{10}$ . Como Python puede escribir el número, otra posibilidad es usar esa representación.


- Ejecutar:

```
| a = 123
```

```
b = str(a)
len(b)
```

- b) En el apartado anterior, ¿qué relación hay entre `len(b)` y la cantidad de cifras en `a`?

Proponer un método para encontrar la cantidad de cifras (en base 10) de un entero positivo basado en esta relación.

- c) Usando el método propuesto en el apartado anterior, encontrar la cantidad de cifras (en base 10) de  $123^{456}$  y comparar con el resultado del [ejercicio 3.23](#). 

## 5.2. None

Así como en matemáticas es útil tener el número 0 o el conjunto vacío, en programación es útil tener un objeto que «tenga valor nulo», o que sea «el dato vacío». En Python este valor se llama **None** (*nada* o *ninguno*).

✎ En otros lenguajes se lo denomina **Null** (*nulo*) en vez de **None**.

Como el número 0 o el conjunto vacío, el concepto no es fácil de entender. Por ejemplo, el [manual de la biblioteca](#) dice sobre **None**:

Es el único valor del tipo **NoneType** y es usado frecuentemente para indicar la ausencia de valor ...

lo que es bastante confuso.

**None** es una constante como lo son `123` o `'mi mama me mima'`, en realidad más análoga a `0` o `''`. Es una de las palabras claves que vimos en el [ejercicio 5.4](#) y no podemos usarla como identificador, del mismo modo que no podemos poner `0 = 123`.

Aunque no se crea, ya hemos visto **None** en acción: es el valor retornado por `print`. Exploramos el tema en el siguiente ejercicio, pero antes de hacerlo no estaría de más repasar el [ejercicio 4.19](#).

## E 5.7 (None).

- a) Poniendo en la terminal:

```
| abs(-4)
```

y luego

```
| a = abs(-4)
```

vemos que una diferencia entre estos dos grupos de instrucciones es que en el primer caso se imprime el valor de `abs(-4)` (y no existe la variable `a`), mientras que en el segundo el valor de `abs(-4)` se guarda en la variable `a`, y no se imprime nada.

Verificar el valor y tipo de `a`.


- b) Cambiando `abs` por `print`, pongamos:

```
| print(-4)
```

y luego

```
| a = print(-4)
```

¿Cómo se comparan los resultados de estos dos grupos con los del apartado anterior? En particular, ¿cuál es el valor de `a`?, ¿y su tipo?

- c) Siempre con `a = print(-4)`, poner `a == None` para ver que, efectivamente, el valor de `a` es `None`.
- d) Poner `None` en la terminal y ver que no se imprime resultado alguno. Comparar con los apartados anteriores. 



- ⚠ La discusión sobre `None` nos alerta sobre la diferencia entre *los resultados* de una acción y *los valores retornados* por esa acción.

Por ejemplo, `print` tiene como resultado que se imprima una cadena de caracteres, pero el valor retornado es `None`.

Como hemos visto numerosas veces, la asignación `a = 4` tiene como resultado que se relacione la variable `a` con el número 4, y uno sospecharía que el valor retornado por esta acción es `None`, pero esto no es así.

En Python hay una oscura diferencia entre *expresiones* y *sentencias*. `1 + 2` y `print('mi mama')` son expresiones y retornan

valores, mientras que `a = 3` es una sentencia y no retorna valor alguno.

Una forma de distinguirlos es justamente haciendo una asignación al resultado. Como hicimos en el [ejercicio 5.7.b](#)), no hay problemas en hacer la asignación `a = print('Ana')`, pero `a = (b = 1)` da error.

No nos meteremos en esas profundidades.

## 5.3. Comentarios

- Es posible averiguar las variables definidas usando `globals`, obteniendo un *diccionario*. Como tantas otras cosas, no veremos ni `globals` ni la estructura de diccionario.





# Capítulo 6

## Módulos

La asignación nos permite referir a algo complejo con un nombre sencillo. Del mismo modo, a medida que vamos haciendo acciones más complejas, el uso de la terminal se hace incómodo y es conveniente ir agrupando las instrucciones, guardándolas en algún archivo para no tener que escribirlas nuevamente, como mencionamos en la [sección 2.3](#). Prácticamente todos los lenguajes de programación tienen algún mecanismo para esta acción, y en Python los archivos correspondientes se llaman *módulos*.

Como mencionamos en la [sección 3.2](#), en estas notas indicamos los nombres de los módulos *conestasletras*, en general omitiendo la extensión (que pueden no tener).

Los módulos de Python vienen básicamente en dos sabores:

- Los módulos *estándares*, que forman parte de la distribución de Python y que amplían las posibilidades del lenguaje. Nosotros vamos a usar explícitamente muy pocos de éstos, sólo *math* y *random*.

🔗 Varios módulos se instalan automáticamente al iniciar IDLE.

- Los módulos que construimos nosotros, ya sea porque Python no tiene un módulo estándar que haga lo que queremos (o no

sabemos que lo tiene), o, como en este curso, porque queremos hacer las cosas nosotros mismos.

Estos módulos son archivos de texto, en donde guardamos varias instrucciones, eventualmente agrupadas en una o más funciones (tema que veremos en el [capítulo 7](#)).

Por otro lado, los módulos se pueden usar de dos formas distintas:

- Si el módulo se llama *pepe*, usando la instrucción

```
| import pepe
```

(sin incluir extensión) ya sea en la terminal o desde otro módulo.

- Si el módulo está en un archivo de texto y se puede abrir en una ventana de IDLE, con el menú *Run Module* de IDLE.

En la práctica, este segundo método es equivalente a escribir todas las sentencias del módulo en la terminal de IDLE y ejecutarlas.

Estas dos formas dan resultados distintos, y nos detendremos a explorar estas diferencias en la [sección 6.4](#). En la mayoría de los casos, usaremos el primer método con los módulos estándares como *math* y el segundo con los módulos que construimos.

- ✎ Para resaltar esta diferencia, a veces los archivos que se usan de la segunda manera (abriéndolos con IDLE) se llaman *scripts* (*guiones*) en vez de módulos. Nosotros no haremos esta distinción: como los que construyamos se pueden usar de cualquiera de las dos formas, para simplificar llamaremos módulos a todos.

Veamos algunos ejemplos.

## 6.1. Módulos propios

En esta sección construiremos nuestros propios módulos, esto es, archivos de texto con extensión *.py* donde se guardan instrucciones de Python.

- ✎ Los archivos deben estar codificados en utf-8, lo que IDLE hace automáticamente.

## E 6.1 (Hola Mundo).

- Abrir una ventana nueva en IDLE distinta de la terminal (menú *File* → *New File*), escribir en ella `print('Hola Mundo')` en un único renglón, y guardar en un archivo con nombre *elprime-ro.py*, prestando atención al directorio en donde se guarda.
  - ✎ Python es muy quisquilloso con las *sangrías*: el renglón no debe tener espacios (ni tabulaciones) antes de `print`.
  - ✎ En este apunte tomaremos directorio (*directory*) como sinónimo de carpeta (*folder*). *Directorio* es el nombre tradicional en computación y es el que usa Python en los módulos de manejo de (¡ejem!) directorios y archivos. La designación de *carpeta* se usa más en las computadoras personales (con interfases gráficas).
  - ✎ Es conveniente guardar *todos* nuestros módulos en el mismo directorio (veremos más adelante por qué), y por prolijidad es mejor que no sea el directorio principal. Por ejemplo, los módulos podrían ponerse en un directorio *Python* o *computación* dentro del directorio *Documentos* (o similar) del usuario.
  - ✎ Algunas versiones de MS-Windows son especialmente rebeldes, pues por defecto el directorio para guardar los módulos es el mismo donde está instalado el ejecutable: no es una buena idea. En este caso la recomendación es crear el directorio como descripto anteriormente y usar un «acceso directo» a él en el «escritorio», permitiendo un rápido acceso desde el diálogo de «guardar».
  - ✎ Hay instrucciones de Python que permiten cambiar el directorio de trabajo (*working directory*), pero no las veremos en el curso.
- Buscando el menú correspondiente en IDLE (*Run* → *Run Module*), ejecutar los contenidos de la ventana y verificar que en la terminal de IDLE se imprime `Hola Mundo`.



- c) *holamundo* es una versión donde se agregaron renglones al principio, que constituyen la *documentación* que explica qué hace el módulo. Incluir estos renglones (donde el texto completo empieza y termina con `"""`), y ejecutar nuevamente el módulo, verificando que el comportamiento no varía.

- ✍ Es una sana costumbre (léase: exámenes) documentar los módulos. En este caso es un poco redundante, pues son pocos renglones y se puede entender qué hace leyéndolos: *lo que abunda no daña*, MVQSYNQF,...
- ✍ El uso de `"""` es similar al de las comillas simples `'` y dobles `"` para encerrar cadenas de caracteres.
- ✍ Recordar no poner más de 80 caracteres por renglón, preferentemente 72 o menos.
- ✍ La *guía de estilos de Python* sugiere que la documentación tenga un renglón inicial que empieza con `"""` y un texto corto; si hay un único renglón debe terminarse con las `"""` finales, pero si hay más texto después del renglón corto, los otros párrafos deben ir separados por un renglón en blanco, y toda la documentación terminar con sólo `"""` en el renglón final.

Más detalles pueden encontrarse en la guía mencionada.

- d) Poniendo ahora `print(__doc__)`, aparecerá el texto que agregamos al principio de *holamundo*.
- e) Sin embargo, poniendo `help(holamundo)` da error.

☞ *Porque no hemos usado import.*



## 6.2. Ingreso interactivo de datos

Cuando trabajamos sólo con la terminal de IDLE, podemos asignar o modificar valores sin mucho problema. La situación cambia si se ejecuta o importa un módulo y queremos ingresar datos a medida que se requieren.

**E 6.2.** *holapepe* es una variante de *holamundo*, donde el usuario ingresa su nombre, y la computadora responde con ese nombre. La función *input* se encarga de leer el dato requerido.

- a) Ejecutar el módulo, comprobando su comportamiento, y usando también *print(\_\_doc\_\_)* para leer la documentación.
- b) *pepe* es una variable donde se guarda el nombre ingresado. El nombre ingresado puede no ser «pepe», y puede tener espacios como en «Mateo Adolfo». Verificar el nombre ingresado poniendo *pepe* en la terminal de IDLE.
- c) Python toma cualquier entrada como cadena de caracteres, incluyendo comillas si las hubiere. Probar con las siguientes entradas, ejecutando cada vez el módulo y verificando cada una de ellas poniendo *pepe* antes de ejecutar la siguiente.

i) *que'se'o'*      ii) *123\_123"*      iii) *agu"ero*

- d) ¿Cómo podríamos modificar el renglón final para que se agregue una coma « , » inmediatamente después del nombre? Por ejemplo, si el nombre ingresado es «Mateo», debería imprimirse algo como *Hola Mateo, encantada de conocerte*.

*Sugerencia:* usar concatenación (*ejercicio 4.16*).

- e) Los renglones que se escriben en la ventana del módulo *holapepe* no deben tener sangrías, aunque pueden haber renglones en blanco (pero sin espacios ni tabulaciones): agregar un renglón sin caracteres (con «retorno» o similar) entre el renglón con el primer *print* y el renglón que empieza con *pepe*, y comprobar que el comportamiento no varía. ¶

**E 6.3.** *sumardos* es un módulo donde el usuario ingresa dos objetos, y se imprime la suma de ambos. Además, al comienzo se imprime la documentación del módulo.

- a) Sin ejecutar el módulo, ¿qué resultado esperarías si las entradas fueran *mi* y *mama* (sin comillas)?

Ejecutar el módulo, y ver si se obtiene el resultado esperado.

- b) ¿Qué resultado esperarías si las entradas fueran 2 y 3?  
Ejecutar el módulo, y ver si se obtiene el resultado esperado.

☞ *Python siempre toma las entradas de `input` como cadenas de caracteres.*

- c) Si queremos que las entradas se tomen como números enteros, debemos pasar de cadenas de caracteres a enteros, por ejemplo cambiando

```
a = input('Ingresar algo: ')
```

por


```
a = int(input('Ingresar un entero: '))
```

y de modo similar para `b`.

Hacer estos cambios, cambiar también la documentación y guardar los cambios en el módulo *sumardosenteros*.

Probar el nuevo módulo (ejecutándolo cada vez) con las entradas:


i) 2 y 3      ii) 4.5 y 6      iii) mi y mama

- d) ¿Cómo modificarías el módulo para que Python interprete las entradas como dos números decimales? 

## 6.3. Documentación y comentarios en el código

Sobre todo cuando escribimos muchas instrucciones es bueno ir agregando documentación para que cuando volvamos a leerlas después de un tiempo entendamos qué quisimos hacer. Una forma de documentar es incluir un texto entre triple comillas `"""`, que —como veremos— cuando va al principio es la respuesta a `help`. Otra forma es poner el símbolo `#`: Python ignora este símbolo y todo lo que le sigue en ese renglón. Esta acción se llama *comentar* el texto.

E 6.4. Veamos el efecto en el módulo *holapepe*:

- a) Agregar `#` al principio del primer renglón que empieza con `print`, ejecutar el módulo y ver el efecto producido.
- b) Manteniendo el cambio anterior, en el renglón siguiente cambiar `input()` por `input('¿Cómo te llamas?')`. ¿Cuál es el efecto?
- c) El nombre ingresado queda demasiado junto a la pregunta en `input`. ¿Cómo se podría agregar un espacio a la pregunta para que aparezcan separados?
- d) «Descomentar» el renglón con `print` —o sea, sacar el `#`— y cambiar la instrucción por `print('Hola, soy la compu')` viendo el efecto. 

Cuando se programa profesionalmente, es muy importante que el programa funcione aún cuando los datos ingresados sean erróneos, por ejemplo si se ingresa una letra en vez de un número, o el número 0 como divisor de un cociente. Posiblemente se dedique más tiempo a esta fase, y a la interfaz entre la computadora y el usuario, que a hacer un programa que funcione cuando las entradas son correctas.

Nosotros supondremos que siempre se ingresan datos apropiados, y no haremos (salvo excepcionalmente) detección de errores. Tampoco nos preocuparemos por ofrecer una interfaz estéticamente agradable. En cambio:

*Siempre trataremos de dejar claro mediante la documentación y comentarios qué hace el programa y preguntando qué datos han de ingresarse en cada momento.*

## 6.4. Usando `import`

Al poner `import módulo`, Python no busca el módulo en toda la computadora (lo que llevaría tiempo) sino en una lista de directorios,

en la que *siempre* están los módulos estándares como *math*.

Cuando ejecutamos un módulo en IDLE (como hicimos con *holamundo*), el directorio donde está el archivo correspondiente pasa a formar parte de la lista. De allí la recomendación de poner todos los módulos construidos por nosotros en un mismo directorio, de modo de poder acceder a nuestros módulos rápidamente.

- ✎ Es posible —usando instrucciones que no veremos— encontrar la lista completa de directorios y también cambiar la lista, por ejemplo, agregando otros directorios.
- ✎ En algunas versiones de MS-Windows se puede indicar el directorio que se usa inicialmente por defecto.

- E 6.5.** a) En la terminal poner `a = 5` y verificar el valor poniendo `a`.  
b) Poner `import math` y luego calcular `math.log(3)`.  
c) Reiniciar la terminal de IDLE (con el menú *Shell* → *Restart Shell*), y preguntar el valor de `a` (sin asignar valor a `a`), viendo que da error.

Poner nuevamente `math.log(3)` (sin `import math`) viendo que también da error.

- ☞ *Al reiniciar IDLE (con «Restart Shell») se pierden las asignaciones hechas anteriormente.*



- E 6.6.** Resolver los siguientes apartados en una nueva sesión de IDLE.<sup>(1)</sup>

- a) poner `import sumardos` y ver que da error.
- ☞ *Para importar un módulo no estándar, tenemos que agregar el directorio donde se encuentra a la lista de directorios donde busca Python.*
- b) Abrir el módulo *holamundo* (y no *sumardos*) en IDLE y ejecutarlo (*Run* → *Run Module*), viendo que se imprime `Hola Mundo` (y no hay error).

---

<sup>(1)</sup> *Restart Shell* puede no ser suficiente, dependiendo del sistema operativo.



- c) Volver a poner `import sumardos` en la terminal, viendo que ahora no da error.
- ☞ *Al ejecutar un módulo propio, el directorio donde está se incorpora a la lista de directorios donde Python busca los módulos.*
- d) Al poner `help(sumardos)` (y no `holamundo`), la documentación de `sumardos` aparece al principio de la respuesta.
- ☞ *Cuando hacemos `import módulo` podemos acceder a su documentación con `help(módulo)`.*
- 🔗 Comparar con el [ejercicio 6.1.e](#).
- e) Poner `import holapepe` y luego `help(holapepe)`.
- ☞ *Es posible importar un número arbitrario de módulos, pero sólo se puede ejecutar uno con Run Module en IDLE.*
- f) ¿Qué pasa si ponemos `help(holamundo)`?
- 🔗 Recordar el [ejercicio 6.1.e](#). 🔗

Al importar (con `import`) un módulo, se agregan instrucciones (y variables) a las que ya hay, pero para acceder a los objetos del módulo importado y distinguirlos de los que hayamos definido antes, debemos agregar el nombre del módulo al identificador del objeto, como en `math.pi` o `math.cos`.

Decimos que los objetos creados por el módulo están en el *contexto* (o *espacio* o *marco*) determinado por el módulo, y que son *locales* a él. Los objetos creados fuera de estos contextos se dicen *globales*.

Veamos cómo es esto, ayudándonos con la [figura 6.1](#).

### E 6.7 (contextos).

- a) Poner `import math` y luego `math.pi`, viendo que obtenemos un valor más o menos familiar, pero que si ponemos sólo `pi` (sin `math.`) nos da error.

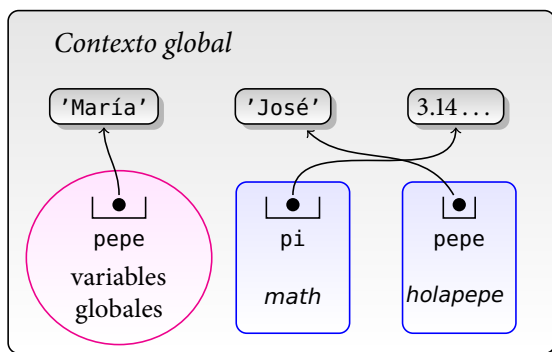


Figura 6.1: Contextos global y de módulos en el [ejercicio 6.7](#).

- ☞ `math.pi` es una variable en el contexto `math`, mientras que `pi` no existe como variable (global) pues no se le ha asignado valor alguno.
- b) Abrir el módulo `holapepe`, ejecutarlo con el menú `Run → Run Module` de IDLE, ingresar el nombre `María`, y verificar lo ingresado poniendo `pepe`.
  - ☞ `pepe` es una variable global, con valor `'María'`.
- c) Sin cerrar la terminal de IDLE, poner `import holapepe`. Nos volverá a preguntar el nombre y ahora pondremos `José`. Preguntando por `pepe` volvemos a obtener `María`, pero si ahora ponemos `holapepe.pepe`, obtendremos `José`.
  - ☞ `holapepe.pepe` es una variable dentro del contexto determinado por `holapepe`, y su valor es `'José'`. En cambio, `pepe` es una variable global y su valor es `'María'`.



## Capítulo 7

# Funciones

Es tedioso escribir las mismas instrucciones varias veces o aún ejecutar un módulo cada vez que queremos probar con distintas entradas. Una solución es juntar las instrucciones en una *función*, que funcionan... esteeee... *se comportan* como las funciones en matemática: dando argumentos de entrada se obtiene una respuesta o salida.

Como en el caso de asignaciones y módulos, la idea es no repetir acciones. Aunque la ventaja de su uso irá quedando más clara a lo largo del curso, en general podemos decir que las funciones son convenientes para:

- poner en un único lugar cálculos idénticos que se realizan en distintas oportunidades,
- o poner por separado alguna acción permitiendo su fácil reemplazo (y con menor posibilidad de error),
- y no menos importante, haciendo el programa más fácil de entender, dejando una visión más global y no tan detallada en cada parte.

Para definir una función en Python usamos el esquema:

```
def función(argumento/s):  
    instrucciones
```

El primer renglón *no debe tener sangrías* (espacios entre el margen izquierdo y la primer letra) y debe terminar con « : », y el segundo *debe tener una sangría de exactamente 4 espacios*.

- El grupo de instrucciones que comienza al aumentar el sangrado se llama *bloque*, y termina cuando la sangría disminuye.

Un bloque de instrucciones puede contener sub-bloques, cuyos sangrados son mayores que el que los contiene.

A medida que vayamos usando los sangrados quedará más claro el efecto.

En la jerga de programación, cuando ponemos  $y = f(x)$  decimos que hacemos una *llamada a f*, que  $x$  se *pasa a* —o es un argumento de—  $f$ , y que  $f(x)$  *retorna* o *devuelve y*.

Sin embargo, a diferencia de matemáticas, en programación las funciones pueden no tener argumentos. En Python siempre retornan algún valor, que puede ser «no visible» como `None` (ver [sección 5.2](#)).

## 7.1. Ejemplos simples

**E 7.1.** Siguiendo las ideas del [ejercicio 6.2](#), vamos a definir una función `hola1` que dado un argumento, imprime «Hola» seguido del argumento. Por ejemplo, queremos que `hola1('Mateo')` imprima «Hola Mateo».

- a) En una nueva ventana de IDLE (no la terminal) empezamos a construir un módulo poniendo la documentación general:

```
"""Ejemplos de funciones con y sin argumentos."""
```

Guardar los cambios poniendo el nombre `holas` al módulo.

- b) Dejando al menos un renglón en blanco después de la documentación, agregar la definición de la siguiente función en el módulo `holas`:

```
def hola1(nombre):  
    """Imprime 'Hola' seguido del argumento."""  
    print('Hola', nombre)
```

- ✎ IDLE pone la sangría automáticamente cuando el renglón anterior termina en « : ».
  - ✎ Observar el uso de ' dentro de las "" en la documentación.
- c) Guardar los cambios y ejecutar el módulo (*Run* → *Run Module*).
- d) En la terminal poner `help(hola1)` para leer la documentación.
- e) En la terminal poner `hola1(`, terminando con « ( » sin otros caracteres ni «retorno», viendo que aparece un cartel con el argumento a poner y la documentación.

☞ *La documentación de una función debe comenzar con un resumen de un único renglón corto, con no más de 60 caracteres, ya que es lo que aparecerá al hacer el procedimiento anterior (poner el nombre de la función seguido de « ( » y nada más).*

- f) Probar la función con las siguientes entradas:

- i) `hola1('Mateo')`
- ii) `hola1('123')`
- iii) `hola1(123)`
- iv) `hola1(1 + 2)`
- v) `hola1(2 > 5)`

- ✎ Como en matemáticas, primero se evalúa el argumento y luego la función (en este caso `hola1`).
- g) ¿Qué pasa si ponemos `hola1` (sin paréntesis ni argumentos) en la terminal? Ver que la respuesta es similar a poner, por ejemplo, `abs` (sin paréntesis ni argumentos).
- h) ¿Qué pasa si ponemos `nombre` en la terminal?

☞ *La variable `nombre` es local a la función `hola1` y no se conoce afuera, siguiendo un mecanismo de contextos similar al de los módulos (ejercicio 6.7).*

- ✎ El tema se explica con más detalle en la [sección 7.3](#).



La función `hola1` que acabamos de definir toma el argumento que hemos llamado `nombre`, pero podemos definir funciones sin argumen-

tos.

**E 7.2.** Dejando al menos un renglón en blanco después de la definición de la función `hola1`, agregar la definición de la siguiente función en el módulo `holas`:

```
def hola2():  
    """Ejemplo de función sin argumento.  
  
    Imprime el dato ingresado.  
    """  
    print('Hola, soy la compu')  
    nombre = input('¿Cuál es tu nombre? ')  
    print('Encantada de conocerte', nombre)
```

✍ `hola2` no tiene argumentos, pero tenemos que poner los paréntesis tanto al definirla como al invocarla.

✍ ¡Atención a las sangrías!


a) Guardar los cambios, ejecutar el módulo, y verificar su documentación poniendo `help(hola2)` y luego `hola2()` (como hicimos en el [ejercicio 7.1.e](#)).

☞ Al poner `hola2()` en la terminal (terminando en «`(`» sin otros caracteres ni «retorno») sólo aparece el primer renglón de la documentación.

En cambio, al poner `help(hola2)` aparecen todos los renglones de la documentación y no sólo el primero.

*Por eso es de suma importancia que la documentación de una función tenga el primer renglón corto con un resumen de lo que hace, separado por un renglón en blanco del resto de la documentación (si la hubiere).*

- b) Verificar el comportamiento de `hola2`, poniendo `hola2()`.
- c) `nombre` es una variable *local* a la función `hola2`: poner `nombre` en terminal y ver que da error.

A diferencia de las variables locales a módulos, no es fácil acceder a las variables locales dentro de una función: ver que `hola2.nombre` y `hola2().nombre` dan error. 

**E 7.3.** Habiendo ejecutado el módulo `holas`, hacer la asignación `a = hola1('Mateo')` y verificar que `a` es `None` al terminar.

Repetir con `a = hola2()`. 

Los ejercicios anteriores nos muestran varias cosas. Por un lado, que en un mismo módulo se pueden definir varias funciones. Por otro, las funciones del módulo `holas` realizan la acción de imprimir pero el valor que retornan es `None`, como la función `print` (ver el [ejercicio 5.7](#)).

**E 7.4 (return).** Basados en el [ejercicio 6.3](#), ahora definimos una función que toma *dos* argumentos y que retorna un valor que podemos usar, para lo cual usamos `return`.


- a) En una ventana nueva de IDLE, poner

```
def sumar2(a, b):
    """Suma los argumentos."""
    return a + b    # resultado de la función
```

Guardar en un archivo adecuado, y ejecutar el módulo.

- b) Conjeturar y verificar el resultado de los siguientes (viendo que no es `None`):
  - i) `sumar2(2, 3.4)`                      ii) `sumar2('pi', 'pa')`
  - iii) `sumar2(1)`                              iv) `sumar2(1, 2, 3)`
- c) Poniendo en la terminal:

```
a = sumar2(1, 2)
a
```

vemos que el efecto es el mismo que haber puesto `a = 1 + 2`. 

E 7.5. a) En cada una de las funciones `hola1` y `hola2` de los ejercicios 7.1 y 7.2 incluir al final la instrucción `return None`, y hacer las asignaciones `a = hola1('toto')` y `a = hola2()`, viendo que en ambos casos el resultado es efectivamente `None`.

b) Cambiando el `return None` anterior por sólo `return`, y ver el efecto.

⇒ Si la última instrucción en la definición de una función es `return` o `return None`, el efecto es el mismo que no poner nada, y el valor retornado es `None`. ¶

E 7.6. Las funciones que definimos pueden usar funciones definidas por nosotros. Poner:

```
def f(x):  
    """Multiplicar por 2."""  
    return 2*x  
  
def g(x):  
    """Multiplicar por 4."""  
    return f(f(x))
```

y evaluar  $f$  y  $g$  en  $\pm 1$ ,  $\pm 2$  y  $\pm 3$ . ¶


## 7.2. Funciones numéricas

En esta sección miramos funciones cuyos argumentos son uno o más números, y el resultado es un número.


E 7.7. Si  $f$  indica la temperatura en grados Fahrenheit, el valor en grados centígrados (o Celsius) está dado por  $c = 5(f - 32)/9$ .

- Expresar en Python la ecuación que relaciona  $c$  y  $f$ .
- En la terminal de IDLE, usar la expresión anterior para encontrar  $c$  cuando  $f$  es 0, 10, 98.
- Recíprocamente, encontrar  $f$  cuando  $c$  es -15, 10, 36.7.




- d) ¿Para qué valores de `f` el valor de `c` (según Python) es entero? ¿Y matemáticamente?
- e) ¿En qué casos coinciden los valores en grados Fahrenheit y centígrados?
- f) Definir una función `acelsius` en Python tal que si `f` es el valor de la temperatura en grados Fahrenheit, `acelsius(f)` da el valor en grados centígrados, y verificar el comportamiento repitiendo los valores obtenidos en [b](#)).
- g) Recíprocamente, definir la función `afahrenheit` que dado el valor en grados centígrados retorne el valor en grados Fahrenheit, y verificarlo con los valores obtenidos en [c](#)). 

**E 7.8.** Construir una función `gmsar(g, m, s)` que ingresando la medida de un ángulo en grados (`g`), minutos (`m`) y segundos (`s`), retorne la medida en radianes.

Por ejemplo,  $12^{\circ} 34' 56.78''$  son  $0.21960 \dots$  radianes. 

**E 7.9.** Definir una función que retorne la cantidad de cifras de un número en base 10 de dos formas:

- a) Usando las ideas del [ejercicio 3.23](#).
- b) Usando las ideas del [ejercicio 5.6](#). 

## 7.3. Variables globales y locales

Las funciones también son objetos, y cuando definimos una función se fabrica un objeto de tipo `function` (*función*), con su propio contexto, y se construye una variable que tiene por identificador el de la función y hace referencia a ella.

Así como para los módulos, en el contexto de una función hay objetos como instrucciones y variables, que son locales a la función.

Los argumentos (si los hubiere) en la definición de una función se llaman *parámetros formales* y los que se especifican en cada llamada se llaman *parámetros reales*. Al hacer la llamada a la función, se realiza

un mecanismo similar al de asignación, asignando cada uno de los parámetros formales a sus correspondientes parámetros reales.

De este modo, si `f` está definida por

```
def f(a, b):  
    ...
```

`a` y `b` son variables locales a la función, y cuando hacemos la llamada `f(x, y)` se hacen las asignaciones `a = x` y `b = y` antes de continuar con las otras instrucciones en `f`.

En líneas generales cuando *dentro del cuerpo de una función* encontramos una variable, entonces:



- si la variable es un argumento formal, es *local* a la función.
- si la variable nunca está en el miembro izquierdo de una asignación (siempre está a la derecha), la variable es *global* y tendrá que ser asignada *antes* de llamar a la función,
- si la variable está en el miembro izquierdo de una asignación, la variable es *local* a la función y se desconoce afuera (como en los *contextos* definidos por módulos),...
- ... salvo que se declare como *global* con `global`, y en este caso será... ¡global!

**E 7.10.** En una nueva sesión de IDLE, realizar los siguientes apartados.

a) Poner en la terminal:

```
def f(x):  
    a = 3  
    return x + a
```

☞ La variable `a` en la definición es *local* a la función, y puede existir una variable `a` fuera de la función que sea *global*.

b) Poner sucesivamente:

```
f(2)  
f  
type(f)
```

y estudiar los resultados.

c) Poner

```
g = f
g(2)
g
type(g)
```

y comprobar que los tres últimos resultados son idénticos al anterior.

d) Poner

```
a = 1
f(a)
a
```

y observar que el valor de la variable global `a` no ha cambiado. 🐦

Podemos pensar que los distintos elementos que intervienen en el [ejercicio 7.10](#) están dispuestos como se muestra en la [figura 7.1](#):

- La función tiene instrucciones, datos y variables locales, que ocupan un lugar propio en memoria, formando un objeto que puede referenciarse como cualquier otro objeto.
- En este caso, `f` y `g` referencian a la misma función.
- La variable global `a` referencia a `1`, mientras que la variable `a` local a la función referencia a `3`.
- La instrucción `f(a)` hace que se produzca la asignación `x = a`, pero `x` es local a la función mientras que `a` es global.

Como vemos, el tema se complica cuando los identificadores (los nombres) de los parámetros formales en la definición de la función coinciden con los de otros fuera de ella, o aparecen nuevas variables en la definición de la función con los mismos nombres que otras definidas fuera de ella.

El siguiente ejercicio muestra varias alternativas más, y alentamos a pensar otras.

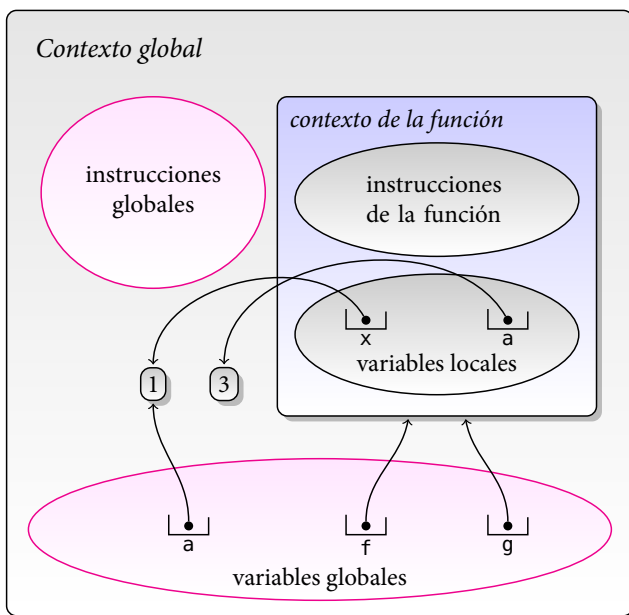


Figura 7.1: Variables globales y locales.

**E 7.11.** a) En la terminal de IDLE poner

```
def f(x):
    """Retorna su argumento."""
    print('el argumento ingresado fue:', x)
    return x
```

y explicar los resultados de los siguientes:

i)	<code>a = f(1234)</code>	ii)	<code>a = 1234</code>	iii)	<code>x = 12</code>
	<code>a == 1234</code>		<code>b = f(a)</code>		<code>y = f(34)</code>
	<code>x</code>		<code>a == b</code>		<code>x == y</code>

b) Reiniciar la terminal (*Shell* → *Restart Shell*), de modo que ni `a` ni `x` estén definidas, poner

```
def f(x):
```

```
def f():
```

```

    """Variable local a con problemas."""
    b = a    # b es local y a es global
    a = 1    # a es local porque se asigna
             # y entonces estamos en problemas
             # porque la usamos antes de asignar
    return a + b

```

Cambiar el orden de los renglones poniendo primero `a = 1` y luego `b = a`, y repetir.

f) ¿Cuál será el resultado de poner

```

x = 1
def f():
    """¿x es local o global?"""
    print(x)
    x = 2
    print(x)

f()

```

¿Por qué?

g) Reiniciar la terminal, poner

```

def f(x):
    """Trata de cambiar la variable global a."""
    global a
    a = 5          # a es... ¡global!
    return x + a

```

y explicar los resultados de las siguientes:

i)   a	ii)   f(1)	iii)   a = 2
	a	f(1)
		a

h) Poner

```

def f(x):
    """Trata de cambiar el argumento con 'global'."""
    global x    # el argumento no puede ser global
    x = 2

```

```
    return x
```

y ver que da error: una variable no puede ser a la vez argumento formal y global. ¶

- **E 7.12.** En otra tónica, podemos usar una variable local con el mismo identificador que la función como en

```
def f(x):
    """Ejemplo retorcido."""
    f = x + 1      # x y f son locales
    return f
```

y ejecutar el bloque

```
f
f(1)
f
```

Desde ya que este uso da lugar a confusiones, y aunque válido para Python, *en el curso está prohibido*. ¶

**E 7.13.** Las funciones pueden considerarse como objetos de la misma categoría que las variables, y podemos tener funciones locales a una función como en el módulo *flocal*.

- a) Ejecutar ese módulo y explicar el resultado del bloque:

```
x = 1
fexterna()
x
```

- b) ¿Cuál es el resultado de ejecutar `finterna()`?, ¿por qué? ¶

**E 7.14.** Siendo como variables, las funciones también pueden pasarse como argumentos a otras funciones, como se ilustra en el módulo *fargumento*.

Predecir los resultados de las siguientes y luego verificarlos:

- a) `aplicar(f, 1)`  
 b) `aplicar(g, 1)`

- c) `aplicar(f, aplicar(f, 1))`
- d) `aplicar(g, aplicar(f, 1))`
- e) `aplicar(g, aplicar(g, 1))`



## 7.4. Comentarios

- Ocasionalmente necesitaremos variables que no son ni locales ni globales para lo que se puede usar `nonlocal` que no veremos en el curso.
- Una variante a la de pasar una función como argumento de otra función ([ejercicio 7.14](#)) son las *funciones recursivas*, que estudiamos en los capítulos [14](#) y [17](#), pero que en pocas palabras pueden describirse como funciones que se llama a sí mismas.





## Capítulo 8

# Tomando control

Las cosas empiezan a ponerse interesantes cuando disponemos de *estructuras de control de flujo*, esto es, instrucciones que nos permiten tomar decisiones sobre si realizar o no determinadas instrucciones o realizarlas repetidas veces. Al disponer de estas estructuras, podremos verdaderamente comenzar a describir *algoritmos*, instrucciones (no necesariamente en un lenguaje de programación) que nos permiten llegar a determinado resultado, y apuntar hacia el principal objetivo de este curso: pensar en los algoritmos y cómo traducirlos a un lenguaje de programación.

Prácticamente todos los lenguajes de programación tienen distintas estructuras de control similares o equivalentes a las de *if* y *while* que estudiamos en este capítulo.

📖 La palabra *algoritmo* surge del nombre Abu Ja'far Muhammad ibn Musa al-Khwarizmi' (~780–850). También la palabra *álgebra* está relacionada con su obra así como *guarismo* con su nombre.

### 8.1. if (si)

Supongamos que al cocinar decidimos bajar el fuego si el agua hierve, es decir, realizar cierta acción si se cumplen ciertos requisitos.

Podríamos esquematizar esta decisión con la sentencia:

**si** el agua hierve **entonces** bajo el fuego.

A veces queremos realizar una acción si se cumplen ciertos requisitos, pero realizar una acción alternativa si no se cumplen. Por ejemplo, si para ir al trabajo podemos tomar el colectivo o un taxi —que es más rápido pero más caro que el colectivo— dependiendo del tiempo que tengamos decidiríamos tomar uno u otro, que podríamos esquematizar como:

**si** es temprano **entonces** tomo el colectivo **en otro caso** tomo el taxi.

En Python podemos tomar este tipo de decisiones, usando **if** (*si* en inglés) para el esquema **si... entonces...**, y el bloque se escribe como otros que ya hemos visto:

```
if condición:    # si el agua hierve entonces
    hacer algo   # bajo el fuego
```

usando « : » en vez de **entonces**.

Para la variante **si... entonces... en otro caso...** usamos **if** junto con **else** (*en otro caso* en inglés), escribiéndose como:

```
if condición:                # si es temprano entonces
    hacer algo                # tomo el colectivo
else:                        # en otro caso
    hacer otra cosa           # tomo el taxi
```

Por supuesto, inmediatamente después de **if** tenemos que poner una condición (una expresión lógica) que pueda evaluarse como verdadera o falsa.

En fin, cuando hay varias posibilidades, como en

1. **si** está Mateo **entonces** lo visito,
2. **en otro caso si** está Ana **entonces** la visito,
3. **si ninguna de las anteriores es cierta entonces** me quedo en casa.

en vez de poner algo como «else if» para **en otro caso si**, en Python se pone **elif**:

```
if condición:           # si está Mateo
    hacer algo           # entonces lo visito
elif otra condición:    # en otro caso si está Ana
    hacer otra cosa      # entonces la visito
else:                   # si ninguna de las anteriores entonces
    hacer una tercer cosa # me quedo en casa
```

Observemos que estamos dando prioridades: en el ejemplo, si Mateo está lo voy a visitar, y no importa si Ana está o no. En otras palabras, si tanto Ana como Mateo están, visito a Mateo y no a Ana.

Veamos algunos ejemplos concretos sencillos.

**E 8.1.** Supongamos que queremos determinar si el número  $x$  es o no positivo, imprimiendo un cartel adecuado.

El esquema a seguir sería algo como:

```
if x > 0:  # si x es positivo
    print(x, 'es positivo')
else:      # en otro caso
    print(x, 'no es positivo')
```

como hacemos en la función **espositivo** (en el módulo *ifwhile*).

- Estudiar la construcción y probar la función con distintos argumentos, numéricos y no numéricos.
- Cambiar la definición de la función de modo que *retorne* verdadero o falso en vez de imprimir, cambiando **print** por **return** adecuadamente.
- Al usar **return** en una función se termina su ejecución y no se realizan las instrucciones siguientes.

Ver si el bloque interno de la función del apartado anterior (las instrucciones a partir de **if x > 0**;) es equivalente a:

```
if x > 0:
    return True
```

```
| return False
```



**E 8.2 (piso y techo).** Recordando las definiciones de *piso* y *techo* (en el ejercicio 3.22), la función `piso` del módulo `ifwhile` imprime los valores correspondientes al piso de un número real usando una estructura `if... elif... else`.

- Estudiar las instrucciones de la función y comparar los resultados de la función `piso` con los de la función `math.floor` para  $\pm 1$ ,  $\pm 2.3$  y  $\pm 5.6$ .
- Cambiando `print` por `return` (en lugares apropiados), modificar `piso` de modo de *retornar* el valor del piso del argumento.
- Construir una función `techo` para retornar el techo de un número real.



**E 8.3 (signo de un número real).** La función  $\text{signo} : \mathbb{R} \rightarrow \mathbb{R}$  se define como:

$$\text{signo } x = \begin{cases} 1 & \text{si } x > 0, \\ -1 & \text{si } x < 0, \\ 0 & \text{si } x = 0. \end{cases}$$

Definir una función correspondiente en Python usando la estructura `if... elif... else`, y probarla con los argumentos  $\pm 1$ ,  $\pm 2.3$ ,  $\pm 5.6$  y 0.

- 🔍 Observar que  $|x| = x \times \text{signo } x$  para todo  $x \in \mathbb{R}$ , y un poco arbitrariamente definimos  $\text{signo } 0 = 0$ .
- 🔍 En este y otros ejercicios similares, si no se explicita «retornar» o «imprimir», puede usarse cualquiera de las dos alternativas (o ambas).



**E 8.4 (años bisiestos).** Desarrollar una función para decidir si un año dado es o no bisiesto.

- 🔍 Según el calendario gregoriano que usamos, los años bisiestos son aquellos divisibles por 4 excepto si divisibles por 100 pero no por 400. Así, el año 1900 no es bisiesto pero sí lo son 2012 y 2000.

Este criterio fue establecido por el Papa Gregorio XIII en 1582, pero no todos los países lo adoptaron inmediatamente. En el ejercicio adoptamos este criterio para cualquier año, anterior o posterior a 1582.

- ✍ A veces con un pequeño esfuerzo podemos hacer el cálculo más eficiente. Un esquema para resolver el problema anterior, si *anio* es el año ingresado, es

```
if anio % 400 == 0:
    print(anio, 'es bisiesto')
elif anio % 100 == 0:
    print(anio, 'no es bisiesto')
elif anio % 4 == 0:
    print(anio, 'es bisiesto')
else:
    print(anio, 'no es bisiesto')
```

Sin embargo, el esquema

```
if anio % 4 != 0:
    print(anio, 'no es bisiesto')
elif anio % 100 != 0:
    print(anio, 'es bisiesto')
elif anio % 400 != 0:
    print(anio, 'no es bisiesto')
else:
    print(anio, 'es bisiesto')
```


es más eficiente, pues siendo que la mayoría de los números no son múltiplos de 4, en la mayoría de los casos haremos sólo una pregunta con el segundo esquema pero tres con el primero. ¶

**E 8.5.** El Gobierno ha decidido establecer impuestos a las ganancias en forma escalonada: los ciudadanos con ingresos hasta \$ 30 000 no pagarán impuestos; aquéllos con ingresos superiores a \$ 30 000 pero que no sobrepasen \$ 60 000, deberán pagar 10 % de impuestos; aquéllos cuyos ingresos sobrepasen \$ 60 000 pero no sean superiores a \$ 100 000 deberán pagar 20 % de impuestos, y los que tengan ingresos superiores a \$ 100 000 deberán pagar 40 % de impuestos.

- a) Definir una función para calcular el impuesto dado el monto

de la ganancia.

- b) Modificarla para determinar también la ganancia neta (una vez deducidos los impuestos).
- c) Modificar las funciones de modo que el impuesto y la ganancia neta se calculen hasta el centavo (y no más).

*Sugerencia:* hay varias posibilidades. Una es usando `round` (ver `help(round)`). 

## 8.2. while (mientras)

La estructura `while` (*mientras* en inglés) permite realizar una misma tarea varias veces. Junto con `for` —que veremos más adelante— reciben el nombre común de *lazos* o *bucles*, y son *estructuras de repetición*.

Supongamos que voy al supermercado con cierto dinero para comprar la mayor cantidad posible de botellas de cerveza. Podría ir calculando el dinero que me va quedando a medida que pongo botellas en el carrito: cuando no alcance para más botellas, iré a la caja. Una forma de poner esquemáticamente esta acción es

**mientras** alcanza el dinero, poner botellas en el carrito.

En Python este esquema se realiza con la construcción

```
while condición:    # mientras alcanza el dinero,  
    hacer algo      # poner botellas en el carrito
```

donde, como en el caso de `if`, la condición debe ser una expresión lógica que se evalúa en verdadero o falso.

Observamos desde ya que:

- Si la condición no es cierta al comienzo, nunca se realiza la acción: si el dinero inicial no me alcanza, no pongo ninguna botella en el carrito.

- En cambio, si la condición es cierta al principio, debe modificarse con alguna acción posterior, ya que en otro caso llegamos a un «lazo infinito», que nunca termina.

Por ejemplo, si tomamos un número positivo y le sumamos 1, al resultado le sumamos 1, y así sucesivamente mientras los resultados sean positivos. O si tomamos un número positivo y lo dividimos por 2, luego otra vez por 2, etc. mientras el resultado sea positivo.

✎ Al menos en teoría. Como veremos más adelante, la máquina tiene un comportamiento «propio».

Por cierto, en el ejemplo de las botellas en el supermercado podríamos realizar directamente el cociente entre el dinero disponible y el precio de cada botella, en vez de realizar el lazo **mientras**. Es lo que vemos en el próximo ejercicio.

**E 8.6.** La función **resto** (en el módulo **ifwhile**) calcula el resto de la división de  $a \in \mathbb{N}$  por  $b \in \mathbb{N}$  mediante restas sucesivas, a partir de un esquema del tipo:

```
r = a           # lo que queda (el resto)
while r >= b:   # mientras pueda comprar otro
    r = r - b   # lo compro y veo lo que queda
```

- Estudiar las instrucciones de la función (sin ejecutarla).
- Todavía sin ejecutar la función, hacemos una *prueba de escritorio*. Por ejemplo, si ingresamos  $a = 10$  y  $b = 3$ , podríamos hacer como se indica en el **cuadro 8.1**, donde indicamos los pasos sucesivos que se van realizando y los valores de las variables **a**, **b** y **r**. Podemos comprobar entonces que los valores de **a** y **b** al terminar son los valores originales, mientras que **r** se modifica varias veces.

✎ Podés hacer la prueba de escritorio como te parezca más clara. La presentada es sólo una posibilidad.

✎ Las pruebas de escritorio sirven para entender el comportamiento de un conjunto de instrucciones y detectar algunos

Paso	acción	a	b	r
0	(antes de empezar)	10	3	sin valor
1	$r = a$			10
2	$r \geq b$ : verdadero			
3	$r = r - b$			7
4	$r \geq b$ : verdadero			
5	$r = r - b$			4
6	$r \geq b$ : verdadero			
7	$r = r - b$			1
8	$r \geq b$ : falso			
9	imprimir $r$			

Cuadro 8.1: Prueba de escritorio para el [ejercicio 8.6](#).

errores (pero no todos) cuando la lógica no es ni demasiado sencilla, como la de los ejemplos que hemos visto hasta ahora, ni demasiado complicada como varios de los ejemplos que veremos más adelante.

Otra forma —algo más primitiva— de entender el comportamiento y eventualmente encontrar errores, es probarlo con distintas entradas, como hemos hecho hasta ahora.

En fin, también es útil usar el menú *Debug* → *Debugger*, con el que podemos ver cómo van cambiando las variables según se van ejecutando las sentencias (usando *Step*), pero no veremos esta técnica en el curso.

- Hacer una prueba de escritorio con otros valores de  $a$  y  $b$ , por ejemplo con  $0 < a < b$  (en cuyo caso la instrucción dentro del lazo **while** no se realiza).
- Ejecutar la función, verificando que coinciden los resultados de la función y de las pruebas de escritorio.
- Observar que es importante que  $a$  y  $b$  sean positivos: dar ejemplos de  $a$  y  $b$  donde el lazo **while** no termina nunca (¡sin ejecutar la función!).



- f) Modificar la función para comparar el valor obtenido con el resultado de la operación  $a \% b$ .
- g)  $a$  es local a la función. ¿Habría algún problema en eliminar  $r$  y dejar sencillamente

```
while a >= b:
    a = a - b
```

en la función?

- h) Vamos a modificar la función de modo de contar el número de veces que se realiza el lazo **while**. Para ello agregamos un contador  $k$  que *antes* del lazo **while** se inicializa a 0 poniendo  $k = 0$  y *dentro* del lazo se incrementa en 1 con  $k = k + 1$ . Hacer estas modificaciones imprimiendo el valor final de  $k$  antes de finalizar la función.
- i) Modificar la función para calcular también el cociente de  $a$  por  $b$ , digamos  $c$ , mediante  $c = a // b$ . ¿Qué diferencia hay entre el cociente y el valor final de  $k$  obtenido en h)?, ¿podrías explicar por qué?

**E 8.7 (algoritmo de la división).** Como ya mencionamos en el [ejercicio 5.5](#), cuando  $a$  y  $b$  son enteros,  $b > 0$ , el *algoritmo de la división* encuentra enteros  $q$  y  $r$ ,  $0 \leq r < b$ , tales que  $a = qb + r$ .

- a) Definir una función **algodiv** para encontrar e *imprimir*  $q$  y  $r$  dados  $a$  y  $b$ ,  $a \geq 0$  y  $b > 0$ , usando sólo restas sucesivas.

*Sugerencia:*

```
q = 0           # cantidad de restas hechas
r = a           # lo que queda inicialmente
while r >= b:   # mientras pueda restar b
    r = r - b   # lo resto
    q = q + 1   # haciendo una resta más
```

☞ Si se usa la sugerencia, observar que en cada paso mantene-  
mos el invariante  $a = qb + r$  pues  $a = 0b + a = 1b + (a - b) =$   
...

- b) Extender la función anterior para considerar también el caso en que  $a$  sea negativo (siempre con  $0 \leq r < b$  y usando sólo sumas y restas).

*Sugerencia:*

```
if a >= 0:
    while r >= b:
        ...
else:    # a < 0
    while r < 0:
        ...
```



**E 8.8 (cifras III).** En los ejercicios 3.23 y 5.6 (y 7.9) vimos distintas posibilidades para encontrar las cifras de un número entero positivo. Una tercer posibilidad es ir dividiendo sucesivamente por 10 hasta llegar a 0 (que suponemos tiene 1 cifra), contando las divisiones hechas, imitando lo hecho en el ejercicio 8.6 sólo que dividimos en vez de restar.

Informalmente pondríamos:

$c \leftarrow 0$

**repetir:**

$c \leftarrow c + 1$

$n \leftarrow n // 10$

**hasta que**  $n = 0$

- El esquema anterior está escrito en *seudocódigo*: una manera informal para escribir algoritmos. Las operaciones se denotan como en matemáticas (y no como Python), de modo que « $=$ » es la igualdad mientras que « $\leftarrow$ » es la asignación, aunque nos tomamos la libertad de indicar con « $//$ » la división entera y poner comentarios como en Python.

Python no tiene la estructura **repetir... hasta que...**, pero podemos imitarla usando **break** en un «lazo infinito»:

```
c = 0
while True:    # repetir...
    c = c + 1
```

```

n = n // 10
if n == 0:      # ... hasta que n es 0
    break

```

Es decir: con **break** (o **return** si estamos en una función) podemos interrumpir un lazo.



*Hay que tener cuidado cuando **break** está contenido dentro de lazos anidados (unos dentro de otros), pues sólo sale del lazo más interno que lo contiene (si hay un único lazo que lo contiene no hay problemas).*

La función **cifras** (en el módulo **ifwhile**) usa esta estructura para calcular la cantidad de cifras en base 10 de un número entero, sin tener en cuenta el signo, pero considerando que 0 tiene una cifra.

- Estudiar las instrucciones de la función, y probarla con distintas entradas enteras (positivas, negativas o nulas).
- ¿Qué pasa si se elimina el renglón **n = abs(n)**?
- ¿Habría alguna diferencia si se cambia el lazo principal por

```

while n > 0:
    c = c + 1
    n = n // 10      ?

```

- Ver que los resultados de los ejercicios 3.23 y 5.6 coinciden con el obtenido al usar **cifras**.
- Con **return** salimos inmediatamente de la función: ver que cambiando **break** por **return c** (y comentando la aparición final de **return c**), el comportamiento de la función no varía.
- break** tiene como compañero a **continue**, que en vez de salir del lazo inmediatamente, saltea lo que resta y vuelve nuevamente al comienzo del lazo.

Posiblemente no tengamos oportunidad de usar **continue** en el curso, pero su uso está permitido (así como el de **break**).



## 8.3. El algoritmo de Euclides

En esta sección vemos uno de los resultados más antiguos que lleva el nombre de algoritmo.

Dados  $a, b \in \mathbb{N}$ , el *máximo común divisor entre  $a$  y  $b$* , indicado con  $\text{mcd}(a, b)$ , se define<sup>(1)</sup> como el máximo elemento del conjunto de divisores comunes de  $a$  y  $b$ :

$$\text{mcd}(a, b) = \max \{d \in \mathbb{Z} : d \mid a \text{ y } d \mid b\}.$$

- ✎ Para  $a$  y  $b$  enteros, la notación  $a \mid b$  significa  *$a$  divide a  $b$* , es decir, existe  $c \in \mathbb{Z}$  tal que  $b = c \times a$ .
- ✎  $\{d \in \mathbb{Z} : d \mid a \text{ y } d \mid b\}$  no es vacío (pues contiene a 1) y está acotado superiormente (por  $\min\{a, b\}$ ), por lo que  $\text{mcd}(a, b)$  está bien definido.
- ✎ La denominación *máximo común divisor* es la de uso tradicional en matemáticas. Más recientemente en algunas escuelas se la ha cambiado a *máximo divisor común*.<sup>(2)</sup>

Para completar la definición para cualesquiera  $a, b \in \mathbb{Z}$ , definimos

$$\begin{aligned} \text{mcd}(a, b) &= \text{mcd}(|a|, |b|), \\ \text{mcd}(0, z) &= \text{mcd}(z, 0) = |z| \quad \text{para todo } z \in \mathbb{Z}. \end{aligned}$$

No tiene mucho sentido  $\text{mcd}(0, 0)$ , y más que nada por comodidad, *definimos*  $\text{mcd}(0, 0) = 0$ , de modo que la relación anterior sigue valiendo aún para  $z = 0$ .

Cuando  $\text{mcd}(a, b) = 1$  es usual decir que los enteros  $a$  y  $b$  son *primos entre sí* o *coprimos* (pero  $a$  o  $b$  pueden no ser primos: 8 y 9 son coprimos pero ninguno es primo).

- ✎ Para nosotros, un número  $p$  es primo si  $p \in \mathbb{N}$ ,  $p > 1$ , y los únicos divisores de  $p$  son  $\pm 1$  y  $\pm p$ . Así, los primeros primos son 2, 3, 5, 7 y 11.

<sup>(1)</sup> ¡Como su nombre lo indica!

<sup>(2)</sup> Cuanto más se confunda, mejor.

En el libro VII de los Elementos, Euclides enuncia una forma de encontrar el máximo común divisor, lo que hoy llamamos *algoritmo de Euclides* y que en el lenguaje moderno puede leerse como:

*Para encontrar el máximo común divisor (lo que Euclides llama «máxima medida común») de dos números enteros positivos, debemos restar sucesivamente el menor del mayor hasta que los dos sean iguales.*

- ✎ En la escuela elemental a veces se enseña a calcular el máximo común divisor efectuando primeramente la descomposición como producto de primos. Sin embargo, la factorización en primos es computacionalmente difícil, y en general bastante menos eficiente que el algoritmo de Euclides, que aún después de 2000 años es el más indicado (con pocas variantes) para calcular el máximo común divisor.
- ✎ Un poco antes de Euclides con Pitágoras y el descubrimiento de la irracionalidad de  $\sqrt{2}$ , surgió el problema de la *commensurabilidad* de segmentos, es decir, si dados dos segmentos de longitudes  $a$  y  $b$  existe otro de longitud  $c$  tal que  $a$  y  $b$  son múltiplos enteros de  $c$ . En otras palabras,  $c$  es una «medida común». Si  $a$  es irracional (como  $\sqrt{2}$ ) y  $b = 1$ , entonces no existe  $c$ , y el algoritmo de Euclides no termina nunca.

**E 8.9 (algoritmo de Euclides I).** La versión original del algoritmo de Euclides para encontrar  $\text{mcd}(a, b)$  cuando  $a$  y  $b$  son enteros positivos podría ponerse como

**mientras**  $a \neq b$ :

**si**  $a > b$ :

$a \leftarrow a - b$

**en otro caso:**   # acá es  $b > a$  (8.1)

$b \leftarrow b - a$

    # acá es  $a = b$

**retornar**  $a$

- a) Construir una función `mcd` traduciendo a Python el esquema anterior (sólo para enteros positivos). Probarla con entradas positivas, e. g., `mcd(612, 456)` da como resultado 12.
- b) Ver que si algún argumento es nulo o negativo el algoritmo no termina (¡sin ejecutar la función!).
- c) Agregar instrucciones al principio para eliminar el caso en que alguno de los argumentos sea 0, y también para eliminar el caso en que algún argumento sea negativo.
- d) Modificar la función de modo que a la salida escriba también los valores originales de  $a$  y  $b$ , por ejemplo si las entradas son  $a = 12$  y  $b = 8$  que imprima

| El máximo común divisor entre 12 y 8 es 4



**E 8.10 (algoritmo de Euclides II).** Invirtiendo el proceso que hicimos en el [ejercicio 8.6](#), en la versión original del [esquema \(8.1\)](#) podemos cambiar las restas sucesivas por divisiones enteras y restos, como hacemos en la función `mcddr` en el módulo `ifwhile`.

- a) Verificar el funcionamiento tomando distintas entradas (positivas, negativas o nulas).
- b) En vista de que el algoritmo original puede no terminar dependiendo de los argumentos, ver que `mcddr` termina en un número *finito* de pasos, por ejemplo en no más de  $|b|$  pasos.

*Ayuda:* en cada paso, el resto es menor que el divisor.

- c) Modificar la función de modo que imprima la cantidad de veces que realizó el lazo `while`.
- d) ¿Qué pasa si se cambia la instrucción `while b != 0` por `while b > 0`?



**E 8.11.** Una de las primeras aplicaciones de `mcd` es «simplificar» números racionales, por ejemplo, escribir  $12/8$  como  $3/2$ . Definir una función que dados los enteros  $p$  y  $q$ , con  $q \neq 0$ , encuentre e imprima  $m \in \mathbb{Z}$  y  $n \in \mathbb{N}$  de modo que  $\frac{p}{q} = \frac{m}{n}$  y  $\text{mcd}(m, n) = 1$ .

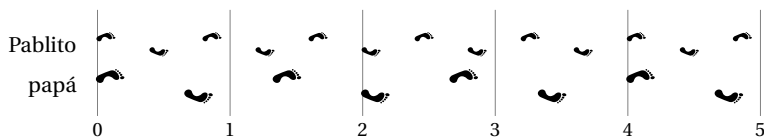


Figura 8.2: Pasos de Pablito y su papá.

⚠ ¡Atención con los signos de  $p$  y  $q$ !




**E 8.12.** El *mínimo común múltiplo* de  $a, b \in \mathbb{N}$ ,  $\text{mcm}(a, b)$ , se define en forma análoga al máximo común divisor: es el menor entero del conjunto  $\{k \in \mathbb{N} : a \mid k \text{ y } b \mid k\}$ .

- a) En la escuela nos enseñan que si  $a, b \in \mathbb{N}$  entonces

$$\text{mcm}(a, b) \times \text{mcd}(a, b) = a \times b.$$

Definir una función para calcular  $\text{mcm}(a, b)$  para  $a, b \in \mathbb{N}$ , usando esta relación.

- b) ¿Cómo podría extenderse la definición de  $\text{mcm}(a, b)$  para  $a, b \in \mathbb{Z}$ ? ¿Cuál sería el valor de  $\text{mcm}(0, z)$ ? 

**E 8.13 (Pablito y su papá I).** Pablito y su papá caminan juntos tomados de la mano. Pablito camina 2 metros en exactamente 5 pasos, mientras que su padre lo hace en exactamente 3 pasos.

- a) Resolver con lápiz y papel: si empiezan a caminar juntos, ¿cuántos metros recorrerán hasta marcar nuevamente el paso juntos?, ¿y si el padre caminara  $2\frac{1}{2}$  metros en 3 pasos?

*Aclaración:* se pregunta si habiendo en algún momento apoyado simultáneamente los pies izquierdos, cuántos metros después volverán a apoyarlos simultáneamente (ver [figura 8.2](#)).

*Respuesta:* 4 y 20 metros respectivamente.

- b) ¿Qué relación hay entre el máximo común divisor o el mínimo común múltiplo y el problema de Pablito y su papá?

- ✎ El ejercicio no requiere programación, lo que postergamos para el [ejercicio 8.14](#).
- ✎ Otros ejemplos «clásicos» para introducir el mínimo común múltiplo son:
- En un engranaje con dos ruedas de distinto tamaño se pinta el segmento que une los centros cuando las ruedas están quietas, y se quiere ver cuántas vueltas da cada una hasta que vuelva a verse el segmento como uno solo.
  - Los viajeros de comercio que recorren pueblos en tiempos distintos, y hay que ver cuándo vuelven a encontrarse en determinado pueblo.
  - Los semáforos que cambian de luces con determinadas frecuencias y hay que ver cuándo vuelven a ponerse del mismo color.
- ✎ Recordando el tema de la conmensurabilidad mencionado al introducir el algoritmo de Euclides, no siempre el problema tiene solución. Por ejemplo, si Pablito hace 1 metro cada 2 pasos y el papá  $\sqrt{2}$  metros cada 2 pasos.
- Como para la computadora todos los números son racionales, el problema siempre tiene solución computacional. ¶

**E 8.14 (Pablito y su papá II).** Definir una función para resolver en general el problema de Pablito y su papá ([ejercicio 8.13](#)), donde las entradas son el número de pasos y la cantidad de metros recorridos tanto para Pablito como para su papá.

Concretamente, los argumentos de la función son los enteros positivos  $p_b$ ,  $n_b$ ,  $d_b$ ,  $p_p$ ,  $n_p$  y  $d_p$ , donde:

- $p_b$  : número de pasos de Pablito  
 $n_b/d_b$  : metros recorridos por Pablito en  $p_b$  pasos  
 $p_p$  : número de pasos del papá  
 $n_p/d_p$  : metros recorridos por el papá en  $p_p$  pasos ¶





## Capítulo 9

# Sucesiones (secuencias)

Muchas veces necesitamos trabajar con varios objetos a la vez, por ejemplo cuando estamos estudiando una serie de datos. Una manera de agrupar objetos es mediante las *sucesiones* que estudiamos en este capítulo.

Las sucesiones de Python tienen varias similitudes con los conjuntos (finitos) de matemáticas. Así, podemos ver si cierto elemento está o no, agruparlos (como en la unión de conjuntos), encontrar la cantidad de elementos que tienen, e inclusive existe la noción de sucesión *vacía*, que no tiene elementos.

No obstante, las sucesiones no son exactamente como los conjuntos de matemáticas, ya que pueden tener elementos repetidos (y que cuentan para su longitud), y el orden es importante ('**nata**' no es lo mismo que '**tana**').

Justamente las cadenas de caracteres como '**nata**' —que hemos visto en el [capítulo 4](#)— son sucesiones.

Acá veremos tres nuevos tipos de sucesiones de Python: *tuplas* (**tuple**), *listas* (**list**) y *rangos* (**range**).

🔗 Python tiene la estructura **set** (*conjunto*) que no estudiaremos.

En la [sección 12.3](#) veremos cómo interpretar listas como conjuntos.

- Python tiene seis tipos de sucesiones: los cuatro ya mencionados (`str`, `list`, `tuple` y `range`), y `bytes` y `bytearray` que no veremos.

## 9.1. Índices y secciones

Las sucesiones comparten una serie de operaciones en común, como el cardinal o *longitud* —dado por `len`— que ya vimos para cadenas. Veamos otras dos: *índices* y *secciones*.

**E 9.1 (índices de sucesiones).** Si la sucesión `a` tiene  $n$  elementos, éstos pueden encontrarse individualmente con `a[i]`, donde  $i$  es un *índice*,  $i = 0, \dots, n - 1$ . Índices negativos cuentan desde el final hacia adelante ( $i = -1, \dots, -n$ ), y poniendo  $i$  fuera del rango  $[-n, n - 1]$  da error.

Resolver los siguientes apartados con `a = 'Mateo Adolfo'`.

a) Encontrar

i) `a[0]`    ii) `a[1]`    iii) `a[4]`    iv) `a[10]`    v) `a[-1]`

Para los próximos apartados suponemos que `n` es la longitud de `a`, i. e., que se ha hecho la asignación `n = len(a)`.


- b) Ver que `a[0]` y `a[n-1]` dan la primera y la última letras de `a`.  
c) Ver que `a[-1]` y `a[-n]` dan la última y la primera letras de `a`.  
d) Ver que `a[n]` y `a[-n-1]` dan error.  
e) Conjeturar el resultado de

i) `a[1.5]`    ii) `a[1.0]`



**E 9.2 (secciones de sucesiones).** Además de extraer un elemento con un índice, podemos extraer una parte o *sección* (*slice* en inglés, que también podría traducirse como *rebanada*) de una sucesión correspondiente a un rango de índices consecutivos (sin «saltos»).

Poniendo `a = 'Ana Luisa y Mateo Adolfo'` en la terminal, hacer los siguientes apartados.

- a) Ver qué hacen las siguientes instrucciones, verificando en cada caso que el valor de `a` no ha cambiado, y que el resultado es del mismo tipo que la sucesión original (`str` en este caso):
- i) `a[0:3]`    ii) `a[1:3]`    iii) `a[3:3]`    iv) `a[3:]`  
 v) `a[:3]`    vi) `a[:]`    vii) `a[-1:3]`    viii) `a[3:-1]`
- b) En base al apartado anterior, ¿podrías predecir el resultado de `a == a[:4] + a[4:]`? (Recordar el [ejercicio 4.16](#)).
- c) Es un error usar un único índice fuera de rango como vimos en el [ejercicio 9.1](#). Sin embargo, al seccionar en general no hay problemas, obteniendo eventualmente la cadena vacía `''`:
- i) `a[2:100]`    ii) `a[100:2]`  
 iii) `a[-100:2]`    iv) `a[-100:100]`
- d) ¿Qué índices habrá que poner para obtener `'Mateo'`?
- e) Encontrar `u`, `v`, `x` y `y` de modo que el resultado de `a[u:v] + a[x:y]` sea `'Ana y Mateo'`.
- f) Con algunas sucesiones podemos encontrar *secciones extendidas* con un tercer parámetro de *paso* o *incremento*. Ver qué hacen las instrucciones:
- i) `a[0:10:2]`    ii) `a[:10:2]`    iii) `a[-10::2]`  
 iv) `a[:,2]`    v) `a[:,:-1]` 

## 9.2. tuple (tupla)


Los elementos de las cadenas de caracteres son todos del mismo tipo. En cambio, las *tuplas* son sucesiones cuyos elementos son objetos arbitrarios. Se indican separando los elementos con comas «`,`», y encerrando la tupla entre paréntesis (no siempre necesarios) «`( )` y «`<>`».

### E 9.3 (tuplas).


- a) Poner

```
a = (123, 'mi mamá', 456)
b = 123, 'mi mamá', 456
```

y resolver los siguientes apartados.

- i) Encontrar el valor, el tipo (con **type**) y la longitud (con **len**) de **a** y **b**, comprobando que son idénticos.
  - ii) ¿Cuál te parece que sería el resultado de **a[1]**?, ¿y de **a[20]**?
- b) Construir una tupla con un único elemento es un tanto peculiar.
- i) Poner **a = (5)** y verificar el valor y tipo de **a**.  
 No da una tupla para no confundir con los paréntesis usados al agrupar.
  - ii) Repetir para **a = (5,)**.
- c) Por el contrario, la *tupla vacía* (con longitud 0) es « **()** »:
- i) Poner **a = ()** y encontrar su valor, tipo y longitud.
  - ii) Comparar los resultados de **a = (),** y de **a = ,** con los del apartado b).
- d) Una de las ventajas de las tuplas es que podemos hacer asignaciones múltiples en un mismo renglón: verificar los valores de **a** y **b** después de poner

```
| a, b = 1, 'mi mamá'
```

 Desde ya que la cantidad de identificadores a la izquierda y objetos en la tupla a la derecha debe ser la misma (o un único identificador a la izquierda), obteniendo un error en otro caso.


- e) Del mismo modo, podemos preguntar simultáneamente por el valor de varios objetos. Por ejemplo:

```
| a = 1  
| b = 2  
| a, b
```



**E 9.4.** Algunas veces es conveniente tener más de un valor como resultado. Por ejemplo, en el algoritmo de la división (ejercicios 5.5

y 8.7) en ocasiones queremos conocer tanto el valor del cociente,  $q$ , como el del resto,  $r$ .

- a) Averiguar qué hace la función `divmod` y usarla para distintos valores.
- b) Con los valores de  $q$  y  $r$  como en el [ejercicio 5.5](#), construir una función que *retorne* la tupla  $(q, r)$  y luego comparar los resultados con los de `divmod` para diferentes entradas (positivas y negativas). 

**E 9.5.** Definamos la función `asigmult` como

```
def asigmult(x, y):  
    """Asignación simultánea de 2 objetos."""  
    a = x  
    b = y  
    return a, b
```

imitando la asignación múltiple de dos objetos. Ver que

```
>>> a, b = asigmult(1, 'mi mama')
```

da los mismos resultados que los del [apartado d\)](#) del [ejercicio 9.3](#).

Repetir con el [ejercicio 9.3.e](#). 

**E 9.6 (intercambio).** La asignación múltiple en tuplas nos permite hacer el *intercambio* de variables (*swap* en inglés), poniendo en una el valor de la otra y recíprocamente.

- a) Poner


```
a = 1  
b = 2
```

y verificar los valores de  $a$  y  $b$ .

- b) Poner ahora

```
a, b = b, a
```

y verificar nuevamente los valores de  $a$  y  $b$ .

 El efecto del intercambio se ilustra en la [figura 9.1](#).

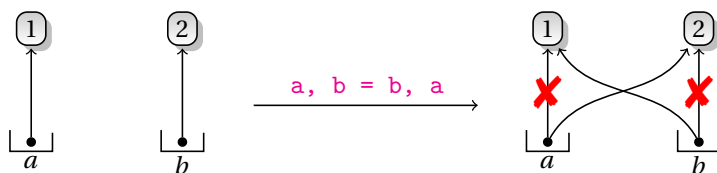


Figura 9.1: Efecto del intercambio de variables.

- c) También podemos hacer operaciones en combinación con el intercambio, como en

```
a, b = 1, 2
a, b = a + b, a - b
a, b
```

- d) En la función `mcddr` (en el módulo `ifwhile`) reemplazar los renglones

```
r = a % b
a = b
b = r
```

por el único renglón

```
a, b = b, a % b
```

¿Hay alguna diferencia en el comportamiento?

- e) Repetir los apartados anteriores usando la función `asigmult` del [ejercicio 9.5](#), viendo si hay diferencias en los comportamientos.



## 9.3. list (lista)

Nosotros usaremos tuplas muy poco. Las usaremos algunas veces para indicar coordenadas como en  $(1, -1)$ , para el intercambio mencionado en el [ejercicio 9.6](#), para hacer asignaciones múltiples como en el [ejercicio 9.3.d](#)), y ocasionalmente como argumentos o resultados de funciones como `divmod`.

Usaremos mucho más *listas*, similares a las tuplas, donde también los elementos se separan por « , » pero se encierran entre corchetes (ahora siempre) « [ » y « ] ». Como con las tuplas, los elementos de una lista pueden ser objetos de cualquier tipo, como en [1, [2, 3]] o [1, ['mi', (4, 5)]].

**E 9.7 (listas).** a) Poniendo en la terminal

```
| a = [123, 'mi mamá', 456]
```

i) Repetir el [ejercicio 9.3.a](#)).

ii) ¿Cómo se puede obtener el elemento 'mi mamá' de *a*?,  
¿y la 'i' en 'mi mamá'?

*Aclaración:* se piden expresiones en términos de *a* e índices.

b) A diferencia de las tuplas, la construcción de listas con un único elemento o de la *lista vacía* son lo que uno esperaría: verificar el valor, tipo y longitud de *a* cuando

i) *a* = []      ii) *a* = [5]      iii) *a* = [5,]

⇒ [5,] y (5,) son secuencias de longitud 1, lo mismo que [5], pero (5) no es una secuencia.

c) A veces podemos mezclar tuplas y listas. Ver los resultados de:

```
| a, b = [1, 2]
| [a, b] = 1, 2
```

d) Aunque hay que escribir un poco más (porque con las tuplas no tenemos que escribir paréntesis), también podemos hacer asignaciones múltiples e intercambios con listas como hicimos con tuplas (en los ejercicios [9.3](#) y [9.6](#)).

i) En la terminal poner [a, b] = [1, 2] y verificar los valores de *a* y *b*.

ii) Poner [a, b] = [b, a] y volver a verificar los valores de *a* y *b*.





**E 9.8 (mutables e inmutables).** Una diferencia esencial entre tuplas y listas es que podemos modificar los elementos de una lista, y no los de una tupla.

- Poner en la terminal `a = [1, 2, 3]` y verificar el valor de `a`.
- Poner `a[0] = 5` y volver a verificar el valor de `a`.
- Poner `b = a` y verificar el valor de `b`.
- Poner `a[0] = 4` y verificar los valores de `a` y `b`.
- Poner `a = [7, 8, 9]` y verificar los valores de `a` y `b`.

⇒ *Cambios de partes de `a` se reflejan en `b`, pero una nueva asignación a `a` no modifica `b`.*

- Repetir los apartados anteriores cambiando a tuplas en vez de listas, es decir, comenzando con

```
| a = 1, 2, 3
```

y ver en qué casos da error.



El ejercicio anterior muestra que podemos modificar los elementos de una lista (sin asignar la lista completa), y por eso decimos que las listas son *mutables*. En cambio, las tuplas son *inmutables*: sus valores no pueden modificarse y para cambiarlos hay que crear un nuevo objeto y hacer una nueva asignación.

✍ Dentro de ciertos límites, como vemos en el [ejercicio 9.15](#).

**E 9.9.** Los números y cadenas también son inmutables. Comprobar que

```
| a = 'mi mamá'
| a[0] = 'p'
```

da error.



Las listas permiten que se cambien individualmente sus elementos, y también que se agreguen o quiten.


**E 9.10 (operaciones con listas).** Pongamos



```
a = ['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a']
```

- a) Sin usar Python: ¿cuántos elementos tiene `a`?, ¿cuál es el valor de `a[3]`?
- b) En cada uno de los siguientes, después de cada operación verificar el valor resultante de `a` y su longitud con `len(a)`.

- |                                   |                                   |
|-----------------------------------|-----------------------------------|
| i) <code>a.append('b')</code>     | ii) <code>a.pop()</code>          |
| iii) <code>a.pop(0)</code>        | iv) <code>a.insert(3, 's')</code> |
| v) <code>a.insert(-1, 'x')</code> | vi) <code>a.reverse()</code>      |

✎ Con `help(list)` obtenemos información sobre éstas y otras operaciones de listas. Para algún método en particular podemos poner, por ejemplo, `help(list.pop)`. 

Operaciones como `append`, `insert` y `pop` se llaman *métodos*, en este caso de la *clase* `list`, y se comportan como funciones (anteponiendo el objeto de la clase correspondiente, como en `a.pop()`). Varios de estos métodos modifican la lista sobre la cual actúan, y a veces el valor que retornan es `None`.

En realidad, podemos modificar parte de una lista sin necesidad de usar `pop`, `append` o `insert`, sino sólo asignaciones a secciones, como ilustra el ejercicio siguiente.

### E 9.11. Ejecutar:

```
a = [1, 2, 3, 4, 5]
a[2:4]
a[2:4] = [6, 7, 8, 9]
a
a[1:-3]
a[1:-3] = [3, 4]
a
a[1:-1]
a[1:-1] = []
a
```



En particular, si `a[-1]` y `a[i]` existen, podemos decir que

<code>a.append(x)</code>	equivale a	<code>a[-1:] = [a[-1], x]</code>
<code>a.insert(i, x)</code>	equivale a	<code>a[i:i+1] = [x, a[i]]</code>
<code>a.pop(i)</code>	equivale a	$\begin{cases} x = a[i] \\ a[i:i+1] = [] \end{cases}$

aunque, como vimos, las asignaciones de secciones permiten operaciones más complejas ya que podemos cambiar más de un elemento.

🔗 En todos los casos `a.insert(i, x)` es equivalente a `a[i:i] = [x]`, aún en el caso donde `a` es la lista vacía y `i` es 0.

**E 9.12.** Si `a` es una lista (por ejemplo `a = [1, 2, 3]`), ¿cuáles son las diferencias entre `a.reverse()` y `a[::-1]`?

*Ayuda:* mirar el resultado de cada operación (lo que Python escribe en la terminal) y el contenido de `a` al final de cada una de ellas. 🗣



**E 9.13.** La mutabilidad hace que debamos ser cuidadosos cuando las listas son argumentos de funciones.

a) Por ejemplo, definamos

```
def f(a):
    """Agregar 1 a la lista a."""
    a.append(1)
```

Poniendo

```
a = [2]
b = a
f(a)
b
```

vemos que `a` y `b` se han modificado, y tomando ambos el valor `[2, 1]`.

b) Pero si hacemos una asignación a una lista *dentro* de una función, la lista pasa a ser una variable local a la función y no se modifica afuera.

```
def g(a):  
    """Asignar la lista."""  
    print("Al entrar a la función:", a)  
    a.append(2)  
    print("En el medio de la función:", a)  
    a = a + [3]  
    print("Antes de salir de la función:", a)
```

y luego evaluar

```
a = [1]  
g(a)  
a
```



**E 9.14.** Si queremos trabajar con una copia de la lista **a**, podemos poner **b = a[:]**.

a) Evaluar

```
a = [1, 2, 3]  
b = a  
c = a[:]  
a[0] = 4
```

y comprobar los valores de **a**, **b** y **c**.

b) Modificar la función **f(a)** del [ejercicio 9.13](#), de modo de *retornar* una copia de **a** a la que se le ha agregado 1, *sin modificar* la lista **a**.



**E 9.15.** En realidad la cosa no es tan sencilla como sugiere el [ejercicio 9.14](#). La asignación **a = b[:]** se llama una copia «playa» o «plana» (*shallow*), y está bien para listas que no contienen otras listas. Pero en cuanto hay otra lista como elemento las cosas se complican como en los ejercicios [9.8](#) y [9.13](#).

a) Por ejemplo:


```
a = [1, [2, 3]]  
b = a[:]
```

```
a[1][1] = 4
a
b
```

- b) El problema no es sólo cuando el contenedor más grande es una lista:

```
a = (1, [2, 3])
b = a
a[1][1] = 4      # ¡modificamos una tupla!
b
```

- ✎ El comportamiento en [a\)](#) es particularmente fastidioso cuando miramos a matrices como listas de listas, porque queremos una copia de la matriz donde no cambien las entradas.

En la jerga de Python queremos una copia «profunda» (*deep*). No vamos a necesitar este tipo de copias en lo que hacemos (o podemos arreglarnos sin ellas), así que no vamos a insistir con el tema. Los inquietos pueden ver la muy buena explicación en el [manual de la biblioteca](#) (*Data Types* → *copy*) y el módulo estándar *copy*. 


Dada la mutabilidad de las listas, debemos recordar:

*Cuando pasamos una lista (o parte de ella) como argumento de una función, hay que preguntarse siempre si al terminar la función la lista original*

- *debe,*
- *puede, o*
- *no debe*

*modificarse.*

El siguiente ejercicio muestra otro caso de mutabilidad aguda.

 **E 9.16 (problemas con asignaciones múltiples).** Si bien las asigna-

ciones múltiples e intercambios pueden ser convenientes (como en los ejercicios 9.3 y 9.6), debemos tener cuidado cuando hay listas involucradas porque son mutables.

- a) Conjeturar el valor de `x` después de las instrucciones

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
```

y luego verificar la conjetura con Python.

- b) Comparar con el comportamiento de la función `asigmult` del ejercicio 9.5. ¶

## 9.4. range (rango)

La última sucesión de Python que veremos es `range` o *rango*, una progresión aritmética de enteros. A diferencia de las cadenas de caracteres, las tuplas y las listas, `range` es una enumeración *virtual*, en el sentido de que sus elementos no ocupan lugar en la secuencia. Como las cadenas de caracteres y a diferencia de tuplas y listas, los elementos de `range` son siempre del mismo tipo: enteros.

`range` tiene entre uno y tres argumentos, que deben ser enteros y se usan en forma similar a las secciones en el ejercicio 9.2.

- Si hay tres argumentos el primero es donde comienza, el segundo es el siguiente al valor final, y el último se interpreta como el *paso* o *incremento* de la progresión. El incremento puede ser positivo o negativo, pero no nulo.
- Si tiene dos argumentos, se interpreta que el paso (el tercero que falta) es 1, y con un único argumento se interpreta que el primer elemento de la progresión es 0.
- Si el valor del incremento es positivo y el valor inicial es mayor o igual al final, el resultado (después de tomar `list`) es la lista vacía, `[]`.

Análogos resultados se obtienen cuando el incremento es negativo y el valor inicial es menor o igual al final.

**E 9.17 (range).** a) Usando `help`, encontrar qué hace `range`.

b) ¿Cuál es el resultado de `range(6)`?

c) Encontrar el tipo de `range(6)` (con `type`).

d) Ver los resultados de:

i) `list(range(6))`      ii) `len(range(6))`  
 iii) `list(range(0, 6))`      iv) `list(range(6, 6))`

e) Sin evaluar, conjeturar el valor y longitud de `list(range(15, 6, -3))`, y luego verificar la conjetura en Python.

f) Los rangos admiten operaciones comunes a las sucesiones, como índices, secciones o longitudes. Evaluar:

i) `list(range(7))`      ii) `range(7)[4]`  
 iii) `range(7)[1:5:2]`      iv) `list(range(7)[1:5:2])`  
 v) `len(range(7))`      vi) `len(range(7)[1:5:2])` ¶

## 9.5. Operaciones comunes

Vimos que índices y secciones son operaciones que se pueden realizar en todas las sucesiones, y también que la longitud se puede determinar con `len` en todos los casos. Veamos algunas más.

**E 9.18 (in).** Análogamente a la noción matemática de pertenencia en conjuntos, la instrucción `in` (literalmente *en*) de Python nos permite decidir si determinado elemento está en un contenedor.

a) `'a' in 'mi mama'`      b) `'b' in 'mi mama'`  
 c) `' ' in 'mi mama'`      d) `'_' in 'mi mama'`  
 e) `'a' in ''`      f) `' ' in ''`  
 g) `'_' in ''`      h) `1 in [1, 2, 3]`  
 i) `1 in [[1, 2], 3]`      j) `1 in range(5)`

⚠ A diferencia del comportamiento en otras sucesiones, `in` en cadenas también nos permite decidir si una cadena es parte de otra,

es decir, si es una *subcadena*. Nosotros no veremos este uso en el curso.



**E 9.19.** Definir una función **essecuencia** que determine si su argumento es o no una secuencia (cadena, tupla, lista o rango). Dar ejemplos donde esa función retorne **False**.

*Sugerencia:* repasar los ejercicios 4.9 y 9.18.



**E 9.20.** La concatenación que vimos en el **ejercicio 4.16** con «+», se extiende a tuplas y listas (los sumandos tienen que ser del mismo tipo). Ver el resultado de:

a) `[1, 2] + [3, 4]`

b) `(1, 2) + (3, 4)`

c) `[1, 2] + (3, 4)`



**E 9.21 (cambiando el tipo de sucesión).** Es posible cambiar el tipo de una sucesión a otra, dentro de ciertas restricciones.

a) Determinar el tipo de `a = [1, 2, 3]` y encontrar:

i) `str(a)`      ii) `tuple(a)`      iii) `list(a)`

b) Si `a = (1, 2, 3)`, ver que `tuple(list(a))` es `a`.

c) De la misma forma, ver que cuando `a = [1, 2, 3]`, entonces `list(tuple(a))` es `a`.

d) Determinar el tipo de `a = 'Ana Luisa'` y encontrar:


i) `str(a)`      ii) `tuple(a)`      iii) `list(a)`


iv) ¿Es cierto que `str(list(a))` es `a`?

✍ Si convertimos una cadena a lista (por ejemplo), podemos recuperar la cadena a partir de la lista de caracteres usando la función **sumar** que veremos en el **ejercicio 10.7**. Python tiene el método **join** que no veremos en el curso.



## 9.6. Comentarios

- La mutabilidad de las listas es un arma de doble filo. Por un lado al usarlas aumenta la rapidez de ejecución, pero por otro nos dan más de una sorpresa desagradable, como se puede apreciar con la aparición de numerosos símbolos  en la [sección 9.3](#).
- El concepto de mutabilidad e inmutabilidad no existe en lenguajes como C o Pascal. Estos lenguajes tienen el concepto de «pasar por valor o referencia», que tiene ciertas similitudes.
- La estructura de arreglo (listas con objetos del mismo tipo consecutivos en memoria) es de suma importancia para cálculos científicos de gran tamaño, y no está implementada en Python. Para nosotros será suficiente usar listas porque los ejemplos son de tamaño muy reducido.

 *numpy* (<http://numpy.scipy.org/>) es un módulo no estándar de Python que implementa arreglos eficientemente.

- Varios de los ejercicios están inspirados en los ejemplos del [tutorial](#) y del [manual de referencia](#) de Python.





## Capítulo 10

# Recorriendo sucesiones

Muchas veces tendremos que repetir una misma acción para cada elemento de una sucesión. Por ejemplo, para contar la cantidad de elementos en una lista la recorreríamos sumando un 1 por cada elemento. Algunas de estas acciones son tan comunes que Python tiene funciones predefinidas para esto, y para contar la cantidad de elementos usaríamos directamente `len`.

Sin embargo, a veces no hay funciones predefinidas o cuesta más encontrarlas que hacerlas directamente. Para estos casos, Python cuenta con la sentencia `for` (*para*) que estudiamos en este capítulo.

### 10.1. `for` (*para*)

Si queremos recorrer los elementos de una sucesión, para contarlos o para ver si alguno satisface cierta propiedad, intentaríamos algo como:

**hacer** algo con **cada** elemento de una sucesión.

Veremos un esquema similar cuando veamos listas por comprensión,<sup>(1)</sup> pero normalmente la sintaxis es un poco al revés:

---

<sup>(1)</sup> Concretamente, el [esquema \(10.14\)](#).

**para cada** elemento de la sucesión **hacer** algo con él.

Esta última versión se hace en Python usando **for**, en la forma

```
for x in iterable: # para cada x en iterable
    hacer_algo_con x
```

(10.1)

donde *iterable* puede ser una sucesión (cadena, tupla, lista o rango).

- ✎ Recordar que las secuencias tienen un orden que se puede obtener mirando los índices, como hicimos al principio del [capítulo 9](#).
- ✎ Las similitudes con el uso de **in** ([ejercicio 9.18](#)) son intencionales.
- ✎ Hay otros iterables (que no son sucesiones) en los que se puede usar **for**, como los archivos de texto que estudiamos en el [capítulo 11](#).

Miremos algunos ejemplos sencillos.

**E 10.1.** Consideremos **a = [3, 5, 7]**.

- a) Si queremos imprimir los elementos de **a** podemos poner

```
for x in a:      # para cada elemento de a
    print(x)     # imprimirlo (con el orden en a)
```

☞ *for toma los elementos en el orden en que aparecen en la secuencia.*

- b) Si queremos encontrar la cantidad de elementos, como en **len(a)**, imitando lo hecho en el [ejercicio 8.6.h](#)), llamamos **long** al contador y ponemos

```
long = 0
for x in a:
    long = long + 1 # la acción no depende de x
long
```

- c) Ver que las instrucciones en **a)** son equivalentes a:


```
for i in range(len(a)):
    print(a[i])
```

d) La construcción

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

es equivalente a

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

Comprobarlo cambiando el lazo **for** en **c)** por uno **while** (y definiendo adecuadamente *n*). 



*La similitud de las construcciones*

```
for x in a:      # para cada elemento
    ...          # hacer algo con x
```

y

```
for i in range(len(a)): # para cada índice
    ...                 # hacer algo con a[i]
```

*hace que muchas veces se confundan los elementos de una sucesión con los índices correspondientes: en general, el objeto  $x = a[i]$  de la sucesión **a** es distinto del índice **i**.*



**E 10.2.** La variable que aparece inmediatamente después de **for** puede tener cualquier nombre (identificador), pero hay que tener cuidado porque la variable no es local a la construcción.

Comprobar que el valor de **x** cambia en cada una de las siguientes:

a) **x = 5**    # **x** no está en la lista

```
for x in [1, 2, 3]:
    print(x)
```

```
x          # x no es 5
```

b) **x = 2**    # **x** está en la lista

```
b = [1, x, 3]
```

```

for x in b:
    print(x)
x          # x no es 2

```



**E 10.3.** En el [ejercicio 9.18](#) vimos que la construcción `x in a` nos dice si el objeto `x` está en la secuencia `a`. Podemos imitar esta acción recorriendo la sucesión `a` buscando el objeto `x`, terminando en cuanto lo encontremos:

```

t = False  # todavía no lo encontré
for y in a:
    if y == x:
        t = True    # lo encontré
        break      # no vale la pena seguir
t          # si lo encontré (True) o no (False)

```

(10.2)

- Comprobar la validez del esquema probándolo para los valores de `a` y `x` del [ejercicio 9.18](#).
- Dentro de una función podemos reemplazar `break` por `return` en el [esquema \(10.2\)](#), eliminando la variable `t`:

```

for y in a:
    if y == x:
        return True
return False

```

(10.3)

Definir una función `estaen(a, x)` que determina si `x` está en la sucesión `a` usando esta variante (agregando encabezado y documentación), y compararlo con los resultados de `x in a` para los valores de `a` y `x` del [ejercicio 9.18](#).



**E 10.4.** En ocasiones no sólo queremos saber si el objeto `x` está en la secuencia `a`, sino también conocer las posiciones (índices) que ocupa.


Definir una función `posiciones(a, x)` con el siguiente esquema (agregando encabezado y documentación):

```

p = []
for i in range(len(a)):
    if x == a[i]:
        p.append(i)
return p

```

(10.4)

- a) Probar la función para distintos valores de  $a$  y  $x$ .  
 ¿Qué pasa si  $a$  es la secuencia vacía?, ¿y si  $x$  no está en  $a$ ?
- b) ¿Cómo podría expresarse la instrucción `x in a` en términos de `posiciones(a, x)`? 

**E 10.5 (máximos y mínimos).** A veces queremos encontrar el máximo de una secuencia y su ubicación en ella.

- a) Definir una función `maxpos(a)` para encontrar el máximo de una secuencia `a` y la primera posición que ocupa siguiendo el esquema (agregar encabezado y documentación):

```

n = len(a)
if n == 0:
    print('*** secuencia vacía')
    return # retorna None y termina
xmax, imax = a[0], 0
for i in range(1, n):
    if a[i] > xmax:
        xmax, imax = a[i], i
return xmax, imax

```

- b) Ver los resultados de:

i) <code>maxpos([1, 2, 3])</code>	ii) <code>maxpos((1, 2, 3))</code>
iii) <code>maxpos(1, 2, 3)</code>	iv) <code>maxpos('mi mama')</code>
v) <code>maxpos([1, 'a'])</code>	vi) <code>maxpos([])</code>

- c) Python tiene la función `max`. Averiguar qué hace, repetir el apartado anterior (cambiando `maxpos` por `max`) y comparar los comportamientos.

d) Análogamente, considerar el caso del mínimo de una secuencia y compararla con la función `min` de Python.

Python tiene la función `enumerate` (en realidad una clase, como `int` o `list`) que combina los índices con los elementos de una sucesión. Nosotros no la veremos en el curso.

Veamos cómo podemos copiar la idea del contador `long` en el [ejercicio 10.1.b](#)) para sumar los elementos de una secuencia. Por ejemplo, para encontrar la suma de los elementos de

```
a = [1, 2.1, 3.5, -4.7]
```

con una calculadora, haríamos las sumas sucesivas

$$1 + 2.1 = 3.1, \quad 3.1 + 3.5 = 6.6, \quad 6.6 - 4.7 = 1.9. \quad (10.5)$$

Para repetir este esquema en programación es conveniente usar una variable, a veces llamada *acumulador* (en vez de contador), en la que se van guardando los resultados parciales, en este caso de la suma. Llamando `s` al acumulador, haríamos:

```
s = 1           # valor inicial de s
s = s + 2.1     # s -> 3.1
s = s + 3.5     # s -> 6.6
s = s + (-4.7)  # s -> 1.9
```

que es equivalente a las [ecuaciones \(10.5\)](#) y puede escribirse con un lazo `for` como:

```
s = 0           # acumulador, inicialmente en 0
for x in a:     # s será sucesivamente
    s = s + x    # 1, 3.1, 6.6 y 1.9
s
```

(10.6)

**E 10.6 (sumas y promedios).** Usando el [esquema \(10.6\)](#), definir una función `suma(a)` que dada la sucesión `a` encuentra su suma.

a) Evaluar `suma([1, 2.1, 3.5, -4.7])` para comprobar el comportamiento.

b) Ver el resultado de `suma(['Mateo', 'Adolfo'])`.

☞ *Da error porque no se pueden sumar un número (el valor inicial del acumulador) y una cadena.*

c) ¿Cuál es el resultado de `suma([])`?

☞ *Cuando la secuencia correspondiente es vacía, el lazo `for` no se ejecuta.*

d) Python tiene la función `sum`: averiguar qué hace esta función, y usarla en los apartados anteriores.

e) Usando `suma` y `len`, definir una función `promedio` que dada una lista de números construya su promedio. Por ejemplo, `promedio([1, 2.1, 3.5])` debería dar como resultado 2.2.

*Atención:* el promedio, aún de números enteros, en general es un número decimal. Por ejemplo, el promedio de 1 y 2 es 1.5.



Repasando lo que hicimos, en particular el esquema del [ejercicio 10.5](#), vemos que podríamos poner una única función para sumar números o cadenas de caracteres si ponemos el primer elemento de la sucesión como valor inicial del acumulador. Repitiendo lo hecho en ese ejercicio, arbitrariamente decidimos que cuando la sucesión es vacía, como `'` o `[]`, pondremos un cartel avisando de la situación y retornaremos `None`.

Tendríamos algo como:

```
if len(a) == 0:      # nada para sumar
    print('*** Atención: Sucesión vacía')
    return           # nos vamos
s = a[0]             # el primer elemento de a
for x in a[1:]:      # para c/u de los restantes...
    s = s + x
return s
```

**E 10.7.** Definir una nueva función **sumar** en base a estas ideas y comprobar su comportamiento en distintas sucesiones como:

- |                     |                              |
|---------------------|------------------------------|
| a) [1, 2, 3]        | b) 'Mateo'                   |
| c) [1, [2, 3], 4]   | d) ['Mateo']                 |
| e) [[1, 2], [3, 4]] | f) ['Mateo', 'Adolfo']       |
| g) []               | h) ['M', 'a', 't', 'e', 'o'] |
| i) ''               | j) (,)                       |
| k) range(6)         | l) range(6, 1)               |
| m) range(1, 6, 2)   |                              |



**E 10.8 (sumas de Gauss).** Calcular las sumas

$$s_n = 1 + 2 + \cdots + n = \sum_{k=1}^n k,$$

de dos formas:

- a) Mediante una función **gauss1(n)** que usa un lazo **for** con el esquema:

```
s = 0
for i in range(1, n + 1):
    s = s + i
return s
```

- b) Mediante una función **gauss2(n)** que usa la fórmula

$$s_n = \frac{n \times (n + 1)}{2}.$$

⚠ Usar división entera.



**E 10.9 (factorial).** Recordemos que para  $n \in \mathbb{N}$ , el factorial se define por

$$n! = 1 \times 2 \times \cdots \times n. \quad (10.7)$$

- a) Definir una función **factorial(n)** para calcular  $n!$  usando la ecuación (10.7) con el esquema:



```
f = 1 # acumulador para producto
for i in range(1, n + 1):
    f = f * i
```

(10.8)

✎ En realidad, tanto en la ecuación (10.7) como en el esquema (10.8), podríamos empezar multiplicando por 2 en vez de por 1. Empezamos desde 1 como regla mnemotécnica.

b) La fórmula de Stirling establece que cuando  $n$  es bastante grande,

$$n! \approx n^n e^{-n} \sqrt{2\pi n}.$$

Construir una función `stirling` para calcular esta aproximación, y probarla para  $n = 10, 100$  y  $1000$ , comparando los resultados con `factorial`. ¶

El teorema del binomio<sup>(2)</sup> expresa que si  $x$  y  $y$  son números,

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}, \quad (10.9)$$

donde

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k!} \quad (10.10)$$

son los *coeficientes binomiales* o *números combinatorios* ( $0 \leq k \leq n$ ).

✎ Cuando  $x = y = 1$ , (10.9) se reduce a  $2^n = \sum_{k=0}^n \binom{n}{k}$ , que tiene una bonita interpretación combinatoria: si el conjunto total tiene cardinal  $n$ , el miembro izquierdo es la cantidad total de subconjuntos mientras que  $\binom{n}{k}$  es la cantidad de subconjuntos con exactamente  $k$  elementos.

**E 10.10.** Hacer el cálculo de los tres factoriales ( $n!$ ,  $k!$  y  $(n-k)!$ ) en el segundo miembro de las igualdades en (10.10) es ineficiente, y es mejor hacer la división entera a la derecha de esas igualdades porque involucra muchas menos multiplicaciones.

<sup>(2)</sup> A veces llamado *teorema del binomio de Newton*.

- a) Volcar esta idea en una función para calcular el coeficiente binomial  $\binom{n}{k}$  donde los argumentos son  $n$  y  $k$  ( $0 \leq k \leq n$ ).
- b) Como  $\binom{n}{k} = \binom{n}{n-k}$ , si  $k > n/2$  realizaremos menos multiplicaciones evaluando  $\binom{n}{n-k}$  en vez de  $\binom{n}{k}$  usando el producto a la derecha de (10.10).

Agregar un **if** a la función del apartado anterior para usar las operaciones a la derecha de (10.10) sólo cuando  $k \leq n/2$ . ¶

**E 10.11 (sumas acumuladas).** En muchas aplicaciones —por ejemplo de estadística— necesitamos calcular las *sumas acumuladas* de una lista de números. Esto es, si la lista es  $a = (a_1, a_2, \dots, a_n)$ , la lista de acumuladas es la lista de sumas parciales,

$$s = (a_1, a_1 + a_2, a_1 + a_2 + a_3, \dots, a_1 + \dots + a_n),$$

o, poniendo  $s = (s_1, s_2, \dots, s_n)$ , más formalmente tenemos

$$s_k = \sum_{i=1}^k a_i, \quad k = 1, \dots, n. \quad (10.11)$$

✎ Ponemos las listas  $a$  y  $s$  como vectores para indicar que el orden es importante.

Esto se parece mucho a las ecuaciones (10.5) y podemos combinar los esquemas (10.6) y (10.15) para obtener el siguiente esquema, en el que suponemos que la lista original es **a**:

```
suma = 0
acums = []
for x in a:
    suma = suma + x
    acums.append(suma)
acums
```

(10.12)

- a) Definir una función **acumuladas** siguiendo el **esquema** (10.12), y aplicarla para calcular las acumuladas de los impares entre 1 y 19.



Una posibilidad usando listas por comprensión (que veremos en la [sección 10.3](#)) es construir las sumas

```
| [sum(a[:k]) for k in range(1, len(a))]
```

pero es terriblemente ineficiente porque volvemos a empezar cada vez.

b) A partir de (10.11), podemos poner

$$s_1 = a_1, \quad s_k = s_{k-1} + a_k \quad \text{para } k > 1,$$

y de aquí que

$$a_1 = s_1, \quad a_k = s_k - s_{k-1} \quad \text{para } k > 1. \quad (10.13)$$

Construir una función **desacumular** que dada la lista de números **b**, encuentre la lista **a** tal que **acumuladas(a)** sea **b**.

*Atención:* los índices en (10.11) y (10.13) empiezan con 1, pero Python pone los índices a partir de 0.

Quienes hayan estudiado derivadas e integrales podrán reconocer que «acumular» es como la integración y «desacumular» es como la derivación. Tal vez pueda verse mejor esta relación tomando « $\Delta x = 1$ » en (10.11) y (10.13)

Así, en el marco de matemáticas finitas es muy natural que un proceso sea el inverso del otro, lo que en matemáticas del continuo es el teorema fundamental del cálculo.



Vale la pena comparar el uso de acumuladores para la suma, el producto y otras operaciones binarias similares cuando se aplican a más de dos valores.

Por ejemplo:

- Para la suma de números ponemos inicialmente **s = 0** y en cada paso **s = s + x**.
- Para el producto de números ponemos inicialmente **p = 1** y en cada paso **p = p \* x**.
- Para ver si *todos* los valores lógicos de una lista son verdaderos, ponemos inicialmente **t = True** y en cada paso **t = x and t**.

En este caso, dado que una vez que **t** sea falso, siempre lo seguirá siendo, podemos poner (como en el [esquema \(10.2\)](#)):

```
t = True
for x in a:
    if not x:
        t = False
        break # salir del lazo
t
```

Consideraciones similares se aplican para ver si *alguno* de los valores lógicos de una lista es verdadero.

- Finalmente, como vimos en el [ejercicio 10.5](#) para el máximo y en el [ejercicio 10.7](#) para la suma (de números u otros objetos), cuando las secuencias a considerar son no vacías podemos poner a **secuencia[0]** como valor inicial del acumulador.

🔗 Acá vienen preguntas «filosóficas» de cuánto valen:

- la suma de ningún número,
- el producto de ningún número, o
- el máximo de una lista vacía.

Ninguna tiene mucho sentido.

Es decir, ponemos el acumulador inicialmente en 0 para la suma o en 1 para el producto sólo por una cuestión de simplicidad, pensando en que la lista con la que estamos trabajando es no vacía.

🔗 **E 10.12 (el río y los caballos).** Veamos algunos peligros que pueden surgir al cambiar de caballo en la mitad del río.

- a) Si cambiamos la secuencia sobre la que se itera dentro del lazo, podemos tener un lazo infinito (¡no ejecutar lo que sigue!):

```
b = [1]
for x in b:
    b.append(1)
```

Para verificar que el lazo no termina, paremos después de 10 pasos:

```

b = [1]
for x in b:
    b.append(1)
    if len(b) == 10:    # o 20,...
        break
b

```

- b) En este caso, trabajar con índices no es equivalente, pues `len(b)` queda fijo al comenzar el lazo: evaluar

```

b = [1]
for i in range(len(b)):
    b.append(1)
b

```

- c) Tratando de eliminar las apariciones de `x` en la lista `a` (modificándola), ponemos

```

def sacar(a, x):
    """Eliminar x de la lista a."""
    for i in range(len(a)):
        if a[i] == x:
            a.pop(i)

```

Probar la función cuando `a = [1, 2, 3, 4]` y

i) `x = 5`      ii) `x = 3`

¿Cuál es el problema? ¿Cómo se podría arreglar?

*Sugerencia:* recorrer la lista desde atrás hacia adelante. ¶

## 10.2. Un alto en el camino

En esta sección nos damos un respiro y vemos algunas aplicaciones.

**E 10.13.** Python tiene el método `reverse` que «da vuelta» una lista modificándola (ver el [ejercicio 9.10](#)). Esta acción se puede hacer eficientemente con un lazo `while`, tomando dos índices que comienzan en los extremos y se van juntando:


```

      *
    * *
  * * *
* * * *
* * * * *
```

Figura 10.1: «Triángulo de estrellas» con 5 estrellas en la parte inferior.



```

i, j = 0, len(a) - 1
while i < j:
    a[i], a[j] = a[j], a[i]      # intercambiar
    i, j = i + 1, j - 1         # nuevos i y j
```

- a) Definir una función **darvuelta** siguiendo este esquema y probarla con distintas listas.
- b) ¿Cómo se podría cambiar el lazo **while** por uno **for**? 

**E 10.14.** Construir una función **estrellas(n)** que imprima un «triángulo de estrellas» como se muestra en la [figura 10.1](#) para  $n = 5$ . El triángulo debe tener una estrella en la punta y  $n$  estrellas en la parte de abajo.

*Ayuda:* la cantidad de espacios en blanco iniciales disminuye a medida que bajamos y recordar la concatenación ([ejercicio 4.16](#)), la multiplicación de entero por cadena ([ejercicio 4.20](#)) y la «casita» del [ejercicio 4.23](#).

-  Como mencionamos en el [ejercicio 10.1.d](#)), podríamos usar tanto un lazo **for** como un lazo **while**. 


**E 10.15.** Construir una función **navidad(n)** que imprima un «arbolito de Navidad» como se muestra en la [figura 10.2](#) para  $n = 4$ . El árbol debe tener una estrella en la punta y  $n$  estrellas en la parte de abajo. 



Figura 10.2: «Arbolito de Navidad» con 4 estrellas en la parte inferior.

### 10.3. Listas por comprensión

Los conjuntos se pueden definir por *extensión*, explicitando cada uno de sus elementos como en  $A = \{1, 2, 3\}$ , o por *comprensión*, dando una propiedad característica como en  $B = \{n^2 : n \in A\}$ . En este ejemplo particular, también podemos poner en forma equivalente la definición por extensión  $B = \{1, 4, 9\}$ .

De modo similar, las listas también pueden definirse por extensión, poniendo explícitamente los elementos como en

```
| a = [1, 2, 3]
```

como hemos hecho hasta ahora, o *por comprensión* como en

```
| b = [n**2 for n in a]
```

y en este caso el valor de **b** resulta **[1, 4, 9]**: para cada elemento de **a**, en el orden dado en **a**, se ejecuta la operación indicada (aquí elevar al cuadrado).

Este mecanismo de Python se llama de *listas por comprensión*, y tiene la estructura general:

```
| [f(x) for x in iterable] (10.14)
```

donde **f** es una función definida para los elementos de *iterable*, y es equivalente al lazo **for**:

```

lista = [] # acumulador
for x in iterable:
    lista.append(f(x))
lista

```

(10.15)

Comparándolas, la [construcción \(10.14\)](#) no ahorra mucho en escritura respecto de [\(10.15\)](#), pero tal vez sí en claridad: en un renglón entendemos qué estamos haciendo.

Otras observaciones:

- Recordemos que, a diferencia de los conjuntos, las sucesiones pueden tener elementos repetidos y el orden es importante.
- El *iterable* en [\(10.14\)](#) o [\(10.15\)](#) tiene que estar definido, en caso contrario obtenemos un error.

**E 10.16.** a) Evaluar `[n**2 for n in a]` cuando

i) `a = [1, 2]`    ii) `a = [2, 1]`    iii) `a = [1, 2, 1]`

viendo que la operación `n**2` se realiza en cada elemento de `a`, respetando el orden.

b) Cambiando listas por tuplas en el apartado anterior, ver los resultados de:

i) `[n**2 for n in (1, 2, 1)]`

ii) `(n**2 for n in (1, 2, 1))`

☞ Veremos algo de generadores en el [capítulo 17](#).



**E 10.17.** Resolver los siguientes con `a = [1, 2, 3]`.

a) Conjeturar el valor y longitud de cada uno de los siguientes, y luego comprobar la conjetura con Python:

i) `b = [2 * x for x in a]`

ii) `c = [[x, x**2] for x in a]`

iii) `c = [(x, x**2) for x in a]`

b) En general, las tuplas pueden crearse sin encerrarlas entre paréntesis. ¿Qué pasa si ponemos `c = [x, x**2 for x in a]` en el apartado anterior?





**E 10.18.** En el [ejercicio 10.2](#) vimos que la variable `x` usada en `for` no es local al `for`. En cambio, las variables dentro de la definición por comprensión se comportan como variables locales.

Evaluar:

```
x = 0
a = [1, 2, 3]
b = [x + 1 for x in a]
x
```



**E 10.19.** Hay muchas formas de encontrar la suma de los  $n$  primeros impares positivos.

Una forma es generar la lista y luego sumarla:

```
impares = [2 * k + 1 for k in range(n)]
sum(impares)
```

o directamente `sum([2 * k + 1 for k in range(n)])`.

Python nos permite hacer esto en un único paso, eliminando la construcción de la lista:

```
sum(2 * k + 1 for k in range(n))
```

En fin, podemos seguir las ideas de los acumuladores (que es lo que hace en el fondo Python):

```
s = 0
for k in range(n):
    s = s + 2 * k + 1
s
```

- Probar los tres esquemas anteriores, viendo que dan los mismos resultados.
- Calcular las sumas para  $n = 1, 2, 3, 4$ . ¿Qué se observa? Conjeturar una fórmula para  $n$  general.  
*Ayuda:* la suma tiene que ver con  $n^2$ .
- ¿Podrías demostrar la fórmula?



**E 10.20.** Comparar las funciones `gauss1(n)` y `gauss2(n)` del [ejercicio 10.8](#) para  $n = 1, \dots, 10$ :

a) Construyendo dos listas y viendo si son iguales:

```
uno = [gauss1(n) for n in range(1, 11)]
dos = [gauss2(n) for n in range(1, 11)]
uno == dos
```

b) Usando un lazo `for`

```
t = True
for n in range(1, 11):
    if gauss1(n) != gauss2(n):
        t = False
        break
if not t:
    print('Falla para', n)
```

✎ La versión con listas parece más clara pero es menos eficiente, tanto en el uso de la memoria (cuando  $n$  es grande) como en que construimos las listas completas y luego las comparamos, siendo que es posible que difieran para valores chicos y no valga la pena construir las listas completas.


Ver también la [discusión sobre acumuladores](#) (página 114).



**E 10.21.** Queremos definir una función `sublista(a, pos)` que da una lista de los elementos de la lista (o secuencia) `a` que están en las posiciones `pos`. Por ejemplo

```
>>> sublista([5, 3, 3, 4, 1, 3, 2, 5], [2, 4, 6, 4])
[3, 1, 2, 1]
```

Dar dos definiciones distintas de la función `sublista`, una usando un lazo `for` como en el [esquema \(10.15\)](#) y la otra usando comprensión.

Si `indices = posiciones(lista, x)` (`posiciones` se definió en el [ejercicio 10.4](#)), ¿qué es `sublista(lista, indices)`? 

## 10.4. Filtros

Las listas por comprensión nos permiten fabricar listas a partir de sucesiones, pero podríamos tratar de construir una nueva lista sólo

con los elementos que satisfacen cierta condición, digamos  $p(x)$  (que debe tener valores lógicos).

Así, queremos una variante del **esquema (10.15)** de la forma:

```
lista = []
for x in iterable:
    if p(x):
        lista.append(f(x))
lista
```

(10.16)

Correspondientemente, el **esquema en (10.14)** se transforma en

```
[f(x) for x in iterable if p(x)]
```

(10.17)

y se llama *listas por comprensión con filtros*.

Por ejemplo,

```
[x for x in range(-5, 5) if x % 2 == 1]
```


da una lista de todos los impares entre  $-5$  y  $4$  (inclusivos), es decir,  $[-5, -3, -1, 1, 3]$ .

⚠ Mientras que  $[f(x) \text{ for } x \text{ in } \text{lista}]$  tiene la misma longitud de  $\text{lista}$ ,  $[f(x) \text{ for } x \text{ in } \text{lista} \text{ if } p(x)]$  puede tener longitud menor o inclusive ningún elemento.

**E 10.22.** Dando definiciones por comprensión con filtros, encontrar:

- Los números positivos en  $[1, -2, 3, -4]$ .
- Las palabras terminadas en «o» en

```
['Ana', 'Luisa', 'y', 'Mateo', 'Adolfo']
```

*Ayuda:* el índice correspondiente al último elemento de una sucesión es  $-1$ . 

**E 10.23.** Usando filtros (en vez de un lazo **for**), dar una definición alternativa de la función **posiciones** del **ejercicio 10.4**.

Comparar ambas versiones cuando  $a = \text{'mi mama me mima'}$  (sin tildes) y  $x$  es:

- $\text{'m'}$
- $\text{'i'}$
- $\text{'a'}$
- $\text{'A'}$



E 10.24. ¿Qué hace la función

```
def haceralgo(a, x):  
    return [y for y in a if y != x] ?
```



## 10.5. Comentarios

- **for** tiene distintos significados dependiendo del lenguaje de programación y no siempre existe. En general, **for** puede reemplazarse por un lazo **while** adecuado, como mencionamos en el [ejercicio 10.1.d](#).
- Las estructuras de listas por comprensión y de filtro no están disponibles en lenguajes como Pascal o C. Claro que el [esquema \(10.16\)](#) puede realizarse en estos lenguajes sin mayores problemas.
- En lenguajes que tienen elementos de *programación funcional*, la estructura `[f(x) for x in secuencia]` muchas veces se llama *map*, mientras que la estructura `[x for x in iterable if p(x)]` a veces se llama *filter* o *select*.

La estructura `[f(x) for x in iterable if p(x)]` generaliza *map* y *select*, y puede recuperarse a partir de ambas.

Python tiene las funciones **map** y **filter** con características similares a las descritas (ver `help(map)` o `help(filter)`). No veremos estas estructuras en el curso, usando en cambio listas por comprensión con filtros.

- Los lenguajes *declarativos* como SQL (usado en bases de datos) tienen filtros muy similares a los que vimos.



## Capítulo 11

# Formatos y archivos de texto

En esta capítulo vemos distintas formas de imprimir secuencias usando `for` y variantes de `print`, y cómo guardar información en archivos y luego recuperarla.

### 11.1. Formatos

Empezamos estudiando cómo imprimir secuencias en forma prolija, especialmente cuando se trata de encolumnar la información en tablas. Desde ya que aplicaciones más específicas, como procesadores de texto o de diseño de página, hacen esta tarea mucho más sencilla.

**E 11.1 (opciones de `print`).** Pongamos `a = 'Mateo Adolfo'` en la terminal.

- a) Ver el resultado de

```
for x in a:  
    print(x)
```

- b) `print` admite un argumento opcional de la forma `end=algo`. Por defecto, omitir esta opción es equivalente a poner `end='\n'`, iniciando un nuevo renglón después de cada instrucción `print`.  
Poner

```
for x in a:
    print(x, end='')
```

viendo el resultado.

- c) En algunos casos la impresión se junta con `>>>`, en cuyo caso se puede agregar una instrucción final `print('')`. Mejor aún es poner:

```
for x in a[::-1]:
    print(x, end='')
print(a[-1])
```

- d) `print` por defecto imprime separando los argumentos con espacios (' '), pero se puede cambiar usando la opción `sep`:

```
print(1, 2, 3)
print(1, 2, 3, sep=', ')
```



**E 11.2 (construcción de tablas).** Muchas veces queremos la información volcada en tablas.

- a) Un ejemplo sencillo sería construir los cuadrados de los primeros 10 números naturales. Podríamos intentar poniendo (en un módulo que se ejecutará)

```
for n in range(1, 11):
    print(n, n**2)
```

Probar este bloque, viendo cuáles son los inconvenientes.

- b) Uno de los problemas es que nos gustaría tener alineados los valores, y tal vez más espaciados. Se puede mejorar este aspecto con *formatos* para la impresión. Python tiene distintos mecanismos para *formatear*<sup>(1)</sup> y en el curso nos restringiremos al método `format` usando llaves (« { » y « } »).

Veamos cómo funciona con un ejemplo:

```
for n in range(1, 11):
    print('{0:2d} {1:4d}'.format(n, n**2))
```

<sup>(1)</sup> Sí, es una palabra permitida por la RAE.

El primer juego de llaves tiene dos entradas separadas por «:». La primera entrada, `0`, corresponde al índice de la primera entrada después de `format`. La segunda entrada del primer juego de llaves es `2d`, que significa que vamos a escribir un entero con 2 dígitos (`d`) y se alinean a derecha porque son números.

El segundo juego de llaves es similar.

- i) Lo que está encerrado entre comillas pero fuera de las llaves, son caracteres que se escriben «textualmente».<sup>(2)</sup> En el ejemplo, hay un espacio entre los dos juegos de llaves: modificarlos a ningún espacio o varios para ver el comportamiento.

También cambiar el espacio por algo como «mi mamá me mima».

✍ Si desean escribirse llaves en el texto intermedio deben ponerse dobles: `{{ o }}`.

- ii) Ver el efecto de cambiar, por ejemplo, `{1:4d}` a `{1:10d}`.  
 iii) Cambiar el orden poniendo `'{1:4d} {0:2d}'`, y ver el comportamiento.  
 iv) Poner ahora

```
for n in range(1, 11):
    print('{0:2d} {0:4d}'.format(n, n**2))
```

y comprobar que el segundo argumento (`n**2`) se ignora.

- c) Las cadenas de caracteres usan la especificación `s` (por *string*), que se supone por defecto:

```
print('{0} {0:s} {0:20s} {0}'.format(
    'Mateo Adolfo'))
```

A diferencia de los números que se alinean a la derecha, las cadenas se alinean a la izquierda, pero es posible modificar la alineación con «>» y «^». Por ejemplo

<sup>(2)</sup> ¡Bah!, tal cual.

```
print('-{0:20}-{1:^20}-{2:>20}-.format(
    'izquierda', 'centro', 'derecha'))
```

También se puede especificar < para alinear a la izquierda, pero en cadenas es redundante.

- d) Cuando trabajamos con decimales, las cosas se ponen un tanto más complicadas. Un formato razonable es el tipo **g**, que agrega una potencia de 10 con **e** si el número es bastante grande o bastante chico, pudiendo adecuar la cantidad de espacios ocupados después del « : » y la cantidad cifras significativas después del « . »:

```
a = 1234.5678901234
print('{0} {0:g} {0:15g} {0:15.8g}'.format(a))
print('{0} {0:g} {0:15g} {0:15.8g}'.format(1/a))
a = 1234567890.98765
print('{0} {0:g} {0:15g} {0:15.8g}'.format(a))
print('{0} {0:g} {0:15g} {0:15.8g}'.format(1/a))
```

- e) Se pueden poner las llaves sin nada adentro, en cuyo caso se usan los formatos por defecto correspondientes a los argumentos en el orden que aparecen. Ejecutar:

```
print(1, 2)
print('{} {}'.format(1, 2))
print(1, 2, sep=',')
print('{}{}'.format(1, 2))
print(1/3, 2/7)
print('{} {}'.format(1/3, 2/7))
```

- ✎ Nosotros no veremos más posibilidades de formatos. Las especificaciones completas de todos los formatos disponibles están en el *manual de la biblioteca de Python* (String Services). ¶

**E 11.3.** Imprimir una tabla del seno donde la primera columna sean los ángulos entre 0° y 45° en incrementos de 5 poniendo:

```
import math
```



```
aradianes = math.pi/180
for grados in range(0, 46, 5):
    print('{0:6}   {1:<15g}'.format(
        grados, math.sin(grados * aradianes)))
```

Probar estas instrucciones y agregar un encabezado con las palabras 'grados' y 'seno' centradas en las columnas correspondientes. ¶

**E 11.4.** Las instrucciones siguientes construyen la tabla de verdad para la conjunción  $\wedge$  («y» lógico):

```
formato = '{0:^5s}   {1:^5s}   {2:^5s}'
print(formato.format('p', 'q', 'p y q'))
print(19 * '-')
for p in [False, True]:
    for q in [False, True]:
        print(formato.format(
            str(p), str(q), str(p and q)))
```

- Modificarlas para obtener la tabla de verdad para la disyunción  $\vee$  (el «o» lógico).
- Modificarlas para que aparezcan las cuatro columnas  $p$ ,  $q$ ,  $p \wedge q$ ,  $p \vee q$ . ¶

## 11.2. Archivos de texto

Cuando se genera mucha información, como en las tablas, en general no queremos imprimirla en pantalla sino guardarla en un archivo en el disco de la computadora para su uso posterior, tal vez en otra computadora.

En este curso estudiaremos exclusivamente *archivos de texto*, que pueden leerse con un *editor de textos* como IDLE. En general, estos archivos tienen extensión *.txt*, pero podrían tener cualquier otra como *.dat* o *.sal*, o ninguna.

Otra alternativa es que el archivo sea *binario* (en vez de texto). Los *programas ejecutables* (muchas veces con terminación *.exe*) son

ejemplos de archivos binarios. Cuando estos archivos se abren con un editor de texto, aparecen «garabatos».

Tenemos dos tareas: *escribir* en archivos para guardar la información y *leer* esos archivos para recuperarla. En cualquiera de los dos casos el mecanismo es parecido:

- Relacionar el nombre del archivo en el disco de la computadora, llamémoslo *externo*, con un nombre que le daremos en el programa, llamémoslo *interno*, lo que se hace mediante el comando **open** (*abrir* en inglés).
  - ✎ Esto es parecido a lo que hicimos en el módulo *holapepe* (ejercicio 6.2): **pepe** es el nombre interno y el nombre externo puede ser '**Ana Luisa**' o cualquier otro.
- Informarle a Python de que queremos leer o escribir el archivo, lo que hacemos poniendo '**r**' para leer (por `read`, *leer*) o '**w**' para escribir (por `write`, *escribir*).
- Describir la codificación a usar en el archivo. Nosotros usaremos siempre utf-8, y lo indicamos por **encoding='utf-8'**.
  - ✎ Otra codificación usual es la ISO-8859-1 o Latin-1, para la que pondríamos **encoding='latin-1'**.
- Una vez concluida la tarea con el archivo, lo cerramos con **close**.

Como en la escuela, empecemos por la lectura.

**E 11.5 (lectura de archivos de texto).** *santosvega.txt* es un archivo de texto codificado en utf-8 con los primeros versos del poema *Santos Vega* de Rafael Obligado.

Nuestro objetivo es que Python lea el archivo y lo imprima en la terminal de IDLE.

- ✎ Es conveniente tener una copia del archivo en el mismo directorio/carpeta donde están nuestros módulos Python.
- a) Ubicándonos con IDLE en el directorio donde está el archivo *santosvega.txt*, ponemos en terminal:

```
archivo = open('santosvega.txt', 'r',  
               encoding='utf-8')
```

Observar que `archivo` es el identificador de una variable, el primer argumento de `open` es el nombre del archivo en la computadora (una cadena de caracteres), `'r'` indica que vamos a leer el archivo, y con `encoding='utf-8'` indicamos que está escrito con la codificación utf-8.

- ✎ No es necesario poner `'r'` para leer el archivo, pues se supone por defecto si se omite. La incluimos por simetría con la escritura.
- ✎ Es posible que la codificación por defecto sea utf-8 (y no sea necesario explicitarla) pero esto depende de la configuración del sistema operativo: MVPQC.<sup>(3)</sup>

- b) Averiguar el valor y el tipo de la variable `archivo`.
- c) Para leer efectivamente los contenidos, ponemos

```
archivo.read()
```

que nos mostrará el texto del archivo en una única cadena, incluyendo caracteres como `\n`, indicando el fin de renglón, y eventualmente algunos como `\t`, indicando tabulación (no en este caso).

☞ El método `read` pone todos los contenidos del archivo de texto en una única cadena de caracteres.

- d) Volver a repetir la instrucción `archivo.read()`, viendo que ahora se imprime sólo `''` (la cadena vacía), lo que indica que no quedan datos del archivo para leer.

- ✎ Se pueden dar instrucciones para ir al principio del archivo (o a cualquier otra posición) sin volver a abrirlo, pero no lo veremos en el curso.

- e) Cuando el archivo no se va a usar más, es conveniente liberar los recursos del sistema, *cerrando* el archivo.

---

<sup>(3)</sup> Más vale prevenir que curar.

Poner

```
archivo.close()    # cerrar el archivo
archivo.read()     # volver a leerlo
```

viendo que la última instrucción ahora da error.

- f) En este ejemplo no hay mucho problema en leer el renglón impreso mediante `archivo.read()` porque se trata de un texto relativamente corto. En textos más largos es conveniente imprimir (o lo que sea que queramos hacer) cada renglón separadamente, lo que podemos hacer poniendo

```
archivo = open('santosvega.txt', 'r',
               encoding='utf-8')
for renglon in archivo:
    print(renglon)
archivo.close() # ya no usamos más el archivo
```

✍ El método `readline` también nos permite leer un renglón. Nosotros no lo veremos.

- g) El resultado del apartado anterior es que se deja otro renglón en blanco después de cada renglón: en el texto original los renglones terminan con `'\n'`.

Podemos usar la técnica del [ejercicio 11.1](#) poniendo

```
archivo = open('santosvega.txt', 'r',
               encoding='utf-8')
for renglon in archivo:
    print(renglon, end='')
archivo.close()
```

Comprobar el resultado de estas instrucciones.

- h) A pesar del lazo `for` en el [apartado f\)](#), `archivo` no es una sucesión de Python. Probarlo poniendo

```
archivo = open('santosvega.txt', 'r',
               encoding='utf-8')
archivo[0]
```

```
len(archivo)
archivo.close()      # siempre cerrar al terminar
```

✎ No es una sucesión pero es un *iterable*, como lo son las sucesiones. Veremos algo más sobre el tema en el [ejercicio 11.11](#).

- i) Hacer que Python cuente la cantidad de renglones (incluyendo renglones en blanco), usando algo como

```
nrenglones = 0
for renglon in archivo:
    nrenglones = nrenglones + 1
print('El archivo tiene', nrenglones,
      'renglones')
```

✎ Es la misma construcción del [ejercicio 10.1.b](#).



**E 11.6.** En el módulo [dearchivoaterminal](#) hacemos una síntesis de lo que vimos anteriormente: preguntamos por el nombre del archivo de texto a imprimir en la terminal, abrimos el archivo, lo imprimimos y finalmente lo cerramos.

- a) Probarlo ingresando los nombres [santosvega.txt](#), [dearchivoaterminal.py](#) y [unilang8.txt](#) (un archivo con textos en distintos idiomas).
- b) Reemplazar o comentar el renglón donde se hace la asignación a `lectura`, cambiándolo por el siguiente en el que se elimina la parte de `encoding`:

```
lectura = open(entrada, 'r')      # abrirlo
```

y volver a «leer» los archivos en [a](#)).


¿Cuáles son las diferencias en el comportamiento?, ¿por qué?




El caso de la escritura es bastante similar, sólo que en vez de `read` para leer usaremos `write` para escribir.


**E 11.7 (escritura de archivos de texto).** El módulo `tablaseno` guarda una tabla del seno, la primera columna en grados y la segunda con el valor correspondiente del seno como hicimos en el [ejercicio 11.3](#): en vez de imprimir por pantalla, la tabla se guarda en un archivo.

Observamos que:

- La variable `archivo` es el nombre interno (dentro del módulo) que se relaciona con el nombre externo `tablaseno.txt` mediante la instrucción `open`.
- El archivo `tablaseno.txt` se guardará en el directorio de trabajo.  
 Recordar el [ejercicio 6.1](#) y las notas allí.
- `open` lleva como segundo argumento `'w'` (por write, *escribir*), indicando que vamos a escribir el archivo.





*Hay que tener cuidado pues si existe un archivo con el mismo nombre (y en el mismo directorio) éste será reemplazado por el nuevo.*

-  Hay formas de verificar si el archivo ya existe y en ese caso no borrarlo, pero no las veremos en el curso.

Por ejemplo, la opción `'x'` en vez de `'w'` hace que no se pueda escribir un archivo existente.

- `encoding='utf-8'` indica la codificación que usaremos (utf-8).
  - En el lazo `for` reemplazamos `print` por `archivo.write`, y en el texto a imprimir agregamos al final `\n`, que no poníamos con `print`.
  - Todo termina *cerrando* (`close`), el archivo que abrimos con `open`, con la instrucción `archivo.close()`.
- a) Ejecutar el módulo `tablaseno`, y abrir el archivo `tablaseno.txt` con un editor de texto (por ejemplo, el mismo IDLE) para verificar sus contenidos.

- b) ¿Qué pasa si se elimina `\n` en el argumento de `archivo.write`?
- c) Cambiar en todos los casos `archivo` por `salida` y verificar el comportamiento.
- d) Cambiar el nombre `tablaseno.txt` por `tabla.sal` y estudiar las diferencias.
- e) Cambiar el módulo de modo de preguntar interactivamente el nombre del archivo a escribir. 


**E 11.8.** Tomando partes de los módulos `dearchivoaterminal` y `tabla-seno`, construir una función `copiar(entrada, salida)` que copie el contenido del archivo de nombre `entrada` en el de nombre `salida` (ambos archivos en el disco de la computadora). 

**E 11.9 (split).** Abriendo el archivo `tablaseno` con IDLE vemos el formato de la tabla: cada renglón tiene dos números y varios espacios (`' '`). Para recuperar los números debemos eliminar los espacios, lo que podemos hacer con el método `split` para cadenas de caracteres.

- a) Ver qué hace el método `split` poniendo `help(str.split)`.
- b) Ver el resultado de:

```
a = 'mi_mamá_me.....imima!'
a
a.split()
a
```


donde en `a` ponemos arbitrariamente espacios entremedio.

 `a` no se modifica pues las cadenas son inmutables.

- c) Como el anterior, poniendo

```
a = 'mi\nmamá_me\nmima'
```

Ver también el resultado de `print(a)`, observando las diferencias.

- d) ¿Qué resultado da `''.split()` (sin espacios)?, ¿y con uno o más espacios como `' '.split()`? 

**E 11.10.** Usando `split`, imprimir por pantalla las palabras (una por renglón) con 5 o más letras en el archivo `santosvega.txt`. 

**E 11.11 (leyendo tablas).** En el [ejercicio 6.3](#) vimos que cuando se ingresan datos con `input`, Python los interpreta como cadenas de caracteres y debemos pasarlos a otro tipo si es necesario. Algo similar sucede cuando se lee un archivo de texto.

- a) Ubicándose en el directorio correspondiente, en la terminal poner

```
archivo = open('tablaseno.txt', 'r',  
               encoding='utf-8')
```

donde `tablaseno.txt` es el archivo construido en el [ejercicio 11.7](#).


- b) Usando `for` para construir una lista por comprensión (en forma similar al [ejercicio 11.5.f](#)), ver el resultado de las siguientes instrucciones:

```
a = [x for x in archivo] # lista de renglones  
a  
a[0]                     # primer renglón  
a[0].split()
```

- c) Teniendo una idea del comportamiento, podemos fabricar una lista con los números asociados a la tabla:

```
b = []  
for x in a:  
    g, s = x.split()  
    b.append([int(g), float(s)])  
b
```

- d) Con el esquema anterior construimos dos listas: `a` y `b`. Modificarlo de modo de construir la lista `b` directamente, sin construir `a`.

*Ayuda:* eliminar la construcción de la lista por comprensión usando un lazo `for` para recorrer los renglones (antes cerrar el archivo y volver a abrirlo). 



### 11.3. Comunicación con otras aplicaciones

La posibilidad de leer y escribir archivos de texto con Python nos brinda una interfaz —bastante elemental— para intercambiar información con otras aplicaciones.

A modo de ejemplo, en esta sección intercambiaremos datos entre Python y aplicaciones de planillas de cálculo (como MS-Excel y similares), usando el formato *csv* (*comma separated values*), de archivos de texto, en el que los renglones son las filas de la planilla y los datos de las columnas están separados por comas (y no por espacios) en el archivo. Normalmente, estos archivos tienen extensión *.csv*.

**E 11.12.** Empezamos por ver cómo podemos usar `split` para distinguir datos separados por comas.

- a) Ejecutar y estudiar el resultado

```
a = ['{},{}'.format(n, n**2) for n in range(10)]  
a    # ver los contenidos
```

- b) Usar `split` descomponiendo cadenas separadas por comas:

```
for x in a:  
    print(x.split(sep=', '))
```



**E 11.13.** En este ejercicio volcamos los datos de una planilla de cálculo a un archivo de texto que leeremos con Python.

- a) En una planilla de cálculo crear una tabla del seno: la columna de la izquierda representará ángulos entre  $0^\circ$  y  $90^\circ$  en incrementos de  $10^\circ$ , y la columna de la derecha tendrá el seno del ángulo correspondiente.
- b) Guardar la planilla como archivo de texto «delimitado por comas», con la extensión *.csv*.

✎ Estamos suponiendo que los números en la planilla tienen «punto» decimal y no «coma» decimal. Caso contrario habrá que hacer ajustes.

- c) Modificando adecuadamente el módulo [dearchivoaterminal](#), leer la tabla del archivo de texto imprimiéndola por pantalla con el formato del [ejercicio 11.3](#).

Ayuda: usar la opción `sep=', '` de `split`:

```
...  
for r in archivo:  
    g, s = r.split(sep=',')  
    ...
```

recordando que se leen cadenas que habrá que pasar a números. ¶

**E 11.14.** En este ejercicio construimos un archivo de texto con Python que leeremos con una planilla de cálculo.

- a) Modificando el módulo [tablaseno](#), guardar en un archivo de texto con extensión `csv` una la tabla del coseno para ángulos entre  $0^\circ$  y  $90^\circ$  en incrementos de  $10^\circ$ , donde en cada renglón los datos están separados por comas, usando un formato como el del [ejercicio 11.12.a](#)).

✍️ Recordar terminar cada renglón con `\n`.

- b) Abrir el archivo en una aplicación de planilla de cálculo. ¶

## 11.4. Comentarios

- Algunos entusiastas de Python prefieren el esquema

```
with open('datos.dat') as lectura:  
    for renglon in lectura:  
        print renglon
```

ya que el archivo se cierra aún si hubo errores en el medio. Nuestro esquema es (algo) más parecido al de otros lenguajes.



## Capítulo 12

# Clasificación y búsqueda

Siempre estamos buscando algo y es mucho más fácil encontrarlo si los datos están clasificados u ordenados, por ejemplo, una palabra en un diccionario. No es sorpresa que búsqueda y clasificación sean temas centrales en informática: el éxito de Google se basa en los algoritmos de clasificación y búsqueda que usa.

Es usual que en los cursos básicos de programación (como éste) se enseñen algoritmos elementales de clasificación, que en general necesitan del orden de  $n^2$  pasos para clasificar una lista de longitud  $n$ . En contraposición, el método de clasificación que usa Python es bastante sofisticado y rápido, necesitando del orden de  $n \times \log n$  pasos (entre comparaciones y asignaciones).

No vamos a tener tiempo en el curso para estudiar los métodos elementales como inserción directa, selección directa o intercambio directo (burbujeo).<sup>(1)</sup> De cualquier forma, vamos a ver el *método de conteo* en el [ejercicio 12.4](#) que extiende técnicas ya vistas, y es muy sencillo y rápido (usando del orden de  $n$  pasos) en ciertas circunstancias.

Análogamente, con [in](#) o los ejercicios [10.3](#) y [10.4](#) podemos buscar un objeto en una lista recorriéndola *linealmente* (mirando cada elemento en el orden en que aparece), pero podemos mejorar la eficiencia

---

<sup>(1)</sup> El interesado puede consultar el excelente libro de [Wirth \(1987\)](#).

si la lista está ordenada usando búsqueda binaria, lo que hacemos en la [sección 12.4](#).

Empezamos mencionando brevemente algunas funciones del módulo estándar *random* que usaremos para tener ejemplos de mayor tamaño.

## 12.1. El módulo *random*

Así como para disponer de funciones matemáticas como seno o logaritmo usamos el módulo *math*, para trabajar con números aleatorios en Python usamos el módulo *random* (*aleatorio* en inglés).

**E 12.1.** Como en otras ocasiones, restringiremos el uso de las funciones del *random* a sólo tres:

✎ Hay muchas más que no veremos: usar `help(random)`.

Por supuesto, arrancamos con

```
| import random
```

para incorporar las funciones del módulo.

- a) `random.random` es la función básica en *random*. Genera un decimal (`float`) aleatoria y uniformemente en el intervalo  $[0,1)$ . A partir de esta función se pueden generar casi todas las otras en el módulo *random*.

```
| random.random()  
| random.random()
```

- b) En muchos casos nos interesa trabajar con un rango de números enteros (no decimales). `random.randint` es especialmente útil, dando un entero entre los argumentos (inclusivos).

```
| random.randint(-10, 10)  
| [random.randint(-2, 3) for i in range(20)]
```

✎ Una función similar es `random.randrange`, que no veremos en el curso para no confundir. El nombre `randint` es común a otros lenguajes.

- c) `random.shuffle` (*shuffle* = barajar, mezclar) genera una permutación aleatoria «in situ» de una lista.

```
a = [2, 3, 5, 7, 11, 13, 17]
random.shuffle(a)          # retorna None
a
random.shuffle('mi mama')  # da error
random.shuffle((1, 5, 8))  # da error
random.shuffle(range(5))   # da error
```

- d) ¿Cómo podría obtenerse una permutación aleatoria de los caracteres en 'mi mama me mima', como 'mi mmmamamei a '?

*Sugerencia:* una posibilidad es pasar la cadena a lista de caracteres, usar `random.shuffle` y luego construir una cadena con la función `sumar` del [ejercicio 10.7](#). ¶

## 12.2. Clasificación

**E 12.2.** Para clasificar una lista (modificándola) podemos usar el *método* de listas `sort` (*clasificar* en inglés). Si no queremos modificar la lista o trabajamos con una cadena de caracteres o tupla —que no se pueden modificar— podemos usar la *función* `sorted` (*clasificada* en inglés) que da una nueva lista.

Verificar los resultados de los siguientes en la terminal de IDLE, donde usamos la función `sumar` del [ejercicio 10.7](#):

```
a) import random
   a = list(range(10))  # [0, 1, 2,...]
   random.shuffle(a)    # retorna None
   a                    # se modificó
   sorted(a)            # una lista
   a                    # no se modificó
```

```

a.sort()          # retorna None
a                 # se modificó
a = list(range(10))
random.shuffle(a)
a
a.sort(reverse=True) # clasificar al revés
a

b) a = 'Mi mamá me mima' # con mayúscula y tilde
a.sort()              # da error porque es inmutable
b = sorted(a)
b                     # da una lista
a                     # no se modificó
sumar(b)

```

☞ Es posible que la «M» (mayúscula) y la «á» (con tilde) no aparezcan en el lugar que esperaríamos. Para que lo hagan hay que cambiar algunas cosas que nos desviarían demasiado. Nuestra solución será sencillamente no usar tildes o diéresis al clasificar caracteres, y tener cuidado al trabajar con mayúsculas, por ejemplo con nombres propios.



Si **t** es una lista, podemos considerar **t.sort()** y **t.reverse()**, que modifican a **t**. Si **s** es una secuencia (no necesariamente mutable), **sorted(s)** da una lista (y **s** no se modifica), pero **reversed(s)** no es una lista sino un objeto del tipo «**reversed**», que no veremos en el curso.

**E 12.3 (key).** A veces queremos ordenar según algún criterio distinto al usual, y para eso —tanto en **sort** como en **sorted**— podemos usar la opción **key** (llave o clave).

**key** debe indicar una función que a cada objeto asigna un valor (usualmente un número, pero podría ser otro tipo como cadena), de modo que se puedan ordenar distintos objetos mediante estos valores.

- a) Por ejemplo, supongamos que tenemos la lista

```
a = [['Pedro', 7], ['Pablo', 6], ['Chucho', 7],  
      ['Jacinto', 6], ['José', 5]]
```

de nombres de chicos y sus edades.

Ordenando sin más, se clasifica primero por nombre y luego por edad:

```
sorted(a)
```

- b) Si queremos clasificar por edad, definimos la función que a cada objeto le asigna su segunda coordenada:

```
def edad(x):  
    return x[1]  
sorted(a, key=edad)
```

Observar que se mantiene el orden original entre los que tienen la misma edad: ['Pablo', 6] está antes de ['Jacinto', 6] pues está antes en la lista `a`.

Esta propiedad del método de clasificación se llama *estabilidad*.

- c) La estabilidad nos permite clasificar primero por una llave y luego por otra.

Por ejemplo, si queremos que aparezcan ordenados por edad, y para una misma edad por orden alfabético, podríamos poner:

```
b = sorted(a)          # orden alfabético primero  
sorted(b, key=edad)    # y después por edad
```

La última llave por la que se clasifica es la más importante. En el ejemplo, primero alfabéticamente (menos importante) y al final por edad (más importante).

- d) Consideremos una lista de nombres:

```
nombres = ['Ana', 'Gabi', 'Mari', 'Mateo',  
            'Geri', 'Guille', 'Maggie', 'Gero',  
            'Pablo', 'Chucho', 'Jacinto', 'José']
```

y llamemos `n` a la longitud de esta lista.

- i) Construir una lista **edades**, con **n** enteros aleatorios entre 5 y 16 (inclusivos).
- ii) Construir una lista **notas**, con **n** enteros aleatorios entre 1 y 10 (inclusivos).
- iii) Construir una lista **datos** con elementos de la forma **[x, y, z]**, con **x** en **nombres**, **y** en **edades** y **z** en **notas**, siguiendo el orden de las listas originales.
- iv) Clasificar **datos** de manera que los datos aparezcan ordenados por nota inversamente (de mayor a menor), luego por edad, y finalmente por nombre. Es decir, al terminar vendrán primero los que tienen mayores notas, si empatan la nota viene primero el de menor edad, y a una misma edad y nota, se ordenan alfabéticamente por nombre. ¶

**E 12.4 (clasificación por conteo).** Un método de clasificación sencillo y rápido es poner los objetos en «cajas» correspondientes.

Por ejemplo, supongamos que sabemos de antemano que los elementos de

$$a = [a[0], a[1], \dots, a[n-1]]$$

son enteros que satisfacen  $0 \leq a[i] < m$  para  $i = 1, 2, \dots, n$ , y  $m$  es relativamente pequeño.

Podemos imaginar que la lista que tenemos que clasificar son  $n$  bolitas alineadas, cada una con un número entre 0 y  $m - 1$ .

Por ejemplo, si

$$a = [0, 2, 2, 1, 0, 1, 0, 1, 0, 2]$$

las bolitas estarían alineadas como en la [figura 12.1](#) (arriba). Orientándonos con esa figura, consideramos  $m$  cajas numeradas de 0 a  $m - 1$ , colocamos cada bolita en la caja que tiene su número, y finalmente vaciamos los contenidos de las cajas ordenadamente, empezando desde la primera, alineando las bolitas a medida que las sacamos.

En el algoritmo, en vez de «colocar cada bolita en su caja», contamos las veces que apareció:



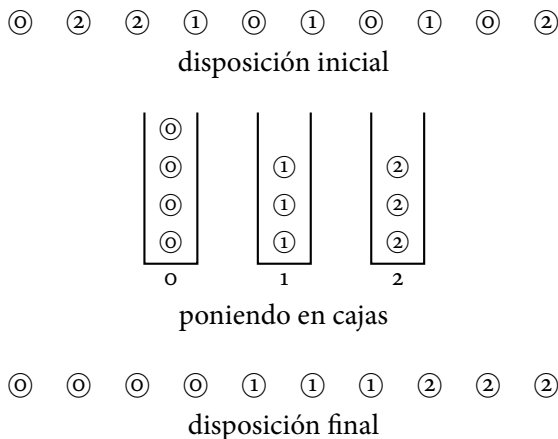


Figura 12.1: Ordenando por conteo.

```

# al principio, las cajas están vacías
cuenta = [0 for i in range(m)]

# ponemos cada bolita en su caja,
# aumentando el número de bolitas en esa caja
for k in a:      # k debe estar entre 0 y m-1
    cuenta[k] = cuenta[k] + 1

# ahora vaciamos las cajas, alineando las
# bolitas a medida que las sacamos
i = 0           # lugar en la línea
for k in range(m):      # para la caja 0, 1,..., m-1
    while cuenta[k] > 0:  # si tiene una bolita
        cuenta[k] = cuenta[k] - 1  # la sacamos
        a[i] = k           # la ponemos en la línea
        i = i + 1         # próximo lugar en la línea
  
```

a) Definir una función `conteo(a, m)` que clasifica la lista `a` con

este método (modificando la lista). Verificar el comportamiento para listas de 20 enteros aleatorios entre 0 y 4 (inclusivos).

Luego comparar con **sort** cuando **a** es una lista de 1000, 10 000 o más números enteros aleatorios entre 0 y 9, comprobando si hay diferencias apreciables en el tiempo de ejecución (trabajar con la misma lista, haciendo copias antes de clasificar).

b) ¿Cómo podríamos encontrar **m** si no se conoce de antemano?

*Aclaración:* suponemos dada una lista con números enteros no negativos.

- ✎ Se puede ver que el método usa del orden de  $n + m$  pasos, asignaciones y comparaciones, lo que lo hace más eficiente que otros algoritmos generales cuando  $m$  es chico, digamos menor que  $n$ .
- ✎ Otra forma de mirar la técnica es pensar que **cuenta[k]** no es otra cosa que la *frecuencia* con la que aparece el dato  $k$ .
- ✎ El método se generaliza al llamado *bucket sort* o *bin sort*,<sup>(2)</sup> y se usa en muchas situaciones. Por ejemplo, una forma de clasificar cartas (naipes) es ordenarlas primero según el «palo» (4 «cajas») y después ordenar cada palo.



## 12.3. Eliminando elementos repetidos de una lista

A veces es conveniente tratar a una sucesión como un conjunto, es decir, no nos interesan los elementos repetidos ni el orden. Python tiene la estructura **set** (*conjunto*) que permite tomar unión, intersección y otras operaciones entre conjuntos. **set** es un iterable (se puede usar con **for**), pero a diferencia de las sucesiones, sus elementos no están ordenados y no pueden indexarse. Nosotros no veremos esta estructura, pero en cambio consideraremos la eliminación de elementos repetidos de una lista, procedimiento que llamamos «purga».

**E 12.5 («purgar» una lista).** En este ejercicio vamos a eliminar los elementos repetidos de una lista pero conservando el orden original

---

<sup>(2)</sup> *Bucket*: balde o cubo, *bin*: caja, *sort*: clasificación.

entre los que sobreviven (algo como la estabilidad). Por ejemplo, si inicialmente `a = [1, 3, 1, 5, 4, 3, 5, 1, 4, 2, 5]`, queremos obtener la lista `[1, 3, 5, 4, 2]`.

- a) Si no queremos modificar `a`, un primer esquema es

```
b = []
for x in a:
    if x not in b:
        b.append(x)
b
```

Definir una función con estas ideas y probarla en el ejemplo dado.

- b) En base al esquema anterior, definir una función `purgar(a)` de modo que `a` se modifique adecuadamente sin usar una lista auxiliar.

*Aclaración:* no debe usarse una lista auxiliar pero elementos de `a` pueden cambiar de posición o ser eliminados.

*Sugerencia:* usar un índice para indicar las posiciones que sobrevivirán.

✎ *Sugerencia si la anterior no alcanza:*

```
m = 0          # a[0],...,a[m - 1]
                # son los elementos sin repetir
for x in a:
    if x not in a[:m]: # primera aparición,
        a[m] = x      # incorporarlo a lo
        m = m + 1     # que quedará
a[m:] = [] # considerar sólo a[0],..., a[m-1]
```

- c) ¿Qué problemas tendría usar el esquema

```
m = 1
for x in a[1:]: # no tocar el primer elemento
    if x not in a[:m]:
        a[m] = x
        m = m + 1
```



Construir una función **purgarordenado** que modifica su argumento siguiendo este esquema.

- ✎ La eficiencia se refiere a que el método del **ejercicio 12.5** en general hace del orden de  $n^2$  comparaciones, mientras que el presentado acá usa del orden de  $n$ .

**E 12.8 (de lista a conjunto).** Definir una función para «pasar una lista a conjunto», ordenándola y purgándola (en ese orden). Por ejemplo, si **a** = [3, 1, 4, 2, 3, 1], el resultado debe ser [1, 2, 3, 4] (la lista original **a** no debe modificarse).

- ✎ En general (no siempre) es más eficiente primero clasificar y después purgar (según el **ejercicio 12.7**) que purgar primero (según el **ejercicio 12.5**) y después clasificar.

**E 12.9 («unión» de listas).** En este ejercicio queremos imitar la unión de conjuntos, construyendo la «unión» de las listas **a** y **b** como la lista ordenada **c** con todos los elementos que están en **a** o **b** (pero aparecen en **c** una sola vez), y **a** y **b** no se modifican. Por ejemplo, si **a** = [3, 1, 4, 2, 3, 1] y **b** = [2, 3, 1, 5, 2], debe ser **c** = [1, 2, 3, 4, 5]. En particular, la «unión» de **a** con **a** son los elementos de **a** clasificados y sin repeticiones (pero sin modificar **a**).

Definir una función para el problema usando un lazo para recorrer simultáneamente **a** y **b** después de «pasarlas de lista a conjunto», generalizando el esquema de **purgarordenado**:

```
# acá:
# ap es la lista a clasificada y purgada,
# bp es la lista b clasificada y purgada
# con longitudes n y m respectivamente
i, j = 0, 0      # índices para ap y bp
c = []          # no hay elementos copiados
while (i < n) and (j < m):
    if ap[i] < bp[j]:
        c.append(ap[i]) # copiar en c
        i = i + 1       # y avanzar en ap
```

```

elif ap[i] > bp[j]:
    c.append(bp[j]) # copiar en c
    j = j + 1      # y avanzar en bp
else:
    c.append(ap[i]) # copiar en c
    i = i + 1      # y avanzar en
    j = j + 1      # ambas listas
c = c + ap[i:]     # copiar el resto de ap
c = c + bp[j:]     # copiar el resto de bp

```

- ✎ Este procedimiento de combinar listas ordenadas se llama «fusión» (*merge*).
- ✎ En general, el esquema propuesto es más eficiente que construir la lista  $c = a + b$  y luego clasificar y purgar (con `sort` y `purgarordenado` respectivamente) la lista  $c$ . ¶

**E 12.10 (estadísticos).** Cuando se estudian estadísticamente los datos numéricos  $(a_1, a_2, \dots, a_n)$ , se consideran (entre otros) tres tipos de «medidas»:


**La media o promedio:**  $\frac{1}{n} \sum_{i=1}^n a_i$ .

**La mediana:** Intuitivamente es un valor  $m$  tal que la mitad de los datos son menores o iguales que  $m$  y la otra mitad son mayores o iguales que  $m$ . Para obtenerla, se ordenan los datos y la mediana es el elemento del medio si  $n$  es impar y es el promedio de los dos datos en el medio si hay un número par de datos.

- ✎ Observar que la mediana puede no ser un dato en el caso de  $n$  par.
- ✎ El algoritmo «find» de [Hoare \(1961\)](#) permite encontrar la mediana sin necesidad de clasificar completamente la lista.

**La moda:** Es un valor  $a_\ell$  con frecuencia máxima, y puede haber más de una moda.

Por ejemplo, si los datos son 8, 0, 4, 6, 7, 8, 3, 2, 7, 4, la media es 4.9, la mediana es 5, y las modas son 4, 7 y 8.

- a) Definir sendas funciones para encontrar la media, mediana y moda de listas de números enteros.
  - b) Aplicar las funciones del apartado anterior a listas de longitud 10, 20 y 30 de números enteros entre 0 y 9 generados aleatoriamente.
- ✎ Si sabemos que los números en las listas no tienen un rango demasiado grande, la moda y la mediana se pueden encontrar usando la técnica del [ejercicio 12.4](#). 

## 12.4. Búsqueda binaria

**E 12.11 (el regalo en las cajas).** Propongamos el siguiente juego:

*Se esconde un regalo en una de diez cajas alineadas de izquierda a derecha, y nos dan cuatro oportunidades para acertar. Después de cada intento nuestro, nos dicen si ganamos (terminando el juego) o si el regalo está hacia la derecha o izquierda.*

- a) Simular este juego en la computadora, donde una «caja» es un número de 1 (extrema izquierda) a 10 (extrema derecha):
  - la computadora elige aleatoriamente una ubicación (entre 10 posibles) para el regalo,
  - el usuario elige una posición (usar `input`),
  - la computadora responde si el regalo está en la caja elegida (y el usuario gana y el juego se termina), o si el regalo está a la derecha o a la izquierda de la posición propuesta por el usuario,
  - si después de cuatro oportunidades no acertó la ubicación, el usuario pierde y la computadora da el número de caja donde estaba el regalo.

- b) Ver que siempre se puede ganar con la estrategia de *búsqueda binaria*: elegir siempre el punto medio del rango posible.
- c) Ver que no siempre se puede ganar si sólo se dan tres oportunidades.
- d) ¿Cuántas oportunidades habrá que dar para  $n$  cajas, suponiendo una estrategia de búsqueda binaria?

*Ayuda:* la respuesta involucra  $\log_2$ .

- e) Repetir el apartado d) (para calcular la cantidad de oportunidades) y luego el apartado a) para la versión donde en vez de tenerlas alineadas, las cajas forman un tablero de  $m \times n$ , y la búsqueda se orienta dando las direcciones «norte», «noreste»,..., «sur»,..., «noroeste».



**E 12.12.** Se ha roto un cable maestro de electricidad en algún punto de su recorrido subterráneo de 50 cuadras. La compañía local de electricidad puede hacer un pozo en cualquier lugar para comprobar si hasta allí el cable está sano, y bastará con detectar el lugar de la falla con una precisión de 5m.

Por supuesto, una posibilidad es ir haciendo pozos cada 5m, pero el encargado no está muy entusiasmado con la idea de hacer tantos pozos, porque hacer (y después tapar) los pozos cuesta tiempo y dinero, y los vecinos siempre se quejan por el tránsito, que no tienen luz, etc.

¿Qué le podrías sugerir al encargado?



Cuando la sucesión  $a$  está ordenada, la búsqueda de  $x$  en  $a$  se facilita enormemente, por ejemplo al buscar en un diccionario o en una tabla.

Uno de los métodos más eficientes para la búsqueda en una secuencia ordenada es la *búsqueda binaria*: sucesivamente dividir en dos y quedarse con una de las mitades, como hicimos en el [ejercicio 12.11](#).

**E 12.13 (búsqueda binaria).** Si queremos saber si un elemento  $x$  está en la lista  $a$  de longitud  $n$ , sin más información sobre la lista debemos inspeccionar todos los elementos de  $a$ , por ejemplo si  $x$  no está, lo



que lleva del orden de  $n$  pasos. La cosa es distinta si sabemos que  $a$  está ordenada no decrecientemente, es decir,  $a[0] \leq a[1] \leq \dots \leq a[n-1]$ .

La idea del método de búsqueda binaria es partir la lista en dos partes iguales (o casi) y ver de qué lado está el elemento buscado, y luego repetir el procedimiento, reduciendo la longitud de la lista en dos cada vez.

a) Un primer intento es:

```
poco = 0
mucho = n - 1
while poco < mucho:
    medio = (poco + mucho) // 2
    if a[medio] <= x:    # x está a la derecha
        poco = medio
    else:
        mucho = medio  # x está a la izquierda
# a continuación comparar x con a[mucho]
```

Ver que este esquema no es del todo correcto, y puede llevar a un lazo infinito.

*Sugerencia:* considerar  $a = [1, 3]$  y  $x = 2$ .

b) El problema con el lazo anterior es que cuando

$$mucho = poco + 1, \quad (12.1)$$

como  $medio = \lfloor (poco + mucho)/2 \rfloor$  obtenemos

$$medio = poco. \quad (12.2)$$

i) Verificar que (12.1) implica (12.2).

ii) Verificar que, en cambio, si  $mucho \geq poco + 2$  entonces siempre debe ser  $poco < medio < mucho$ .

c) Ver que el siguiente esquema es correcto:

```

poco = 0
mucho = n - 1
while poco + 1 < mucho:
    medio = (poco + mucho) // 2
    if a[medio] <= x:
        poco = medio
    else:
        mucho = medio
# ahora comparar x con a[mucho] y con a[poco]

```

y usarlo para definir una función `busbin(a, x)` para buscar un elemento en una lista (¡ordenada no decrecientemente!) retornando `False` o `True` y en este caso imprimiendo su posición.

- d) Probar `busbin(a, x)` cuando `a` es una lista ordenada de 100 números enteros elegidos al azar entre 1 y 1000 (posiblemente repetidos), en los siguientes casos:
- i) `x` es elegido aleatoriamente en el mismo rango pero puede no estar en la lista `a` (hacerlo varias veces).
  - ii) `x = 0`.
  - iii) `x = 1001`.
- e) ¿Cómo se relacionan el método de búsqueda binaria (en este ejercicio) y el del regalo en las cajas ([ejercicio 12.11](#))? Por ejemplo, ¿cuál sería la lista ordenada?
- 📖 Python tiene distintas variantes y aplicaciones de búsqueda binaria en el módulo estándar `bisect`, que no veremos en el curso (consultar el [manual de la biblioteca](#)). 📖

## 12.5. Comentarios

- Los interesados en el apasionante tema de clasificación y búsqueda pueden empezar mirando el libro de [Wirth \(1987\)](#) y luego profundizar con el de [Knuth \(1998\)](#). El libro de [Sedgewick y](#)

Wayne (2011) es de un nivel intermedio, pero está en inglés.

- La «fusión» del [ejercicio 12.9](#) da lugar a un método de clasificación que consiste en fusionar sublistas ordenadas cada vez más grandes, por ejemplo empezando con sublistas de tamaño uno. El algoritmo de clasificación que usa Python es una combinación de ese y otros métodos.
- Python usa funciones *hash*<sup>(3)</sup> para encontrar rápidamente elementos en una secuencia no ordenada, es decir, en general no usa ni un recorrido lineal ni búsqueda binaria.

Los tres libros mencionados anteriormente incluyen descripciones de esta técnica y algoritmos asociados.

Los curiosos pueden poner `help(hash)` y después probar con `hash('mama')`, `hash('mami')`, `hash(1)`, `hash(1.0)`, etc.

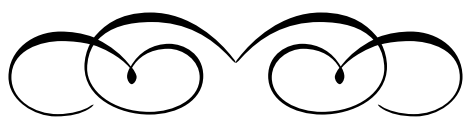


---

<sup>(3)</sup> A veces traducidas como *transformaciones de llave*.

## Parte II

### *Popurrí*



## Capítulo 13

# El largo y serpenteante recorrido

En este capítulo estudiamos una diversidad de problemas que pueden resolverse mediante variantes del recorrido de listas.

### 13.1. Números de Fibonacci

Los números de Fibonacci  $f_n$  se definen mediante:

$$f_1 = 1, \quad f_2 = 1, \quad \text{y} \quad f_n = f_{n-1} + f_{n-2} \quad \text{para } n \geq 3, \quad (13.1)$$

obteniendo la sucesión 1, 1, 2, 3, 5, 8, ...:

$$1, 1, 2 (= 1 + 1), 3 (= 2 + 1), 5 (= 3 + 2), 8 (= 5 + 3), \dots$$

En la [figura 13.1](#) ilustramos la construcción de  $f_n$ : en la columna de la izquierda está el valor de  $n$  y en las otras dos están los valores de  $f_n$  y  $f_{n-1}$ . Para calcular  $f_3$  necesitamos los valores de  $f_2$  y  $f_1$  lo que indicamos con flechas azules, y en ese momento  $f_{n-1} = f_2$  lo que indicamos con una flecha roja. Lo mismo sucede para valores mayores de  $n$ :  $f_n$  se calcula a partir de dos valores anteriores (flechas azules) y el nuevo  $f_{n-1}$  es el viejo  $f_n$  (flechas rojas).

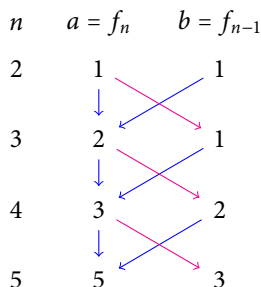


Figura 13.1: Ilustración del cálculo de los números de Fibonacci.

**E 13.1.** Para el cálculo de  $f_n$  con  $n \geq 3$  sólo necesitamos conocer los dos valores anteriores, y podemos usar un lazo **for** en el que conservamos los dos últimos encontrados. Para fijar ideas, llamamos  $a$  al valor de  $f_n$  y  $b$  al valor de  $f_{n-1}$ , como está indicado en la [figura 13.1](#).

Llegamos a algo como

```
a, b = 1, 1           # los dos primeros
for i in range(n - 2): # ojo que no es n
    a, b = a + b, a    # i no se usa
return a
```

🔍 Observar el uso de intercambio/asignaciones múltiples en la construcción **a, b = a + b, a**.

Definir una función para calcular  $f_n$  en base a estas ideas.



Los números de Fibonacci están íntimamente relacionados con  $\tau$  (pronunciado *tau*), el *número de oro*, también llamado *proporción áurea*,

$$\tau = \frac{1 + \sqrt{5}}{2} = 1.61803 \dots, \quad (13.2)$$

que aparece muchas veces en matemáticas, las artes y la arquitectura.

🔍 A veces se usa  $\varphi$  o  $\phi$  para designarlo (en vez de  $\tau$ ).

$\tau$  es la solución positiva de la ecuación

$$x^2 = x + 1,$$

y la otra solución es

$$\tau' = \frac{1 - \sqrt{5}}{2} = -\tau^{-1}.$$

De la teoría general de relaciones de recurrencia, se ve que

$$f_n = \frac{\tau^n - (\tau')^n}{\sqrt{5}} = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}, \quad (13.3)$$

relación comúnmente llamada de Binet o de Euler-Binet. En particular, esta relación muestra que  $f_n$  crece *exponencialmente*, ya que  $\tau > 1$ ,  $|\tau'| < 1$ , y  $f_n \approx \tau^n$  para  $n$  grande.

🔗 ¡El miembro derecho en (13.3) es un número entero!

### E 13.2 (fórmula de Euler-Binet).

- a) Definir una función **binet** para calcular  $f_n$  según (13.3).

*Sugerencia:* usar división común y no entera.

- b) Comparar los valores de **binet** con los valores obtenidos en el [ejercicio 13.1](#) para  $n = 1, \dots, 10$ .
- c) Comparar el  $n$ -ésimo número de Fibonacci  $f_n$  con el redondeo (**round**) de

$$\frac{\tau^n}{\sqrt{5}} = \frac{(1 + \sqrt{5})^n}{2^n \sqrt{5}}. \quad (13.4)$$

¿A partir de qué  $n$  son iguales?, ¿podrías decir por qué?

- d) Construir una función **binet2** para calcular  $f_n$  según (13.4).
- e) Calcular  $f_{345}$  de dos formas distintas: con la función definida en el [ejercicio 13.1.13.1](#) y con la función **binet2**.

¿Son iguales los valores encontrados?, ¿cuántos dígitos tiene cada uno?



✎ Las diferencias se deben a errores numéricos en el cálculo de la raíz cuadrada y otras operaciones con decimales. ¶

**E 13.3 (E. Zeckendorf, 1972).** Todo número entero se puede escribir, de forma única, como suma de (uno o más) números de Fibonacci no consecutivos, es decir, para todo  $n \in \mathbb{N}$  existe una única representación de la forma

$$n = b_2 f_2 + b_3 f_3 + \cdots + b_m f_m, \quad (13.5)$$

donde  $b_m = 1$ ,  $b_k \in \{0, 1\}$ , y  $b_k \times b_{k+1} = 0$  para  $k = 2, \dots, m-1$ .

Por ejemplo,  $10 = 8 + 2 = f_6 + f_3$ ,  $27 = 21 + 5 + 1 = f_8 + f_5 + f_2$ .

✎ En esta representación, tenemos que escribir  $f_1 = 1 = 1 \times f_2$ .

Definir una función para calcular esa representación (dando por ejemplo una lista de los índices  $k$  para los que  $b_k = 1$  en la **igualdad (13.5)**). Comprobar con algunos ejemplos la veracidad de la salida.

💡 *Atención: ¡no construir más de una vez cada número de Fibonacci!*

*Sugerencia:* en cambio, construir una lista con los números de Fibonacci que sean necesarios.

✎ Algo como

```
a, b = 1, 1
fibs = [1, 1] # len(fibs) = índice del último
while a <= n:
    a, b = a + b, a
    fibs.append(a)
# acá a > n: ya fabricamos todos los
# números de Fibonacci que necesitamos
...
```

¶

## 13.2. Sucesiones que se entrelazan

La **figura 13.1** nos da la idea que los números de Fibonacci se construyen a partir de listas que se entrelazan, pero se trata de, básicamente, la misma lista que se pone en ambos lados.

De cualquier modo, es muy común que tengamos que combinar dos o más sucesiones para fabricar una nueva sucesión o un número, como hicimos en el [ejercicio 12.3.d](#)), donde teníamos nombres, edades y notas que se combinaban para formar datos. En estos casos es conveniente trabajar directamente con índices, especialmente si las listas tienen distinta longitud, como hicimos en el [ejercicio 12.9](#).

En esta sección vemos algunos ejemplos más de esta situación.

### 13.2.1. Álgebra Lineal

**E 13.4.** Dados los vectores  $x$  y  $y$  de  $\mathbb{R}^n$ , su *producto interno* o *producto escalar* es el número

$$x \cdot y = \sum_{i=1}^n x_i y_i.$$

Definir una función `interno(x, y)` para calcular ese número, donde  $x = [x[0], x[1], \dots, x[n-1]]$  y análogamente para  $y$ .

¿Qué relación hay entre el producto interno y el [ejercicio 3.10](#)? ¶

Podemos una matriz de  $\mathbb{R}^{m \times n}$  como una lista de  $m$  listas que son las *filas* o *renglones*, y cada uno de éstas es una lista de  $n$  números.

Recordemos que la transpuesta de una matriz  $A = [a_{ij}] \in \mathbb{R}^{m \times n}$  es la matriz  $B = [b_{ij}] \in \mathbb{R}^{n \times m}$ , donde

$$b_{ij} = a_{ji} \quad \text{para } 1 \leq i \leq n, 1 \leq j \leq m.$$

**E 13.5.** Definir una función `transpuesta(matriz)` para encontrar la transpuesta de una matriz, expresando a las matrices como listas de listas.

Por ejemplo

```
>>> transpuesta([[1, 2], [3, 4], [5, 6]])
[[1, 3, 5], [2, 4, 6]]
```



### 13.2.2. Relojeando

Supongamos que tenemos un tiempo expresado en horas, minutos y segundos, por ejemplo, 12 hs 34 m 56.78 s y queremos pasarlo a segundos.

✎ En el [ejercicio 7.8](#) hicimos algo parecido.

Una forma de resolver el problema es pasar horas a segundos, minutos a segundos, y luego sumar todo:

$$\begin{array}{rcl}
 12 \text{ hs} & = & 12 \times 3600 \text{ s} = 43200 \text{ s} \\
 34 \text{ m} & = & 34 \times 60 \text{ s} = 2040 \text{ s} \\
 56.78 \text{ s} & = & 56.78 \text{ s} \\
 \hline
 \text{total} & = & 45296.78 \text{ s}
 \end{array} \tag{13.6}$$

Pero parece más sencillo primero pasar horas a minutos y luego minutos a segundos:

$$\begin{aligned}
 12 \text{ hs} &= 12 \times 60 \text{ m} = 720 \text{ m}, \\
 720 \text{ m} + 34 \text{ m} &= 754 \text{ m}, \\
 754 \text{ m} &= 754 \times 60 \text{ s} = 45240 \text{ s}, \\
 45240 \text{ s} + 56.78 \text{ s} &= 45296.78 \text{ s},
 \end{aligned}$$

que podríamos poner en un único renglón:

$$(12 \times 60 + 34) \times 60 + 56.78 = 45296.78.$$

En general, si tenemos  $a$  años,  $d$  días,  $h$  horas,  $m$  minutos y  $s$  segundos, para expresar todo en segundos pondríamos (suponiendo que todos los años tienen 365 días):

$$\begin{array}{ll}
 t = a * 365 + d & \# \text{ días} \\
 t = t * 24 + h & \# \text{ horas} \\
 t = t * 60 + m & \# \text{ minutos} \\
 t = t * 60 + s & \# \text{ segundos}
 \end{array}$$

Si los tiempos están en la lista `tiempo = [a, d, h, m, s]`, y las unidades de las conversiones en la lista `c = [365, 24, 60, 60]`, vemos que las listas no son de igual longitud, y podríamos poner un lazo como el siguiente:

```
n = len(tiempo)      # len(c) es n - 1
t = tiempo[0]
for i in range(1, n):
    t = t * c[i - 1] + tiempo[i]
t      # tiempo pasado a segundos
```

(13.7)

**E 13.6 (expresando el tiempo en segundos).** Tomando como base el [esquema \(13.7\)](#), definir una función `asegs(tiempo)` para pasar el tiempo expresado en años, días, horas, minutos y segundos a segundos.

*Aclaración:* suponemos `tiempo` de la forma `[a, d, h, m, s]`.

Comprobar los resultados cuando los argumentos son:

a) `[0, 0, 12, 34, 56.78]`

b) `[5, 67, 8, 9, 12.34]`



### 13.2.3. Plazos fijos y préstamos

Mucha gente tiene certificados bancarios a plazo fijo que va renovando periódicamente, y en cada vencimiento retira o deposita un dinero extra.<sup>(1)</sup>

Para calcular los sucesivos montos en el período  $n$ , consideramos:

- $c_{n-1}$ : el monto al principio del período,
- $r_{n-1}$ : tasa de interés durante el período (como porcentaje),
- $v_n$ : el monto al vencer el certificado (al final del período),
- $d_n$ : depósito que se agrega para el siguiente período.

$\mathbb{Z}$   $d_n$  es negativo cuando se retira en vez de depositar.

<sup>(1)</sup> En algunos lugares, como la provincia de Santa Fe, además se debe pagar un impuesto por cada certificado.

Entonces para  $n = 1, 2, \dots$ ,

$$v_n = c_{n-1} + c_{n-1} \times \frac{r_{n-1}}{100} = c_{n-1} \times \left(1 + \frac{r_{n-1}}{100}\right), \quad (13.8a)$$

donde hemos usado que *interés* = *capital*  $\times$  *tasa de interés*, y podemos poner

$$c_0 = d_0, \quad y \quad c_n = v_n + d_n \text{ para } n = 1, 2, \dots \quad (13.8b)$$

Las ecuaciones (13.8) nos llevan a un esquema similar al [esquema \(13.7\)](#):

```
c = d[0] # capital inicial = depósito inicial
for i in range(1, n + 1):
    c = c * (1 + r[i - 1]/100) + d[i]
c # monto del nuevo certificado
```

(13.9)

donde

- $r = [r[0], r[1], \dots]$  son las tasas de interés (en porciento) de los períodos, y
- $d = [d[0], d[1], \dots]$  son los depósitos periódicos ( $d[0]$  es el inicial y  $d[i]$  el que se hace al final del período  $i$  si  $i \geq 1$ ).

Observar que en el [esquema \(13.9\)](#),  $d$  tiene un elemento más que la lista  $r$ . Si lo que queremos es el monto que podríamos retirar del banco al vencer el certificado del período  $n$ , tenemos que cambiar el esquema:

**E 13.7.** Modificando adecuadamente el [esquema \(13.9\)](#), definir una función `vencimiento(n, d, r)` que nos diga el monto del certificado al vencimiento del período  $n$ , antes de hacer el depósito (o extracción) correspondiente a ese mes.

*Aclaración:* acá  $d$  y  $r$  deben ser listas de longitud al menos  $n$ , y no se debe usar  $d[n]$ .

Usar la función para calcular el monto al final del tercer período si:

- Maggie hizo un depósito inicial de \$ 10 000 a una tasa de 1.2 % para ese período,
- para el siguiente certificado agregó otros \$ 1 000 con una tasa del 0.95 %,
- antes de hacer el tercer certificado retiró \$ 2 500 e hizo el certificado a una tasa de 1.05 %.



Los préstamos bancarios siguen un mecanismo similar al de los certificados a plazo fijo que describimos anteriormente. A diferencia de los certificados que muchas veces se hacen por una cierta cantidad de días, los préstamos se hacen en general con pagos mensuales, y a veces las tasas y los pagos mensuales son variables.

Considerando:

- $r_{n-1}$ : tasa de interés para el mes  $n$ ,
- $p_{n-1}$ : pago realizado al fin del mes  $n$ ,
- $c_n$ : capital adeudado al fin del mes  $n$  (luego de haber pagado  $p_{n-1}$ ,  $n \geq 1$ ),

llegamos a una variante de las ecuaciones (13.8):

$$c_n = c_{n-1} \times \left(1 + \frac{r_{n-1}}{100}\right) - p_{n-1} \quad \text{para } n = 1, 2, \dots, \quad (13.10)$$

donde  $c_0$  es el monto del préstamo (la deuda inicial).

▮ Para simplificar, supondremos que no hay gastos de administración ni impuestos.

También es común que los pagos y las tasas sean fijas. En este caso es usual considerar una tasa anual *nominal*, llamémosla  $r$ , definiendo el factor

$$t = 1 + \frac{r}{12 \times 100} = 1 + \frac{r}{1200}. \quad (13.11)$$

Así, las ecuaciones (13.10) se convierten en

$$c_n = c_{n-1} \times t - p \quad \text{para } n \geq 1, \quad (13.12)$$

expresando la deuda al final del mes  $n$ , una vez hecho el  $n$ -ésimo pago.

**E 13.8.** Supongamos que tenemos un préstamo de  $c_0$  que pagamos en 12 cuotas mensuales fijas de  $p$ , y los montos adeudados mensualmente son los expresados en (13.12).

- a) Usando inducción, demostrar que la deuda al fin del mes  $n$  es:

$$c_n = \begin{cases} c_0 t^n - p \times \frac{t^n - 1}{t - 1} & \text{si } r \neq 0, \\ c_0 - np & \text{si } r = 0. \end{cases} \quad (13.13)$$

- b) Usando que cuando  $n = 12$  (al terminar de pagar la última cuota) el monto adeudado es nulo, expresar  $c_0$  (el monto del préstamo) en términos de  $p$  y  $r$ .

*Respuesta:*  $c_0 = \frac{p}{t^{12}} \times \frac{t^{12} - 1}{t - 1}$  cuando  $r \neq 0$ .

- c) Del mismo modo, expresar  $p$  en términos de  $c_0$  y  $r$ .

- d) En base a los apartados anteriores, tomando

- $c$  como el capital inicial (indicado anteriormente por  $c_0$ ),
- $r$  la tasa de interés nominal anual (en porcentaje),
- $p$  el pago mensual,

definir las funciones `prestamo12`, `inic12` y `pago12` tales que,

- `prestamo12(c, r, p)` retorna la deuda al fin del mes 12 según la expresión en el apartado a),
- `inic12(r, p)` da el capital inicial según el apartado b),
- `pago12(c, r)` da el valor de la cuota mensual de acuerdo al apartado c).



**E 13.9.** A Gabi le gustaría conseguir un préstamo de \$ 50 000 para hacer unos arreglos en la casa, a pagar en 12 cuotas mensuales fijas. Si el banco hace préstamos de esas características a una tasa fija nominal anual de 48 %:

- a) ¿Cuál sería el monto de los pagos mensuales (redondeado al centavo)?
- b) Gabi no quisiera pagar más de 4 000 mensuales, ¿cuál sería el

máximo monto del préstamo que le puede dar el banco (redondeado en unidades de \$ 100)?



### 13.3. Polinomios

Los polinomios son expresiones de la forma

$$\begin{aligned} P(x) &= \sum_{k=0}^n a_k x^k = a_0 + a_1 x + \cdots + a_n x^n \\ &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0, \end{aligned} \quad (13.14)$$

y son las funciones más sencillas que podemos considerar: para su cálculo sólo se necesitan sumas y productos (y no tenemos que hacer aproximaciones como para el seno o el logaritmo).

Además los usamos diariamente: un número en base 10 es en realidad un polinomio evaluado en 10,

$$1234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10 + 4.$$

Nuestra primer tarea será evaluar un polinomio en forma eficiente, para lo cual observamos que sacando factor común en expresiones cada vez más pequeñas,

$$\begin{aligned} P(x) &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0, \\ &= (a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_1) x + a_0, \\ &= ((a_n x^{n-2} + a_{n-1} x^{n-3} + \cdots + a_2) x + a_1) x + a_0, \\ &\vdots \end{aligned}$$

llegamos a la *regla de Horner*,

$$P(x) = (((\cdots((a_n x + a_{n-1}) x + a_{n-2}) + \cdots) x + a_1) x + a_0). \quad (13.15)$$

**E 13.10 (regla de Horner).** Construir una función que dada una lista de coeficientes (reales)  $(a_0, a_1, a_2, \dots, a_n)$  y un número  $x \in \mathbb{R}$ , evalúe el polinomio  $P(x)$  en la [ecuación \(13.14\)](#).

Hacerlo de tres formas:



- a) Calculando la suma de términos como se indica en la [ecuación \(13.14\)](#), calculando  $x^k$  para cada  $k$ .

✎ Es decir, algo como

```
s = 0
for k in range(n + 1):
    s = s + a[k] * x**k
...
```

- b) Como el anterior, pero las potencias en la forma  $x^{k+1} = x^k \times x$ , guardando  $x^k$  en cada paso.

✎ Es decir, algo como

```
pot = 1
s = a[0]
for k in range(1, n + 1):
    pot = pot * x
    s = s + a[k] * pot
...
```

- c) Usando la [regla de Horner \(13.15\)](#).

✎ Es decir, algo como

```
s = 0
for c in a[::-1]:
    s = s * x + c
...
```



Algunos comentarios:

- La regla de Horner tiene similitudes con otros esquemas que hemos visto: en los esquemas (13.7) o (13.9) multiplicamos y sumamos cosas variables, en la [ecuación \(13.12\)](#) se multiplican y suman cantidades constantes, y ahora multiplicamos por algo fijo y sumamos algo variable.
- En la regla de Horner empezamos multiplicando el coeficiente más importante  $a_n$  (el que le da el grado al polinomio) y terminamos sumando el menos importante,  $a_0$ . En cambio, en las dos

primeras versiones empezamos usando  $a_0$  y terminamos con  $a_n$ .

- En las dos primeras versiones del [ejercicio 13.10](#) se hacen  $n$  sumas,  $n$  productos y se calculan  $n - 1$  potencias, que en la primera versión representan otros  $1+2+\dots+(n-1) = n(n-1)/2$  productos, mientras que en la segunda los productos provenientes de las potencias se reducen a  $n - 1$ .

En cambio, en la regla de Horner tenemos sólo  $n$  sumas y  $n$  productos, y es mucho más eficiente que las dos primeras versiones.

*En la mayoría de los casos es preferible usar la regla de Horner para evaluar polinomios, y evitar la potenciación explícita.*

**E 13.11 (escritura en base entera).** Dado una base  $b$  entera,  $b > 1$ , y un entero  $n \in \mathbb{N}$ , queremos encontrar coeficientes  $a_0, a_1, \dots, a_k$  de modo que

$$n = \sum_{i=0}^k a_i b^i, \quad \text{donde } a_i \in \mathbb{Z} \text{ y } 0 \leq a_i < b. \quad (13.16)$$

Siguiendo las ideas de la regla de Horner en [ecuación \(13.15\)](#), ponemos

$$n = ((\dots((a_k b + a_{k-1}) \times b + a_{k-2}) + \dots) \times b + a_1) \times b + a_0.$$

A partir de estas expresiones, vemos que  $a_0$  se obtiene como resto de la división de  $n$  por  $b$ , por lo que  $b \mid (n - a_0)$ , y  $(n - a_0)/b$  es un entero que tiene resto  $a_1$  cuando dividido por  $b$ , etc.

✎ También usamos este proceso en el [ejercicio 8.8](#).

- Definir una función que dados  $n$  y  $b$  construya una lista con los coeficientes  $a_0, a_1, \dots, a_k$  de la [expresión \(13.16\)](#), con  $a_k \neq 0$ . Por ejemplo, si  $n = 10$  y  $b = 2$ , se debe obtener  $[0, 1, 0, 1]$ .

- ✎ Como el último coeficiente debe ser no nulo (pedimos  $n > 0$ ), la longitud de la lista obtenida debe ser la cantidad de cifras de  $n$  cuando expresado en base  $b$ .
- b) Al escribir un número en alguna base, normalmente se ponen las cifras más significativas adelante: definir una función que dados  $n$  y  $b$  exprese a  $n$  en base  $b$  como cadena de caracteres. Por ejemplo, si  $n = 10$  y  $b = 2$  se debe obtener `'1010'`.
- c) Usando la regla de Horner, definir una función para obtener el número  $n$  (entero positivo) escrito en base 10, dada su representación como cadena de dígitos en la base  $b$ . Por ejemplo, si la cadena es `'1010'` y  $b = 2$  se debe obtener `10`.
- ✎ En Python podemos poner `int(cadena, b)` donde la base  $b$  debe ser un entero tal que  $2 \leq b \leq 36$ , y `cadena` es del tipo `str` con dígitos según la base  $b$ . Si  $b$  se omite se supone que es 10.
- d) En la ecuación (13.16), ¿cómo se relacionan  $k$  y  $\log_b n$  (si  $a_k \neq 0$ )?
- ✎ Recordar también el ejercicio 3.23. ¶

## 13.4. Ecuaciones diofánticas y la técnica de barrido

Comenzamos resolviendo ecuaciones donde las incógnitas son números enteros en las que hay más incógnitas que ecuaciones. y las técnicas aproximadas que veremos en el capítulo 16 no pueden aplicarse.

Por ejemplo, el máximo común divisor y el mínimo común múltiplo de enteros positivos  $a$  y  $b$  pueden pensarse como soluciones a ecuaciones de este tipo: como vimos en el problema de Pablito y su papá (ejercicio 8.13), para el mcm queremos encontrar  $x$  y  $y$  enteros positivos tales que  $ax = by$ . En general el problema tiene infinitas soluciones (si el par  $(x, y)$  es solución, también lo será  $(kx, ky)$  para  $k$  entero positivo), y buscamos el par más chico entre los positivos.

En esta sección nos concentramos en ecuaciones algebraicas (polinómicas) con coeficientes enteros donde sólo interesan las soluciones enteras llamadas *diofánticas* o *diofantinas* en honor a Diofanto de Alejandría (aproximadamente 200–284) quien fue el primero en estudiar sistemáticamente problemas de ecuaciones con soluciones enteras, siendo autor del influyente libro *Aritmética*.

En esta sección veremos varias de estas ecuaciones y su solución mediante la llamada *técnica de barrido*.<sup>(2)</sup>

**E 13.12.** Geri y Guille compraron botellas de vino para una reunión. Ambos querían quedar bien y Guille gastó \$ 125 por botella, mientras que Geri, que es más sibarita, gastó \$ 400 por botella. Si entre los dos gastaron \$ 2875, ¿cuántas botellas compró cada uno?, ¿cuánto gastó cada uno?<sup>(3)</sup>

a) Encontrar una solución (con lápiz y papel).

*Respuesta:* si pedimos que cada uno haya comprado al menos una botella, hay una única solución: Guille compró 7 botellas y Geri compró 5. Si permitimos valores nulos debemos agregar la posibilidad de que Guille haya comprado 23 botellas y Geri ninguna.

b) Definir una función para resolver el problema.

*Ayuda:* indicando por  $x$  la cantidad que compró Guille, y por  $y$  la cantidad que compró Geri, se trata de resolver la ecuación  $125x + 400y = 2875$ , con  $x, y \in \mathbb{Z}$ ,  $x, y \geq 0$ . Por lo tanto,  $0 \leq x \leq \lfloor \frac{2875}{125} \rfloor = 23$ . Usando un lazo **for**, recorrer los valores de  $x$  posibles,  $x = 0, 1, \dots, 23$ , buscando  $y \in \mathbb{Z}$ ,  $y \geq 0$ .

☞ Debe descartarse un doble lazo **for** como en el esquema

```
for x in range(24):      # 2875 // 125  -> 23
    for y in range(8):   # 2875 // 400  -> 7
        if 125 * x + 400 * y == 2875:
            ...
```

<sup>(2)</sup> Bah, ¡a lo bestia!

<sup>(3)</sup> ¡En el [ejercicio 3.10](#) también comprábamos botellas!

que es sumamente ineficiente pues estamos haciendo  $23 \times 7 = 161$  pasos.

✍ En cambio debe usarse (en este caso) un único lazo, ya sea

```
for x in range(1 + 2875 // 125):
    y = (2875 - 125 * x) // 400
    if 125 * x + 400 * y == 2875:
        ...
```

(13.17)

que realiza 9 pasos, o bien

```
for y in range(1 + 2875 // 400):
    x = (2875 - 400 * y) // 125
    ...
```

que realiza 24 pasos.

En este caso es más eficiente el primero (pues hay menos valores de  $x$  que de  $y$ ), pero cualquiera de los dos esquemas es aceptable.

✍ También podríamos poner

```
for x in range(1 + 2875 // 125):
    y = int((2875 - 125 * x) / 400)
    ...
```

pero estaríamos ignorando las ventajas de usar división entera antes que la decimal más una conversión.

✍ En fin, también podríamos poner

```
for x in range(1 + 2875 // 125):
    y = (2875 - 125 * x) / 400
    if y == int(y): # si y es entero
        ...
```

ahorrandos la verificación  $125 * x + 400 * y == 2875$ .

Esta versión puede no dar resultados correctos si  $y$  no es entero pero  $y == \text{int}(y)$  para Python debido a errores numéricos. Por ejemplo:

```
>>> y = (10**20 - 1)/10**20      # decimal
>>> y
1.0
>>> int(y) == y
```

```
True
>>> (10**20 - 1)//10**20      # entero
0
```

- c) Construir una función para resolver en general ecuaciones de la forma  $ax + by = c$ , donde  $a, b, c \in \mathbb{N}$  son dados por el usuario y  $x, y$  son incógnitas enteras no negativas. La función debe retornar todos los pares  $[x, y]$  de soluciones en una lista, retornando la lista vacía si no hay soluciones.

✎ Recordar los comentarios hechos sobre la eficiencia.



La estrategia de resolución del [ejercicio 13.12](#) es recorrer todas las posibilidades, una por una, y por eso se llama de *barrido*.

- ✎ La ecuación que aparece en ese ejercicio es de la forma  $ax + by = c$  y por lo tanto lineal. Como en el caso del mcd y el mcm, hay técnicas mucho más eficientes para resolverlas basadas en el algoritmo de Euclides (e igualmente elementales), pero no las veremos en el curso.

La técnica de barrido puede extenderse para «barrer» más de dos números, como en el siguiente ejercicio.

**E 13.13.** En los partidos de rugby se consiguen tantos mediante tries (5 tantos cada try), tries convertidos (7 tantos cada uno) y penales convertidos (3 tantos cada uno).

Definir una función que ingresando la cantidad total de puntos que obtuvo un equipo al final de un partido, imprima todas las formas posibles de obtener ese resultado. Por ejemplo, si un equipo obtuvo 21 tantos, debería imprimirse algo como:

Posibilidad	Tries	Tries convertidos	Penales
1	0	0	7
2	0	3	0
3	1	1	3
4	3	0	2

- ☛ Teniendo en cuenta los comentarios sobre la eficiencia hechos en el [ejercicio 13.12](#) (y en particular el [esquema \(13.17\)](#)), acá habrá que hacer dos lazos **for** pero no tres.
- ☛ Se pide hacer una tabla razonablemente prolija, usando **print** con espacios adecuados. No se piden columnas alineadas (a derecha, izquierda o centradas) como los de la [sección 11.1](#), pero no vendría mal repasar.



La misma técnica de «barrido» puede usarse para resolver ecuaciones diofánticas no lineales.

**E 13.14.** Definir una función que dado  $n \in \mathbb{N}$  determine si existen enteros no negativos  $x, y$  tales que  $x^2 + y^2 = n$ , exhibiendo en caso positivo un par de valores de  $x, y$  posibles, y en caso contrario imprimiendo un cartel adecuado.

- ☛ Teniendo en cuenta los comentarios del [ejercicio 13.12](#) alrededor del [esquema \(13.17\)](#), *sin usar raíces cuadradas* (o algún sustituto) con las herramientas que tenemos no podemos hacer mucho mejor que algo como:

```
for x in range(1, n + 1):
    for y in range(1, n + 1):
        if x * x + y * y == n:
            ...
```

si bien —como estamos buscando una y no todas las soluciones— podríamos considerar sólo soluciones donde  $x \leq y$ :

```
for x in range(1, n + 1):
    for y in range(x, n + 1):
        if x * x + y * y == n:
            ...
```

En fin, usando la raíz cuadrada podríamos poner

```
for x in range(1, 1 + math.sqrt(n)):
    y = int(math.sqrt(n - x**2))
    if x * x + y * y == n:
        ...
```

pero para  $n$  grande `int(sqrt(n))` en general es distinto del valor teórico  $\lfloor \sqrt{n} \rfloor$ .

- ✎ En 1747 Euler dio una caracterización de los enteros positivos que pueden escribirse como suma de dos cuadrados, y en 1770 Lagrange demostró que todo entero positivo es suma de cuatro cuadrados (algunos eventualmente nulos). ¶

## 13.5. Cribas

Una *criba* (o *cedazo* o *tamiz*) es una selección de elementos de una lista según un criterio que depende de otros elementos de la lista.

Quizás la más conocida de las cribas sea la atribuida a Eratóstenes (276 a. C.–194 a. C.) para encontrar los números primos entre 1 y  $n$  ( $n \in \mathbb{N}$ ), donde el criterio para decidir si un número  $k$  es primo o no es la divisibilidad por los números que le precedieron.

Como ya mencionamos (página 83), para nosotros un número entero  $p$  es primo si  $1 < p$  y sus únicos divisores positivos son 1 y  $p$ , y será conveniente tener presente la siguiente propiedad de demostración sencilla:

**13.15. Propiedad.** Consideremos  $a$ ,  $b$  y  $n$  enteros positivos.

- a) Si  $a \times b = n$ , entonces o  $a \leq \sqrt{n}$  o  $b \leq \sqrt{n}$ .  
 b)  $n$  no es primo si y sólo si existe  $a \in \mathbb{N}$ ,  $1 < a \leq \sqrt{n}$  tal que  $a \mid n$ .

**E 13.16 (criba de Eratóstenes).** Supongamos que queremos encontrar todos los primos menores o iguales que un dado  $n \in \mathbb{N}$ . Recordando que 1 no es primo, empezamos con la lista

$$2 \quad 3 \quad 4 \quad \dots \quad n.$$

Recuadramos 2, y tachamos los restantes múltiplos de 2 de la lista, que no pueden ser primos, quedando

$$\boxed{2} \quad 3 \quad \cancel{4} \quad 5 \quad \cancel{6} \quad \dots$$

Ahora miramos al primer número que no está marcado (no tiene recuadro ni está tachado) y por lo tanto no es múltiplo de ningún número menor: 3. Lo encuadramos, y tachamos de la lista todos los múltiplos de 3 que quedan sin marcar. La lista ahora es



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

y seguimos de esta forma, recuadrando y tachando múltiplos hasta agotar la lista. Los números recuadrados serán todos los primos que no superan  $n$ .

- a) La función **criba** en el módulo **eratostenes** es una implementación de esta idea, donde indicamos que un número está tachado o no con el arreglo **esprimo** que tiene valores lógicos.
- b) Una vez que **i** en el lazo principal supera a  $\sqrt{n}$ , no se modificará su condición de ser primo o no, por la **propiedad 13.15**.

Por lo tanto, podemos reemplazar **range(2, n + 1)** en el lazo principal por **range(2, s + 1)**, donde  $s = \lfloor \sqrt{n} \rfloor$ .

⚠ Recordar que para  $a > 0$ , **int(a)** da por resultado  $\lfloor a \rfloor$ .

Como hemos mencionado, si  $n \in \mathbb{N}$  es muy grande es posible que el valor de **int(sqrt(n))** no sea el teórico  $\lfloor \sqrt{n} \rfloor$ .

Hacer estos cambios, viendo que para  $n = 10, 100, 1000$  se obtienen los mismos resultados.

- ⚠ La criba de Eratóstenes es muy eficiente. La cantidad de pasos que realiza es del orden de  $n \log(\log n)$ , mientras que la cantidad de primos que no superan  $n$ ,  $\pi(n)$ , es del orden de  $n / \log n$  según el teorema de los números primos. Es decir, el trabajo que realiza la criba es prácticamente lineal con respecto a la cantidad de elementos que encuentra.



**E 13.17.** En la cárcel había  $n$  celdas numeradas de 1 a  $n$ , cada una ocupada por un único prisionero. Cada celda podía cambiarse de abierta a cerrada o de cerrada a abierta dando media vuelta a una llave. Para celebrar el Centenario de la Creación de la República, se resolvió dar una amnistía parcial. El Presidente envió un oficial a la cárcel con la instrucción:

*Para cada  $i = 1, 2, \dots, n$ , girar media vuelta la llave de las celdas  $i, 2i, 3i, \dots$*

comenzando con todas las celdas cerradas. Un prisionero era liberado si al final de este proceso su puerta estaba abierta. ¿Qué prisioneros fueron liberados?

*Sugerencia:* ¡no pensar, hacer el programa!

✎ Se puede demostrar que se obtienen los cuadrados  $1, 4, 9, \dots$  que no superan  $n$ .



## 13.6. El problema de Flavio Josefo

En esta sección estudiamos el caso donde la criba se hace sobre una lista «circular»: al último elemento le sigue el primero, y a medida que sacamos elementos el círculo se va reduciendo.

Lo ejemplificamos con el *problema de Flavio Josefo*:

*Durante la rebelión judía contra Roma (unos 70 años d. C.), 40 judíos quedaron atrapados en una cueva. Prefiriendo la muerte antes que la captura, decidieron formar un círculo, matando cada 3 de los que quedaran, hasta que quedara uno solo, quien se suicidaría. Conocemos esta historia por Flavio Josefo (historiador famoso), quien siendo el último de los sobrevivientes del círculo, no se suicidó.*

Desde las matemáticas, nos interesa determinar la posición en la que debió colocarse Flavio Josefo dentro del círculo para quedar como último sobreviviente: la *permutación de Flavio Josefo* es el orden en que van saliendo, incluyendo el que queda al final.

Para analizar el problema —y en vez de ser tan crueles y matar personas— supondremos que tenemos inicialmente un círculo de  $n$  jugadores, y que «sale» del círculo el  $m$ -ésimo comenzando a contar a partir del primero, recorriendo los círculos —cada vez más reducidos— siempre en el mismo sentido.

En la [figura 13.2](#) ilustramos el caso  $n = 5$  y  $m = 3$ . Inicialmente los jugadores están en las posiciones numeradas 1 a 5 (con el sentido

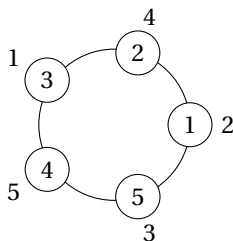


Figura 13.2: Esquema del problema de Flaviio Josefo con  $n = 5$  y  $m = 3$ .

antihorario), y los números fuera del círculo indican en qué momento salieron: saldrán sucesivamente los jugadores numerados 3, 1, 5, 2, quedando al final sólo el número 4, el «sobreviviente».

**E 13.18.** Para estudiar el problema con Python, podríamos considerar una lista `quedan` (¿de los que quedan!), inicialmente con los valores  $0, \dots, n-1$  (en vez de entre 1 y  $n$ ). A medida que sacamos jugadores de `quedan` los incorporamos a la lista `flavio`, que indica el orden en que fueron saliendo las personas. En el caso  $n = 5$  y  $m = 3$ , al terminar `flavio` será `[2, 0, 4, 1, 3]`.

Tendríamos algo como:

```
quedan = list(range(n)) # entre 0 y n-1
nq = n                 # la cantidad que queda
flavio = []            # no salió nadie
s = -1                 # empezar a contar desde 0
for vuelta in range(n): # n vueltas
    s = (s + m) % nq
    sale = quedan.pop(s) # sale del círculo
    nq = nq - 1          # queda uno menos
    flavio.append(sale)
    s = s - 1           # contar a partir de acá
```

(13.18)

a) Ver que el esquema da una solución al problema.

- b) Construir una función **josefo** que toma como argumentos a los enteros positivos **n** y **m**, y retorna la lista **flavio**, indicada anteriormente.
- c) Modificar la función del apartado anterior de modo que retorne la tupla (**flavio**, **salio**), donde **salio** es una lista tal que **salio[j]** indica en qué momento (empezando desde 0) salió la persona numerada **j**.  
En el caso  $n = 5$  y  $m = 3$ , se debe retornar  $([2, 0, 4, 1, 3], [1, 3, 0, 4, 2])$ .
- d) La permutación de Flavio Josefo, **flavio**, es la permutación inversa de **salio**: **flavio[i]** es  $j \Leftrightarrow \text{salio}[j]$  es **i**. Comprobarlo para distintos valores de  $m$  y  $n$ .
- e) ¿Qué posición ocupaba en el círculo inicial Flavio Josefo (en la versión original del problema)?
- f) ¿Cuál es el sobreviviente si  $n = 1234$  y  $m = 3$ ?
- g) Modificar la función de modo que tome los argumentos **n**, **m** y **s** y responda cuántas vueltas sobrevivió la persona que estaba inicialmente en el lugar **s**. ¶

## 13.7. Ejercicios adicionales

**E 13.19 (Potencias binarias).** Otra variante de la idea de la regla de Horner es el cálculo de una potencia «pura». Si  $x \in \mathbb{R}$  y  $n \in \mathbb{N}$ , podemos calcular  $x^n$  escribiendo  $n$  en base 2:

$$n = \sum_{i=0}^k a_i 2^i,$$

y hacer

$$\begin{aligned} x^n &= x^{a_0 + 2a_1 + 2^2a_2 + \dots + 2^{k-1}a_{k-1} + 2^ka_k} = \\ &= x^{a_0} \cdot (x^2)^{a_1} \cdot (x^4)^{a_2} \dots (x^{2^{k-1}})^{a_{k-1}} \cdot (x^{2^k})^{a_k}. \end{aligned}$$


Observando que las potencias de  $x$  son  $x, x^2, x^4, \dots$ , llegamos a un esquema como:

```

pot = x      # x, x**2, x**4, ...
prod = 1     # el producto a retornar
while True:
    if n % 2 == 1:          # coeficiente no nulo
        prod = prod * pot
    n = n // 2
    if n > 0:
        pot = pot * pot    # x**k -> x**(2 * k)
    else:
        break
return prod

```

- Definir una función para el cálculo de potencias «puras» con estas ideas.
- Hacer una función para calcular  $x^n$  usando un lazo **for**.
- Comparando las funciones definidas en los apartados anteriores: ¿cuántas operaciones aritméticas (productos y divisiones) se realizan en cada caso?
- Calcular  $456^{789}$  de tres formas: usando **x\*\*n**, y con las funciones definidas en los dos primeros apartados.

🔗 Debido a su alta eficiencia, una técnica similar se usa para encriptar mensajes usando primos de centenas de cifras. 

**E 13.20 (período de una fracción).** A diferencia de las cifras enteras, que se van generando de derecha a izquierda por sucesivas divisiones (como en la función **cifras** del módulo **ifwhile**), la parte decimal se va generando de izquierda a derecha (también por divisiones sucesivas).

De la escuela recordamos que todo número racional  $p/q$  tiene una «expresión decimal periódica». Por ejemplo:

- $1/7 = 0.1428571428 \dots = 0.\overline{142857}$  tiene período 142857,

- $617/4950 = 0.124646 \dots = 0.12\overline{46}$  tiene período 46 y anteperíodo 12,
- $1/4 = 0.25 = 0.2500 \dots = 0.25\overline{0}$  tiene período 0 y anteperíodo 25,
- hay números como  $1/10 = 0.0\overline{9} = 0.1\overline{0}$  que tienen dos representaciones, una con período 9 y otra con período 0.

- a) Construir una función `periodo(p, q)` que dados los enteros positivos  $p$  y  $q$ , imprima la parte decimal (del desarrollo en base 10) de la fracción  $p/q$ , distinguiendo su período y anteperíodo.

El comportamiento debe ser algo como:

```
>>> periodo(617, 4950)
Los decimales son: [1, 2, 4, 6]
El anteperíodo es: [1, 2]
El período es:     [4, 6]
```

*Sugerencia:* guardar los cocientes y restos sucesivos de la división hasta que se repita un resto.

☞ *Sugerencia si la anterior no alcanza:* en cuanto un resto se repite, se repetirá el cociente, así como todos los restos y cocientes siguientes.

- b) Definir una función que dados el anteperíodo y el período (como listas de números) encuentre el numerador y denominador. Es decir, encontrar básicamente una inversa a la función del apartado anterior.
- c) Encontrar el entero  $n$  entre 1 y 1000 tal que  $1/n$  tiene máxima longitud del período.

*Respuesta:* 983, que tiene período de longitud 982.

- d) Encontrar todos los enteros entre 2 y 1000 para los cuales  $1/n$  tiene período  $n - 1$ . ¿Son todos primos?, ¿hay primos que no cumplen esta condición?



## 13.8. Comentarios

- En cursos de matemática discreta es usual estudiar las relaciones de recurrencia *lineales, homogéneas y con coeficientes constantes*,

$$a_n = A a_{n-1} + B a_{n-2} \quad \text{para } n > 1, \quad (13.19)$$

donde  $A$  y  $B$  son constantes. Se ve que si la ecuación cuadrática *característica*,  $x^2 = Ax + B$ , tiene dos raíces distintas  $r$  y  $r'$  (podrían ser complejas), entonces existen constantes  $c$  y  $c'$  de modo que los números  $a_n$  en (13.19) satisfacen la relación

$$a_n = c r^n + c' (r')^n,$$

siendo la fórmula de Euler-Binet (13.3) un caso particular.

- La forma de calcular las cuotas fijas en préstamos que vimos en la [sección 13.2.3](#) a veces se denomina *sistema francés*.

En contraposición, en el *sistema alemán* se divide el monto inicial por el número de cuotas, mensualmente la deuda disminuye en esa cantidad y se calculan los intereses sobre la deuda que queda. En consecuencia, las cuotas mensuales no son constantes.

- Los números de Fibonacci aparecen en contextos muy diversos, algunos insospechados como la forma de las flores del girasol, y tienen aplicaciones prácticas y teóricas.

Por ejemplo, han sido usados para resolver problemas de confiabilidad de comunicaciones y algunas bases de datos (árboles de Fibonacci) se construyen usando propiedades de estos números.

Vale la pena mencionar que en el Congreso Internacional de Matemáticas de 1900, David Hilbert (1862–1943) propuso una serie de problemas que determinaron en gran parte las investigaciones matemáticas durante el siglo XX. El décimo de estos problemas pide encontrar un algoritmo para determinar si un polinomio con coeficientes enteros, arbitrariamente prescripto, tiene raíces enteras (resolver la ecuación diofántica asociada). En

1970 Yuri Matijasevich, quien entonces tenía 22 años, demostró que no existe tal algoritmo ¡usando números de Fibonacci!

- Posiblemente las ecuaciones diofánticas no lineales más estudiadas sean las de la forma  $x^n + y^n = z^n$ . Cuando  $n = 2$ , las soluciones forman una *terna pitagórica* y están completamente caracterizadas desde la antigüedad. Para  $n > 2$  la ecuación no tiene soluciones. Justamente P. Fermat (1601–1665) escribió en el margen de su copia de la «Aritmética» de Diofanto (traducida por Bachet) sobre la imposibilidad de resolución cuando  $n > 2$ :

*Encontré una demostración verdaderamente destacable, pero el margen del libro es demasiado pequeño para contenerla.*

R. Taylor y A. Wiles demostraron el teorema en 1994, ¡casi 350 años después!

- Divisibilidad y números primos han dejado de ser temas exclusivamente teóricos: el problema más acuciante es encontrar un algoritmo eficiente para factorizar enteros: ¡quien lo encuentre puede hacer colapsar al sistema bancario mundial!

Curiosamente, decidir si un número es primo o no es mucho más sencillo: en 2002, M. Agrawal, N. Kayal y N. Saxena probaron que existe un algoritmo muy eficiente para este problema.

- Si bien se sabe que hay infinitos primos desde Euclides, recién hacia fines del siglo XIX pudieron resolverse algunas cuestiones muy generales sobre cómo se distribuyen.

El *teorema de los números primos*, mencionado en el [ejercicio 13.16](#), fue conjeturado por Gauss cuando tenía unos 15 años, y fue demostrado en 1896 independientemente por J. Hadamard (1865–1963) y C. J. de la Vallée-Poussin (1866–1962).

- Los ejercicios [13.17](#) y [13.18](#) están tomados de [Engel \(1993\)](#).





# Capítulo 14

## Recursión

---

**recursivo, va.** 1. adj. ver recursivo.

---

### 14.1. Introducción

Una forma de sumar los números  $a_0, a_1, a_2, \dots$ , es ir construyendo las sumas parciales que llamamos *acumuladas* en el [ejercicio 10.11](#):

$$s_0 = a_0, s_1 = s_0 + a_1, s_2 = s_1 + a_2, \dots, s_n = s_{n-1} + a_n, \dots,$$

ecuaciones que pueden expresarse más sencillamente como

$$s_0 = a_0 \quad \text{y} \quad s_n = s_{n-1} + a_n \quad \text{para } n \geq 1. \quad (14.1)$$

Cuando la sucesión de números es  $1, 2, 3, \dots$  y cambiamos suma por producto, obtenemos el factorial ([ejercicio 10.9](#)):

$$1! = 1, 2! = 1! \times 2, 3! = 2! \times 3, \dots, n! = (n-1)! \times n, \dots,$$

y en realidad es usual definir  $n!$  mediante

$$0! = 1 \quad \text{y} \quad n! = n \times (n-1)! \quad \text{para } n \in \mathbb{N}. \quad (14.2)$$

Cuando se dan uno o más valores iniciales y una «fórmula» para calcular los valores subsiguientes como en (14.1) o (14.2), decimos que se ha dado una *relación de recurrencia*.

Estas relaciones están estrechamente conectadas con el concepto de *inducción* en matemática y el de *recursión* en programación, aunque los conceptos no son completamente equivalentes. Por ejemplo, la [relación \(14.2\)](#) es la definición inductiva del factorial en matemática, mientras que en programación es común decir que una función es recursiva si en su definición hay una llamada a sí misma (aunque sea a través de otras funciones).

Para nosotros,

*la idea fundamental de recursión es la resolución de un problema usando las soluciones del mismo problema para tamaños más chicos... y así sucesivamente si fuera necesario.*

Por ejemplo, una forma de calcular  $4!$  es calcular primero  $3!$ , para calcular  $3!$  primero calculamos  $2!$ , para calcular  $2!$  primero calculamos  $1!$ , y finalmente  $1! = 1$ .

## 14.2. Funciones definidas recursivamente

**E 14.1.** En Python podemos definir una función recursiva para calcular el factorial basada en la [ecuación \(14.2\)](#), dando el valor inicial para  $n = 1$  y el valor para  $n$  mayores dependiendo del valor anterior:

```
def factorial(n):  
    """n! usando recursión.  
  
    n debe ser entero positivo.  
    """  
    if n == 1:        # paso base
```

```

    return 1
return n * factorial(n-1)

```

- Comparar esta función con la definida en el [ejercicio 10.9](#) para  $n = 1, 2, \dots, 10$ , usando alguna técnica del [ejercicio 10.20](#) para compararlas.
- El esquema anterior no contempla el caso  $n = 0$ , para el cual sabemos que  $0! = 1$ . Modificar la función para incluir también el caso  $n = 0$ .

*Atención:* cambiar la documentación acordemente.

- ¿Qué pasa si  $n < 0$ ?, ¿y si  $n$  no es entero?



Veamos otros ejemplos similares que antes resolvíamos con un lazo **for** o similar, y que ahora podemos resolver usando recursión.

**E 14.2.** Recordando el [ejercicio 10.8](#), definir una función recursiva para calcular las sumas de Gauss

$$s_n = 1 + 2 + \dots + n = \sum_{k=1}^n k,$$

y comparar con las definidas en ese ejercicio.



Las funciones recursivas pueden tener uno o varios argumentos, y podrían no ser números. Si tienen más de un argumento, la recursión puede hacerse tanto en sólo uno de ellos como en varios, como ilustramos en los siguientes ejercicios.

**E 14.3.** Definir una función recursiva para calcular el cociente de la división entre  $a$  y  $b$  cuando  $a$  y  $b$  son enteros positivos, usando que si  $a \geq b$  entonces  $\text{cociente}(a, b) \leftrightarrow \text{cociente}(a - b, b) + 1$ .

*Sugerencia:* **if a < b:...**

- ☞ Acá la función tiene dos argumentos, pero se hace recursión sólo sobre el primero y el segundo actúa como parámetro.



**E 14.4.** Recordando la versión original del algoritmo de Euclides para encontrar el máximo común divisor entre dos enteros positivos (ver [sección 8.3](#)), definir una función recursiva para calcular  $\text{mcd}(a, b)$  cuando  $a$  y  $b$  son enteros positivos.

*Ayuda:* si  $a > b$  entonces  $\text{mcd}(a, b) = \text{mcd}(a-b, b)$ , análogamente para el caso  $a < b$ , y  $\text{mcd}(a, a) = a$ .

✎ En este ejercicio la recursión se hace en ambos argumentos, a diferencia del [ejercicio 14.3](#).

✎ Observar que no se hace la «llamada a  $f(n - 1)$ ».



**E 14.5 (números de Fibonacci II).** En este ejercicio calculamos los números de Fibonacci  $f_n$  ([sección 13.1](#)) usando recursión.

a) Definir una función recursiva para calcular  $f_n$  basada en un esquema del tipo

```
if n < 3:
    return 1
return fibonacci(n - 1) + fibonacci(n - 2)
```

que sigue más o menos literalmente a las [ecuaciones \(13.1\)](#).

✎ Observar que en la versión recursiva se hacen dos llamadas a la misma función.

b) Comparar los resultados entre la versión recursiva y la usando **for** del [ejercicio 13.1](#) para  $n \in \mathbb{N}$ ,  $n \leq 10$ .

c) ¿Qué pasa con la versión recursiva si  $n < 1$ ? ¿es correcto el resultado?



## 14.3. Ventajas y desventajas de la recursión

Explicuemos un poco cómo funciona recursión.

Como sabemos, conceptualmente una función ocupa un lugar en la memoria al momento de ejecución, llamado *marco* o *espacio* o *contexto* de la función. Ese contexto contiene las instrucciones que debe seguir y lugares para las variables locales, como se ilustra en la [figura 7.1](#).

Cada vez que la función se llama a sí misma, podemos pensar que se genera automáticamente una copia de ella (tantas como sean necesarias), cada copia con su contexto propio. Cuando termina su tarea, la copia se borra y el espacio de memoria que ocupaba es liberado (y la copia no existe más).

Este espacio de memoria usado por recursión no está reservado con anterioridad, pues no se puede saber de antemano cuántas veces se usará la recursión. Por ejemplo, para calcular  $n!$  recursivamente se necesitan unas  $n$  copias de la función: cambiando  $n$  cambiamos el número de copias necesarias. Este espacio de memoria especial se llama *stack* o *pila* de recursión. Python impone un límite de unas 1000 llamadas para esa pila.

✎ Un resultado de computación teórica dice que toda función recursiva puede reescribirse sin usar recursión con lazos **while** y listas que se usan como pilas (usando **append** y **pop**) para ir guardando los datos intermedios.

Esto es básicamente lo que haremos al hacer recorridos de grafos en la [capítulo 15](#).

Recursión es muy ineficiente, usa mucha memoria (llamando cada vez a la función) y tarda mucho tiempo. Peor, como no tiene memoria (borra la copia de la función cuando ya se usó) a veces debe calcular un mismo valor varias veces.

Por ejemplo, para calcular el número de Fibonacci  $f_n$  es suficiente poner los dos primeros y construir la lista mirando a los dos anteriores (como hicimos en el [ejercicio 13.1](#)), obteniendo un algoritmo que tarda del orden de  $n$  pasos.

En cambio, para calcular  $f_5$  recursivamente la computadora hace los siguientes pasos, donde cada renglón es una llamada a la función:

$$\begin{aligned} f_5 &= f_4 + f_3 \\ f_4 &= f_3 + f_2 \\ f_3 &= f_2 + f_1 \\ f_2 &\rightarrow 1 \\ f_1 &\rightarrow 1 \end{aligned}$$

$$\begin{aligned}f_2 &\rightarrow 1 \\f_3 &= f_2 + f_1 \\f_2 &\rightarrow 1 \\f_1 &\rightarrow 1\end{aligned}$$

realizando un total de 8 llamadas (haciendo otras tantas copias) además de la inicial.

Una forma intuitiva de ver la ineficiencia de la recursión, es calcular  $f_n$  con las dos versiones del [ejercicio 13.1](#), comparando «a ojo» el tiempo que tarda cada una cuando  $n$  es aproximadamente 30 o 35 (dependiendo de la rapidez de la máquina).

Otra forma más científica es mediante el siguiente ejercicio.

**E 14.6.** Usando un contador global, definir una función para calcular el  $n$ -ésimo número de Fibonacci, imprimiendo la cantidad de llamadas a la función recursiva, siguiendo un esquema del tipo:

```
def llamadasafibo(n):  
    ...  
    global cont  
    def fibc(n):  
        ...  
  
    cont = 0  
    sol = fibc(n)  
    print("Se hicieron", cont, "llamadas")  
    return sol
```

donde la función interna es algo como:

```
def fibc(n):  
    """Fibonacci recursivo con contador global."""  
    global cont  
    cont = cont + 1      # pasó por acá: incrementar  
    if n < 3:  
        return 1  
    return fibc(n - 1) + fibc(n - 2)
```

- ✎ Por supuesto que el peligro al declarar `cont` como global es que haya otra variable global con ese identificador. Una solución artificial es encerrar el contador en una lista, poniendo `cont = [0]` inicialmente, pasar la lista como argumento, y en cada paso poner `cont[0] = cont[0] + 1`. La solución adecuada es usar variables *no locales* con `nonlocal`, que no veremos.

Usando `llamadasafibo`, encontrar la cantidad de llamadas para calcular  $f_5$ ,  $f_{10}$  y  $f_{20}$ .

- ✎ En el [ejercicio 14.8](#) vemos lo absurdo de usar recursión para calcular los números de Fibonacci. ¶

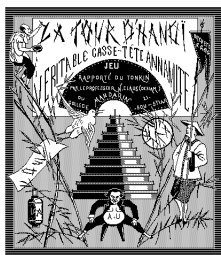
## 14.4. Los Grandes Clásicos de la Recursión

La recursión nos permite hacer formulaciones que parecen más naturales o elegantes, como el algoritmo de Euclides, o el cálculo de los números de Fibonacci, pero en los ejemplos que vimos es más eficiente usar `while` o `for`. En esta sección estudiamos problemas que no son sencillos de resolver usando solamente lazos, y recursión muestra su potencia.

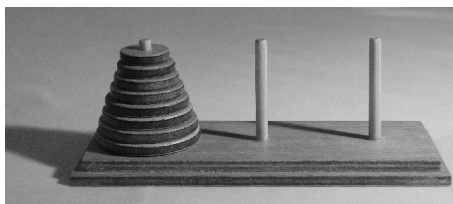
En todo curso de programación que se precie de tal, entre los ejemplos de recursión se encuentran el *factorial*, los *números de Fibonacci*, y las *torres de Hanoi*. Ya vimos el factorial y los números de Fibonacci... o sea...

### E 14.7 (las torres de Hanoi).

*Según la leyenda, en un templo secreto de Hanoi hay 3 agujas y 64 discos de diámetro creciente y los monjes pasan los discos de una aguja a la otra mediante movimientos permitidos. Los discos tienen agujeros en sus centros de modo de encajar en las agujas, e inicialmente todos los discos estaban en la primera aguja con el menor en la cima, el siguiente menor debajo, y así sucesivamente, con el mayor debajo de todos. Un movimiento permitido es la transferencia del*



Tapa del juego original



Fotografía del juego

Figura 14.1: Las torres de Hanoi.

*disco en la cima desde una aguja a cualquier otra siempre que no se ubique sobre uno de diámetro menor. Cuando los monjes terminen de transferir todos los discos a la segunda aguja, será el fin del mundo.*

Nuestra intención es definir una función para indicar los pasos necesarios para realizar esta transferencia cuando hay  $n$  discos, con  $n$  eventualmente distinto de 64. Por ejemplo, a la derecha de la [figura 14.1](#) vemos la posición inicial de  $n = 8$  discos.

Supongamos entonces que tenemos  $n$  discos, «pintados» de 1 a  $n$ , de menor a mayor, y que llamamos a las agujas  $a$ ,  $b$  y  $c$ .

Para pasar los  $n$  discos de  $a$  a  $b$ , en algún momento tendremos que pasar pasar el disco  $n$  de  $a$  a  $b$ , pero únicamente lo podemos hacer si en  $a$  está sólo el disco  $n$  y en  $b$  no hay ningún disco. Es decir, tenemos que pasar los  $n - 1$  discos más chicos de  $a$  a  $c$ , luego el disco  $n$  de  $a$  a  $b$ , y finalmente pasar los  $n - 1$  discos de  $c$  a  $b$ .

Ahora, para pasar  $n - 1$  discos de  $a$  a  $c$ , debemos pasar  $n - 2$  discos de  $a$  a  $b$ , pasar el disco  $n - 1$  a  $c$ , y luego pasar  $n - 2$  discos de  $b$  a  $c$ . Y así sucesivamente.

Esto lo podemos expresar con una función **pasar** que informalmente podríamos poner como:



**función** *pasar*(*n*):

# pasar *n* discos de una aguja a otra

**si** *n* = 1:

    pasar el disco 1

**en otro caso:**

*pasar*(*n* - 1)

    pasar el disco *n*

*pasar*(*n* - 1)

Tenemos que ser un poco más específicos, ya que las agujas en cada caso son distintas. Así, para pasar de la aguja *a* a *b* debemos usar *c* como intermedia, para pasar de *c* a *b* debemos usar *a* como intermedia, etc.

Si genéricamente llamamos *x*, *y* y *z* a las agujas (que pueden ser *a*, *b* y *c* en cualquier orden), podemos poner

```
def pasar(k, x, y, z):
    """Pasar los discos 1,..., k de x a y usando z."""
    if k == 1:
        print('pasar el disco 1 de', x, 'a', y)
    else:
        pasar(k - 1, x, z, y)
        print('pasar el disco', k, 'de', x, 'a', y)
        pasar(k - 1, z, y, x)
```

En la función **hanoi** (en el módulo **recursion1**), pusimos a **pasar** como una función interna, donde las agujas se indican con las letras 'a', 'b' y 'c'.

- a) Agregar un contador (global) para contar la cantidad de veces que se transfiere un disco de una aguja a otra, imprimiendo su valor al terminar.

En base a este resultado (para  $n = 1, 2, 3, 4, 5$ ) conjeturar la cantidad de movimientos necesarios para transferir  $n$  discos de una aguja a otra, y demostrarlo.

*Sugerencia:*  $2^n - 1 = 1 - 2 + 2^n = 1 + 2(2^{n-1} - 1)$ .

- b) Si la pila de discos está inicialmente en la aguja  $a$ , y en el primer movimiento el disco más chico se transfiere a la aguja  $b$ , ¿en qué aguja terminan los  $n$  discos?
- c) Suponiendo que transfieren un disco por segundo de una aguja a otra, ¿cuántos años tardarán los monjes en transferir los 64 discos de una aguja a la otra?
- d) ¿Cuántos años tardaría una computadora en calcular la solución para  $n = 64$ , suponiendo que tarda un nanosegundo (ns) por movimiento<sup>(1)</sup> (nano = dividir por mil millones)?

✎ Un gigahercio (GHz) es  $10^9$  (mil millones) de ciclos por segundo, de modo que las computadoras comunes actuales trabajan a unos 3 ciclos por ns. Suponiendo que la computadora realiza una instrucción por ciclo y teniendo en cuenta que deben hacerse algunos cálculos por movimiento, la estimación de 1 ns por movimiento nos da (a *muy* grandes rasgos) una idea de lo que podría hacer hoy una computadora personal.

- e) Bajo la misma suposición sobre la velocidad de la computadora del apartado anterior, ¿cuál es el valor máximo de  $n$  para calcular los movimientos en 1 minuto?
- f) Modificar la función **hanoi** de modo que las «aguja»  $a$ ,  $b$  y  $c$  sean listas con los números que representan los discos, y que en cada paso se imprima el número de paso y los discos en cada aguja.

Inicialmente debe ser

$$a = [n, n-1, \dots, 1], \quad b = [], \quad c = [],$$

y al terminar debe ser

$$a = [], \quad b = [n, n-1, \dots, 1], \quad c = [].$$

Por ejemplo:

---

<sup>(1)</sup> ¡Y que no hay cortes de luz!

```
>>> hanoi(3)
Paso 0
    a: [3, 2, 1]
    b: []
    c: []
Paso 1
    a: [3, 2]
    b: [1]
    c: []
Paso 2
    a: [3]
    b: [1]
    c: [2]
...
```

*Sugerencia:* usar **pop** para sacar un elemento de una lista y **append** para agregarlo a otra.

- ✎ Hay muchas variantes del problema. Por ejemplo, que los discos no estén inicialmente todos sobre una misma aguja, o que haya más de tres agujas.
- ✎ «Las torres de Hanoi» es un juego inventado en 1883 por el matemático francés Édouard Lucas (1842–1891), quien agregó la «leyenda».

El juego, ilustrado en la [figura 14.1](#), usualmente se les da a los chicos con entre 4 y 6 discos, a veces de distintos colores. Notablemente, variantes del juego se usan en tratamiento e investigación de psicología y neuro-psicología.

Lucas es más conocido matemáticamente por su test de primalidad —variantes del cual son muy usadas en la actualidad— para determinar si un número es primo o no.

- ✎ Python tiene una ilustración animada del problema con 6 discos en el módulo *minimal\_hanoi*.

En MS-Windows, buscarlo en `\Lib\turtledemo` dentro del directorio donde está instalado Python.



## 14.5. Ejercicios adicionales

### E 14.8.

- a) Usando el [ejercicio 14.6](#), primero conjeturar y luego demostrar la relación de recurrencia para la cantidad de llamadas  $c(n)$  de la versión recursiva para calcular el  $n$ -ésimo número de Fibonacci.

*Respuesta:*  $c(n) = c(n-1) + c(n-2) + 1$ .

- b) Ver que  $b(n) = c(n) + 1$  satisface la misma ecuación de recurrencia que  $f_n$ , y por lo tanto  $b(n) = 2f_n$ .

*Sugerencia:* sumar 1 miembro a miembro en la relación de recurrencia para  $c(n)$  y observar que  $b(1) = b(2) = 2$ .

En conclusión: ¡en la versión recursiva básicamente el doble de llamadas a la función interna que el valor mismo de  $f_n$ !



## 14.6. Comentarios

- Las imágenes de la [figura 14.1](#) fueron tomadas respectivamente de<sup>(2)</sup>
  - <http://www.cs.wm.edu/~pkstoc/>
  - [http://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](http://en.wikipedia.org/wiki/Tower_of_Hanoi).



---

<sup>(2)</sup> Enlaces válidos a febrero de 2019.

## Capítulo 15

# Grafos

Muchas situaciones pueden describirse por medio de un diagrama en el que dibujamos puntos y segmentos que unen algunos de esos puntos. Por ejemplo, los puntos puede representar gente y los segmentos unir a pares de amigos, o los puntos pueden representar centros de comunicación y los segmentos enlaces, o los puntos representar ciudades y los segmentos las carreteras que los unen.

La idea subyacente es la de *grafo*, un conjunto de *vértices* (gente, centros o ciudades) y un conjunto de *aristas* (relaciones, enlaces o calles).

### 15.1. Ensalada de definiciones

Comenzamos presentando una «ensalada» de conceptos, definiciones y notaciones, muchos de los cuales pueden no ser familiares o diferir con las de otros autores. La idea es leer esta sección sin tratar de memorizar las definiciones, y volver a ella cuando se use algún término de teoría de grafos que no se recuerde.

El conjunto de vértices se indica por  $V$  y el de aristas por  $E$ . Si llamamos al grafo  $G$ , normalmente pondremos  $G = (V, E)$ .

A fin de distinguir los vértices entre sí es conveniente darles nombres, pero para el uso computacional en general supondremos que si hay  $n$  vértices, éstos se denominan  $1, 2, \dots, n$ , es decir, suponemos  $V = \{1, 2, \dots, n\}$ . Más aún, casi siempre usaremos el nombre  $n$  (o algo que empiece con  $n$ ) para el cardinal de  $V$ .

Las aristas están formadas por dos vértices. En este capítulo sólo consideraremos grafos *simples* en los que no hay aristas uniendo un vértice con sí mismo (aristas que se llaman *lazos* o *bucles*), ni aristas *paralelas* o repetidas que unen los mismos pares de vértices.

En los grafos *dirigidos* o *digrafos*, las aristas están orientadas y se indican como  $(a, b)$  (a veces llamados *arcos* en vez de aristas), distinguiendo entre  $(a, b)$  y  $(b, a)$ . En cambio, en los grafos *no dirigidos* no importa el orden de los vértices que definen una arista, y la arista que une el vértice  $a$  con el vértice  $b$  ( $\Rightarrow a \neq b$ ) por  $\{a, b\}$ . Claro que decir  $\{a, b\} \in E$  es lo mismo que decir  $\{b, a\} \in E$ .

*Salvo mención en contrario, en lo que resta del capítulo sólo consideraremos grafos simples no dirigidos.*

Así como  $n$  es el «nombre oficial» de  $|V|$  (el cardinal de  $V$ ), el «nombre oficial» para  $|E|$  es  $m$ , de modo que en los grafos simples podemos relacionar  $n = |V|$  y  $m = |E|$ : si hay  $n$  elementos, hay  $\binom{n}{2} = n(n-1)/2$  subconjuntos de 2 elementos, de modo que  $m \leq \binom{n}{2}$ .

Si  $e = \{a, b\} \in E$ , diremos que  $a$  y  $b$  son *vecinos* o *adyacentes*, que  $a$  y  $b$  son los *extremos* de  $e$ , o que  $e$  *incide* sobre  $a$  (y  $b$ ). A veces, un vértice no tiene vecinos —no hay aristas que incidan sobre él— y entonces decimos que es un vértice *aislado*.

**Ejemplo 15.1.** En la [figura 15.1](#) representamos un grafo donde  $n = 6$ ,

$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 6\}, \{3, 4\}, \{3, 6\}, \{4, 6\}\},$$

y por lo tanto  $m = 7$ , y el vértice 5 es aislado.



Ejemplo de grafo simple

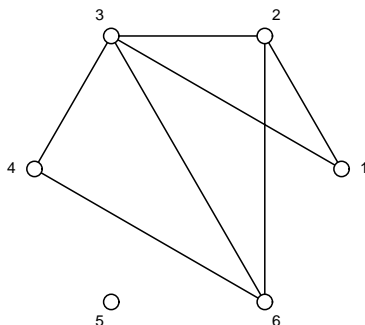


Figura 15.1: Un grafo con 6 vértices y 7 aristas. El vértice 5 es aislado.

Siguiendo con la analogía de rutas, es común hablar de *camino*, una sucesión de vértices —el orden es importante— de la forma  $(v_0, v_1, \dots, v_k)$ , donde  $\{v_{i-1}, v_i\} \in E$  para  $i = 1, \dots, k$ , y sin aristas repetidas (pero pueden haber vértices repetidos). Si  $u = v_0$  y  $v = v_k$ , decimos que el camino  $(v_0, v_1, \dots, v_k)$  es un *camino de  $u$  a  $v$* , o que *une  $u$  y  $v$* , o sencillamente es un *camino  $u-v$* . La *longitud* del camino  $(v_0, v_1, \dots, v_k)$  es  $k$ , la cantidad de aristas que tiene (y *no* la cantidad  $k + 1$  de vértices). Un camino en principio puede tener vértices repetidos, y si se cierra sobre sí mismo de modo que  $v_0 = v_k$ , decimos que se trata de un camino *cerrado* o *ciclo*, mientras que si no tiene vértices repetidos decimos que es un camino *simple*. Del mismo modo, un ciclo es *simple* si no tiene vértices repetidos (salvo el primero y el último).

Así, en el grafo de la [figura 15.1](#):

- $(3, 2, 6, 3, 4)$  es un camino 3–4 de longitud 4,
- $(1, 2, 3)$  es un camino simple,
- $(4, 3, 2, 1, 3, 6, 4)$  es un ciclo,
- $(2, 3, 4, 6, 2)$  es un ciclo simple (no tiene vértices repetidos).

De fundamental importancia es reconocer si un grafo es *conexo*, es decir, si existe un camino desde cualquier vértice a cualquier otro.

La relación « $\sim$ » definida en  $V \times V$  por  $u \sim v$  si y sólo si  $u = v$  o existe un camino  $u-v$ , es una relación de equivalencia,<sup>(1)</sup> y las clases de equivalencia se llaman *componentes conexas* o simplemente *componentes* del grafo. Entonces, un grafo es conexo si y sólo si tiene una única componente.<sup>(2)</sup> Por ejemplo, el grafo de la [figura 15.1](#) no es conexo, pues tiene dos componentes:  $\{1, 2, 3, 4, 6\}$  y  $\{5\}$ .

- ⚡ Podríamos pensar que  $u \sim u$  diciendo que recorremos un camino de longitud 0 (nos quedamos en el mismo lugar). Sin embargo, no es conveniente decir que tal camino es  $(u, u)$ , ya que podría entenderse que hay un *lazo* de  $u$  a  $u$ .

Si el grafo es conexo (y simple), como se puede unir un vértice con los  $n - 1$  restantes, debe tener al menos  $n - 1$  aristas. De modo que para un grafo (simple) conexo,  $m$  tiene que estar básicamente entre  $n$  y  $n^2/2$ .<sup>(3)</sup>

Dada su estructura, es más sencillo trabajar con árboles que con grafos. Un *árbol* es un grafo (simple) conexo y sin ciclos, pero hay muchas formas equivalentes de describirlo, algunas de las cuales enunciamos como teorema (que por supuesto crearemos).

**15.2. Teorema (Caracterizaciones de árboles).** *Dado un grafo simple  $G = (V, E)$  con  $|V| = n$ , las siguientes condiciones son equivalentes:*

- $G$  es un árbol, es decir, es conexo y no tiene ciclos.*
- Para cualesquiera  $a, b \in V$  existe un único camino que los une.*
- $G$  es conexo y  $|E| = n - 1$ .*
- $G$  no tiene ciclos y  $|E| = n - 1$ .*
- $G$  es conexo, y si se agrega una arista entre dos vértices cualesquiera, se crea un único ciclo.*
- $G$  es conexo, y si se quita cualquier arista queda no conexo.*

A veces en un árbol consideramos un vértice particular como *raíz*,

<sup>(1)</sup> Ver también el [ejercicio 15.10](#).

<sup>(2)</sup> Un grafo con un único vértice y sin aristas es conexo y su vértice es aislado.

<sup>(3)</sup> Más precisamente, entre  $n - 1$  y  $n(n - 1)/2$ .



y miramos a los otros vértices como *descendientes* de la raíz: los que se conectan mediante una arista a la raíz son los *hijos*, los que se conectan con un camino de longitud 2 son los *nietos* y así sucesivamente. Dado que hay un único camino de la raíz a cualquier otro vértice, podemos clasificar a los vértices según *niveles*: la raíz tiene nivel 0, los hijos nivel 1, los nietos nivel 2, etc.

Por supuesto, podemos pensar que los nietos son hijos de los hijos, los hijos padres de los nietos, etc., de modo que —en un árbol con raíz— hablaremos de padres, hijos, ascendientes y descendientes de un vértice. Habrá uno o más vértices sin descendientes, llamados *hojas* mientras que la raíz será el único vértice sin ascendientes. También es común referirse al conjunto formado por un vértice (aunque el vértice no sea la raíz) y sus descendientes como una *rama* del árbol.

Cuando en el árbol distinguimos una raíz, muchas veces conviene pensar al grafo como dirigido, *orientando* las aristas o bien en el sentido de los descendientes (empezando desde la raíz), o bien en el sentido de los ascendientes (hacia la raíz), como muestra la [figura 15.4](#), donde el árbol son las aristas en azul, la raíz es 1 y las flechas indican la orientación de las aristas.

## 15.2. Representación de grafos

Antes de meternos de lleno con los algoritmos, tenemos que decidir cómo guardaremos la información de un grafo en la computadora.

Ya sabemos que los vértices serán enteros consecutivos empezando desde 1. Para representar las aristas, como no consideramos la estructura de conjuntos que ofrece Python, en este curso representaremos la arista  $\{a, b\}$  como  $[a, b]$ , aún cuando el orden no importa. Así, en los grafos no dirigidos para nosotros  $[a, b]$  y  $[b, a]$  representan el mismo objeto.

- ✎ Recordando lo hecho en la [el capítulo 12](#) al tratar listas como conjuntos, para conservar la salud mental trataremos de ingresar la arista  $\{u, v\}$  poniendo  $u < v$ , aunque en general no será

necesario. Ver el [ejercicio 15.6](#).

Nos concentraremos en dos formas de representar la información sobre la aristas: mediante la lista de aristas (con sus extremos) y mediante la lista de *adyacencias* o *vecinos*, una lista donde el elemento en la posición  $i$  es a su vez una lista de los vecinos de  $i$ .

Por ejemplo, para el grafo de la [figura 15.1](#), podríamos poner la información como lista de aristas poniendo

```
ngrafo = 6 # cantidad de vértices
aristas = [[1, 2], [1, 3], [2, 3], [2, 6],
           [3, 4], [3, 6], [4, 6]]
```

En la representación mediante lista de vecinos, como los vértices se numeran a partir de 1, será conveniente empezar la lista poniendo **None** en la posición 0 a fin de evitar errores (pero la lista de aristas tiene índices desde 0).

Así, para el grafo de la [figura 15.1](#) podríamos poner

```
vecinos = [None,
           [2, 3], [1, 3, 6], [1, 2, 4, 6],
           [3, 6], [], [2, 3, 4]]
```

En este caso no es necesario especificar la cantidad de vértices: será `len(vecinos) - 1`.

- ✎ Hay información redundante en la representación de un grafo no dirigido mediante lista de vecinos. Por ejemplo, en la lista **vecinos** anterior, como **2** está en **vecinos[1]**, sabemos que  $\{1, 2\}$  es una arista del grafo, y en principio no sería necesario repetir esta información poniendo **1** en **vecinos[2]**.

La redundancia hace que sea preferible el ingreso de la lista de aristas antes que la de vecinos, porque se reducen las posibilidades de error. Esencialmente, ambas requieren del mismo lugar en memoria pues si  $\{a, b\} \in E$ , al ingresarla en la lista de aristas ocupa dos lugares (uno para  $a$  y otro para  $b$ ), y en la lista de vecinos también (un lugar para  $a$  como vecino de  $b$  y otro para  $b$  como vecino de  $a$ ).

- ✎ Otras dos formas de representación de grafos usuales son la *matriz*

de *adyacencias*, una matriz cuyas entradas son sólo 0 o 1 y de modo que la entrada  $ij$  es 1 si y sólo si  $\{i, j\} \in E$ , y la *matriz de incidencias*, una matriz de  $m \times n$ , también binaria, donde la entrada  $ij$  es 1 si y sólo si la arista  $i$ -ésima es incidente sobre el vértice  $j$ .

**E 15.3.** Definir una función para que dada la lista de vecinos imprima, para cada vértice, los vértices que son adyacentes.

Así, para el grafo del [ejemplo 15.1](#) la salida debería ser algo como

Vértice	Vecinos
1	2 3
2	1 3 6
3	1 2 4 6
4	3 6
5	
6	2 3 4



**E 15.4.** Como es útil pasar de una representación a otra, definimos las funciones `dearistasavecinos` y `devecinosaaristas` en el módulo `grafos`.

Comprobar el comportamiento de estas funciones con las entradas del [ejemplo 15.1](#), pasando de una a otra representación (en ambos sentidos).

Observar:

- Como es usual, suponemos que los datos ingresados son correctos: los vértices de las aristas que se ingresan son enteros entre 1 y `ngrafo` y no hay aristas repetidas o de la forma  $\{i, i\}$ .
- En `dearistasavecinos` ponemos explícitamente `vecinos[0] = None`.
- En `devecinosaaristas` sólo agregamos una arista cuando  $v < u$ , evitando poner una arista dos veces.

Pusimos `for v in range(1, ngrafo)`, ya que no existe el vértice 0, y en el lazo debe ser  $v < u \leq ngrafo$ .

- Las inicializaciones de `vecinos` y `aristas` en cada caso.

- Usamos la construcción `for u, v in aristas` en la función `dearistasavecinos`, aún cuando cada arista está representada por una lista (y no una tupla) (ver el [ejercicio 9.7.c](#)). ¶

**E 15.5.** A fin de evitar errores cuando ingresamos datos y no escribir tanto, es conveniente guardar los datos del grafo en un archivo de texto. Por comodidad (y uniformidad), supondremos que el archivo de texto contiene en la primera línea el número de vértices, y en las restantes los vértices de cada arista: *por lo menos debe tener un renglón* (correspondiente al número de vértices), *y a partir del segundo debe haber exactamente dos datos* por renglón.

De modo que el archivo correspondiente al grafo del [ejemplo 15.1](#) contendría (ni más ni menos):

```
6
1 2
1 3
2 3
2 6
3 4
3 6
4 6
```

Definir una función que toma como argumento el nombre de un archivo de texto donde se guarda la información sobre el grafo (como indicada anteriormente). Comprobar la corrección de la función imprimiendo la lista de aristas, una por renglón.

¿Qué pasa si el grafo no tiene aristas? ¶

**E 15.6.** Definir una función `normalizar(aristas)` que modifica la lista de aristas de modo que las aristas sean de la forma  $[u, v]$  con  $u < v$ , y estén ordenadas crecientemente:  $[a, b]$  precede a  $[u, v]$  si  $a < u$  o ( $a = u$  y  $b < v$ ). ¶

**E 15.7.** Un *coloreo* de un grafo  $G = (V, E)$  es asignar un «color» a cada vértice de modo que vértices adyacentes no tienen el mismo «color».

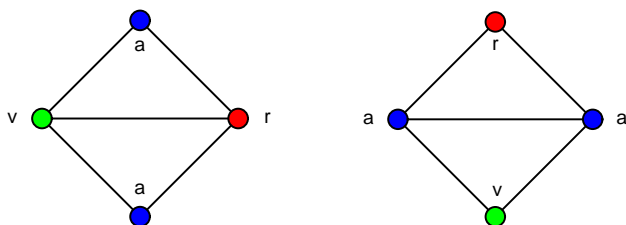


Figura 15.2: Asignando colores a los vértices de un grafo.

Por ejemplo, en la [figura 15.2](#) asignamos los «colores» 'a', 'r' y 'v' a un grafo, siendo el de la izquierda un coloreo pero no el de la derecha.


Definir una función `escoloreo(n, aristas, colores)` que decide si la lista `colores` es un coloreo del grafo asociado: `aristas` es la lista de aristas de un grafo de vértices  $\{1, \dots, n\}$ , y `colores` es una lista de longitud  $n + 1$  de la forma `[colores[0], ..., colores[n]]`.

- ✎ Los valores que toman los elementos de la lista `colores` es irrelevante: podrían ser letras, números o expresiones más complicadas.
- ✎ Como no necesitamos el valor de `colores[0]`, podemos suponer que es `None`, pero no es necesario. ¶

**E 15.8 (grado de vértices).** Dado un grafo  $G = (V, E)$ , para cada vértice  $v \in V$  se define su *grado* o *valencia*,  $\delta(v)$ , como la cantidad de aristas que inciden en  $v$ , o equivalentemente, la cantidad de vecinos de  $v$  (excluyendo al mismo  $v$ ).

Así, en el grafo del [ejemplo 15.1](#), los grados son  $\delta(1) = 2$ ,  $\delta(2) = 3$ ,  $\delta(3) = 4$ ,  $\delta(4) = 2$ ,  $\delta(5) = 0$ ,  $\delta(6) = 3$ .


- a) Definir una función que dado un grafo (ingresado por su cantidad de vértices y una lista de aristas) calcule  $\delta(v)$  para todo  $v \in V$ .
- b) Uno de los primeros teoremas que se ven en teoría de grafos dice que si  $U$  es el conjunto de vértices de grado impar, entonces  $\sum_{v \in U} \delta(v)$  es par.

Definir una función para hacer este cálculo y verificar el teorema para distintos casos (por ejemplo, el grafo de la [figura 15.1](#)). 

**E 15.9.** Es claro que si  $(u = v_0, v_1, \dots, v_k = v)$  es un camino  $u-v$ , entonces  $(v_k, v_{k-1}, \dots, v_0)$  es un camino  $v-u$ , y lo mismo para un ciclo.


Definir una función que ingresando un grafo y una sucesión de vértices  $(v_0, v_1, \dots, v_k)$ ,  $k \geq 1$ :

- a) decida si  $(v_0, v_1, \dots, v_k)$  es un camino, es decir, si  $\{v_{i-1}, v_i\} \in E$  para  $i = 1, \dots, k$ , y no hay aristas repetidas,

 ¡Atención con  $\{u, v\} = \{v, u\}$ !

y en caso afirmativo:


- b) imprima el camino inverso  $v_k, \dots, v_0$ , y

- c) verifique si  $(v_0, v_1, \dots, v_k)$  es un ciclo. 

**E 15.10.** Construir una función que dado un camino  $u-v$  retorne un camino  $u-v$  simple (sin vértices repetidos).

Algunas observaciones:

- Suponemos que  $u \neq v$  y que el camino dado inicialmente es, efectivamente, un camino válido.
- No es necesario saber si hay otras aristas además de las que aparecen en los caminos.
- Se pide un camino simple  $u-v$  pero puede haber más de uno, por ejemplo si la entrada es  $[1, 2, 3, 4, 5, 2, 4, 3]$ , entonces  $[1, 2, 3]$ ,  $[1, 2, 4, 3]$  y  $[1, 2, 5, 4, 3]$  son caminos simples 1-3.

¿Hay alguna relación con el problema de «purgar» una lista ([ejercicio 12.5](#))? 

### 15.3. Recorriendo un grafo

Así como es importante «recorrer» una lista (por ejemplo para encontrar el máximo o la suma), también es importante recorrer un grafo, «visitando» todos sus vértices en forma ordenada, evitando visitar vértices ya visitados, y siempre «caminando» por las aristas del grafo. Exactamente qué hacer cuando se visita un vértice dependerá del problema, y en general podemos pensar que «visitar» es sinónimo de «procesar».

En una lista podemos considerar que los «vecinos» de un elemento son su antecesor y su sucesor (excepto cuando el elemento es el primero o el último), y empezando desde el primer elemento podemos recorrer *linealmente* la lista, mirando al sucesor de turno como hicimos repetidas veces en el [capítulo 10](#). En cambio, en un grafo un vértice puede tener varios vecinos, y el recorrido es más complicado.

No habiendo un «primer vértice» como en una lista, en un grafo elegimos un vértice en donde empezar el recorrido, llamado *raíz*, y luego visitamos a los vecinos, luego a los vecinos de los vecinos, etc., conservando información sobre cuáles vértices ya fueron considerados para no visitarlos nuevamente.

Como los vértices se visitarán secuencialmente, uno a la vez, tenemos que pensar cómo organizarnos para hacerlo, para lo cual apelamos al concepto de *cola* (como la del supermercado): un conjunto al que dinámicamente se le agregan o quitan elementos.

Inicialmente ponemos la raíz en la cola. Posteriormente vamos sacando vértices de la cola para visitarlos y agregando los vecinos de los que estamos visitando. En el [cuadro 15.3](#) mostramos un esquema informal, donde la cola se llama  $Q$  y la raíz  $r$ .

Hay distintos tipos de cola, y entre los más usuales mencionamos:

**Cola *lifo* (last in, first out) o *pila*:** el último elemento ingresado (*last in*) es el primero en salir (*first out*). También la llamamos *pila* porque se parece a las pilas de platos.

**Cola *fifo* (first in, first out):** el primer elemento ingresado (*first*

**Entrada:** un grafo  $G = (V, E)$  y  $r \in V$  (la raíz).

**Salida:** los vértices que se pueden alcanzar desde  $r$  «visitados».

**Comienzo**

$Q \leftarrow \{r\}$

**mientras**  $Q \neq \emptyset$ :

  sea  $i \in Q$

  sacar  $i$  de  $Q$

  «visitar»  $i$    # hacer algo con  $i$

**para todo**  $j$  adyacente a  $i$ :

**si**  $j$  no está «visitado» y  $j \notin Q$ :

      agregar  $j$  a  $Q$

**Fin**

Cuadro 15.3: Esquema del algoritmo *recorrido*.

*in*) es el primero en salir (*first out*). Son las colas que normalmente llamamos... colas, como la del supermercado.

**Cola con prioridad:** Los elementos van saliendo de la cola de acuerdo a cierto orden de prioridad. Por ejemplo, las mamás con bebés se atienden antes que otros clientes.

- ✎ Muchos autores restringen la palabra *cola* a lo que acá llamamos *cola fifo* (la del súper).
- ✎ La lista **quedan** en el **esquema (13.18)** para el problema de Flavio Josefo puede considerarse como una cola: individuos que van saliendo de acuerdo a cierto criterio, aunque no es exactamente de ninguno de los tipos mencionados (fifo, lifo o con prioridad).

Las colas tienen ciertas operaciones comunes: inicializar, agregar un elemento y quitar un elemento:

- Inicializamos la cola con:

| `cola = []`   # la cola vacía



- Para simplificar, vamos a agregar elementos siempre al final de la cola:

```
cola.append(x)
```

- Qué elemento sacar de la cola depende del tipo de cola. Por ejemplo,

```
x = cola.pop()    # para colas lifo (pilas)
x = cola.pop(0)   # para colas fifo
```

- ✎ En Python es mucho más eficiente usar colas lifo, modificando el final con `lista.append(x)` y `x = lista.pop()`, que agregar o sacar en posiciones arbitrarias usando funciones como `lista.insert` o `lista.pop(lugar)`.

Para los ejemplos del curso no hace diferencia porque las listas no son demasiado grandes.

El módulo estándar *collections* implementa listas fifo eficientemente en *collections.deque* (ver el [manual de la biblioteca](#)). Nosotros no lo veremos en el curso.

Volviendo al recorrido de un grafo, una primera versión es la función `recorrido` (en el módulo *grafos*). Esta función toma como argumentos la lista de vecinos (recordando que los índices empiezan desde 1) y un vértice raíz, retornando los vértices para los cuales existe un camino desde la raíz.

En este caso, la cola se ha implementado como pila pues sale el último en ingresar.

La lista `padre` en la función `recorrido` tiene un doble propósito.

Por un lado `padre[v]` nos dice desde qué vértice hemos venido a visitar el vértice `v`. Para destacar la raíz ponemos `padre[raiz] = raiz`,<sup>(4)</sup> y cualquier otro vértice tendrá `padre[v] ≠ v`. Inicialmente el valor es `None` para todo vértice, y al terminar el valor es `None` sólo si no se puede llegar al vértice correspondiente desde la raíz.

Por otro lado `padre` determina un árbol con raíz, y por lo tanto podemos encontrar un camino simple de cualquier vértice en el árbol

---

<sup>(4)</sup> Como siempre, no ponemos tildes en los identificadores para evitar problemas.

a la raíz, lo que explotamos en el [ejercicio 15.15](#).

La [figura 15.4](#) ilustra el resultado de la función `recorrido` aplicada al grafo del [ejemplo 15.1](#) tomando como raíz 1. Podemos apreciar el árbol en color azul, con las flechas que indican el sentido *hijo-padre*. En las aristas está recuadrado el orden con que fueron visitados los vértices y aristas correspondientes, en este caso: 1 (raíz), 3 (usando  $\{1, 3\}$ ), 6 (usando  $\{3, 6\}$ ), 4 (usando  $\{3, 4\}$ ) y 2 (usando  $\{1, 2\}$ ). En cambio, el árbol se reduce a la raíz cuando ésta es 5, y no hay aristas.

### E 15.11 (recorrido de un grafo).

- a) Estudiar la función `recorrido` y comprobar el comportamiento para el grafo del [ejemplo 15.1](#) tomando distintos vértices como raíz, por ejemplo 1, 3 y 5.

🔍 Observar que si la raíz es 1, se «visitan» todos los vértices excepto 5. Lo inverso sucede tomando raíz 5: el único vértice visitado es 5.

- b) Al examinar vecinos del vértice que se visita, hay un lazo que comienza con `for v in vecinos[u]...`

¿Sería equivalente cambiar esas instrucciones por

```
lista = [v for v in vecinos[u]
         if padre[v] == None]
cola = cola + lista
for v in lista:
    padre[v] = u
```

?

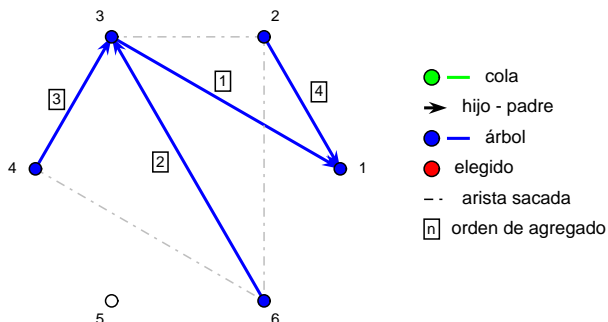


**E 15.12.** Agregar instrucciones a la función `recorrido` de modo que al final se impriman los vértices en el orden en que se incorporaron a la cola, así como el orden en que fueron «visitados» (es decir, el orden en que fueron sacados de la cola).

Por ejemplo, aplicada al grafo del [ejemplo 15.1](#) cuando la raíz es 1 se imprimiría

Orden de incorporación a la cola:

Recorrido con cola LIFO - Paso 5: árbol y cola

Figura 15.4: Recorrido *lifo* del grafo del ejemplo 15.1 tomando raíz 1.

```

1  2  3  4  6
Orden de salida de la cola:
1  3  6  4  2

```


*Sugerencia:* agregar dos listas, digamos **entrada** y **salida**, e ir incorporando a cada una los vértices que entran o salen de la cola.


🔍 Repasar la lista **salio** del ejercicio 13.18.c).



**E 15.13 (componentes).** En el ejercicio 15.11 vimos que no siempre existen caminos desde la raíz a cualquier otro vértice. Lo que hace exactamente la función **recorrido** es construir (y retornar) los vértices de la componente conexa que contiene a la raíz.

- Agregar al grafo del ejemplo 15.1 las aristas  $\{3, 5\}$  y  $\{4, 5\}$ , de modo que el grafo resultante es conexo. Verificarlo corriendo la función **recorrido** para distintas raíces sobre el nuevo grafo.
- En general, para ver si un grafo es conexo basta comparar la longitud (cardinal) de una de sus componentes con la cantidad de vértices. Definir una función que tomando el número de vértices y la lista de aristas, decida si el grafo es conexo o no (retornando **True** o **False**).

- c) Usando la función **recorrido**, definir una función que retorne una lista de las componentes de un grafo. En el grafo original del [ejemplo 15.1](#) el resultado debe ser algo como `[[1, 2, 3, 4, 6], [5]]`. 

**E 15.14.** Agregar instrucciones a la función **recorrido**, de modo que en vez de retornar los vértices del árbol obtenido, retorne la lista de aristas que forman el árbol. 

**E 15.15.** Hacer sendas funciones para los siguientes apartados dado un grafo  $G$ :

- a) Ingresando la cantidad de vértices, la lista de vecinos y los vértices  $s$  y  $t$ ,  $s \neq t$ , se exhiba un camino  $s-t$  o se imprima un cartel diciendo que no existe tal camino.

*Sugerencia:* usar una variante de **recorrido** tomando raíz  $t$  y si al finalizar resulta **padre[s]  $\neq$  None**, construir el camino siguiendo **padre** hasta llegar a  $t$ .

- b) Ingresando la cantidad de vértices y la lista de aristas, se imprima una (y sólo una) de las siguientes:

- i)  $G$  no es conexo,
- ii)  $G$  es un árbol,
- iii)  $G$  es conexo y tiene al menos un ciclo.

*Sugerencia:* recordar el [teorema 15.2](#) y el [ejercicio 15.13.b](#)).

- c) Dados el número de vértices, la lista de aristas y la arista  $\{u, v\}$ , se imprima si hay o no un ciclo en  $G$  que la contiene, y en caso afirmativo, imprimir uno de esos ciclos.

*Ayuda:* una posibilidad es pensar que si hay un ciclo que contiene a la arista  $\{u, v\}$ , debe haber un camino  $u-v$  en el grafo que se obtiene borrando la arista  $\{u, v\}$  del grafo original.

- d) Modificar el [apartado b.iii](#)) de modo de además exhibir un ciclo de  $G$ .

*Sugerencia:* considerar el caso en que un vecino de un vértice que se está visitando ya se ha incorporado a la cola pero no

es el **padre** (esto podría ser más sencillo usando recorrido en profundidad, pero no es necesario).

- ✎ *Sugerencia si la anterior no alcanza:* en recorrido en profundidad, si **u** acaba de incorporarse a la cola y es vecino de **v** que ya se había incorporado, con **padre[u] = w** y **w != v**, entonces recorriendo **padre** hacia atrás a partir de **u** debemos llegar a **v** (sin usar la arista [**u**, **v**]).



**E 15.16 (ciclo de Euler I).** Un célebre teorema de Euler dice que un grafo tiene un ciclo que pasa por todas las aristas exactamente una vez, llamado *ciclo de Euler*, si y sólo si el grafo es conexo y el grado de cada vértice es par.

Recordando el [ejercicio 15.8](#), definir una función que tomando como datos la cantidad de vértices y la lista de aristas, decida (retornando **True** o **False**) si el grafo tiene o no un ciclo de Euler.

- ✎ Un problema *muy* distinto es construir un ciclo de Euler en caso de existir (ver el [ejercicio 15.20](#)).
- ✎ No confundir *ciclo de Euler*, en el que se recorren todas las aristas una única vez (pero pueden repetirse vértices), con *ciclo de Hamilton*, en el que se recorren todos los vértices exactamente una vez (y pueden no usarse algunas aristas).



Como ya observamos, la cola en la función **recorrido** es una pila. Así, si el grafo fuera ya un árbol, primero visitaremos toda una rama hasta el fin antes de recorrer otra, lo que hace que este tipo de recorrido se llame *en profundidad* o de *profundidad primero*. Otra forma de pensarlo es que vamos caminando por las aristas (a partir de la raíz) hasta llegar a una hoja, luego volvemos por una arista (o las necesarias) y bajamos hasta otra hoja, y así sucesivamente.

- ✎ En algunos libros se llama recorrido en profundidad a «visitar» primero todos los vecinos antes de visitar al vértice.
- ✎ El orden en que se recorren los vértices —tanto en el recorrido en profundidad como el recorrido a lo ancho que veremos a continuación— está determinado también por la numeración de los vértices. En la mayoría de las aplicaciones, la numeración

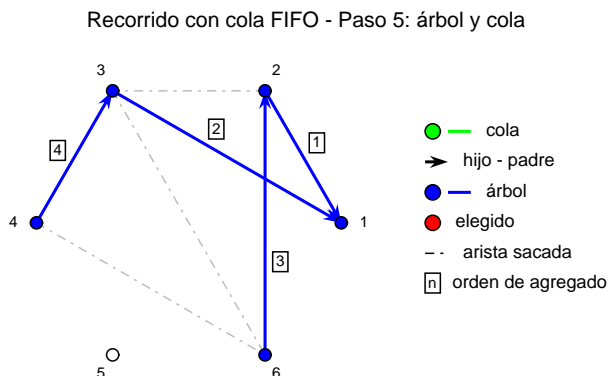


Figura 15.5: Recorrido *fifo* del grafo del [ejemplo 15.1](#) tomando raíz 1.

dada a los vértices no es importante: si lo fuera, hay que sospechar del modelo y mirarlo con cuidado.


Si en la función **recorrido**, en vez de implementar la cola como *lifo* (pila) la implementamos como *fifo*, visitamos primero la raíz, luego sus vecinos, luego los vecinos de los vecinos, etc. Si el grafo es un árbol, visitaremos primero la raíz, después todos sus hijos, después todos sus nietos, etc., por lo que se el recorrido se llama *a lo ancho*.

Al hacer los caminos a la raíz usando la lista **padre**, como hicimos en el [ejercicio 15.15](#), el recorrido a lo ancho construye los caminos más cortos (en cuanto a cantidad de aristas).

### E 15.17 (recorrido a lo ancho).

- Modificar la función **recorrido** de modo que la cola sea ahora *fifo*.

*Sugerencia:* cambiar **pop()** a **pop(0)** en el lugar adecuado.

- Repetir el [ejercicio 15.12](#), comparando las diferencias entre el recorrido en profundidad y el recorrido a lo ancho. 

Una forma dramática de mostrar las diferencias de recorridos *lifo* o *fifo* es construyendo laberintos como los de la [figura 15.6](#).

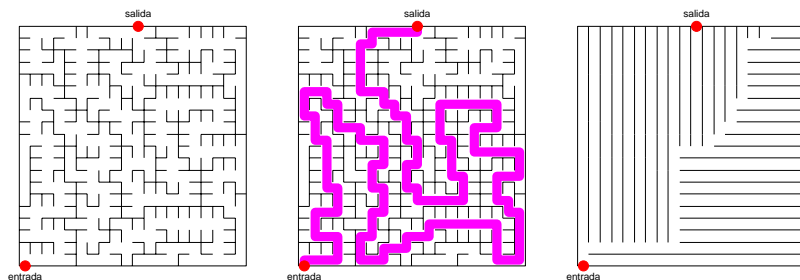


Figura 15.6: Laberinto (izquierda) y solución (centro) usando pila (lifo), y laberinto usando cola fifo (derecha).

Estos laberintos se construyen sobre una grilla de  $m \times n$  de puntos con coordenadas enteras, donde un vértice  $(i, j)$  es vecino del vértice  $(i', j')$  si  $|i - i'| + |j - j'| = 1$ , es decir si uno está justo encima del otro o al costado.

Con `random.shuffle` se da un orden aleatorio a cada lista de vecinos (después de construirlas), y la raíz (entrada) se toma aleatoriamente sobre el borde inferior. Finalmente, se construyen los árboles correspondientes usando la función `recorrido`.

Como se trata de árboles, sabemos que hay un único camino entre cualquier par de vértices. Tomando aleatoriamente un punto de salida sobre el borde superior, podemos construir el único camino entre la entrada y la salida, lo que constituye la «solución» del laberinto.


En la [figura 15.6](#) a la izquierda mostramos el árbol obtenido para cola lifo y a la derecha para cola fifo, para  $m = n = 20$ , usando una misma semilla (con `random.seed`) para tener las mismas listas de vecinos. En la parte central de la figura mostramos la «solución» del laberinto a la izquierda (cola lifo).

Observamos una diferencia notable en la calidad de los árboles. El recorrido con cola fifo o *ancho primero* produce un árbol mucho más «regular». Por supuesto, los árboles tienen la misma cantidad de aristas, pero los caminos obtenidos con recorrido en profundidad primero


produce caminos mucho más largos.

**E 15.18 (recorrido recursivo).** Consideremos el siguiente esquema recursivo para recorrer un grafo a partir de una raíz:

```
def recorrido(vecinos, raiz):  
    """Recorrer el grafo desde la raíz."""  
    def visitar(u):  
        for v in vecinos[u]:  
            if padre[v] == None:  
                padre[v] = u  
                visitar(v)  
    n1 = len(vecinos)  
    vertices = range(n1)  
    padre = [None for v in vertices]  
    padre[raiz] = raiz  
    visitar(raiz)  
    return [v for v in vertices  
            if padre[v] != None]
```

- Decidir si el esquema es correcto, y construir una función en Python haciendo las modificaciones necesarias.
- Modificar la función de modo de imprimir tanto el orden de entrada como el de salida de la función `visitar`.
- Teniendo en cuenta los resultados del apartado anterior, ¿qué relación hay con recorrido a lo ancho y en profundidad? 

## 15.4. Ejercicios adicionales

**E 15.19.** Construir laberintos de  $m \times n$  como los de la [figura 15.6](#) (ver descripción correspondiente). 


**E 15.20 (ciclo de Euler II).** En el [ejercicio 15.16](#) nos hemos referido a la existencia de ciclos de Euler, y nos preocuparemos ahora por encontrar efectivamente uno.



Para demostrar que un grafo conexo con todos los vértices de grado par tiene un ciclo de Euler, se comienza a partir de un vértice y se van recorriendo aristas, borrándolas, hasta que volvamos al vértice original, formando un ciclo. Esto debe suceder porque todos los vértices tienen grado par. Puede ser que el ciclo no cubra a todas las aristas, pero como el grafo es conexo, en ese caso debe haber un vértice en el ciclo construido que tenga al menos dos aristas (aún no eliminadas) incidentes en él, a partir del cual podemos formar un nuevo ciclo, agregarlo al anterior y continuar con el procedimiento hasta haber recorrido y eliminado todas las aristas.

Esta demostración es bien constructiva, y podemos implementar los ciclos como listas, sólo hay que tener cuidado al «juntarlas».

Definir una función para decidir si un grafo es conexo y todos los vértices tienen grado par ([ejercicio 15.16](#)), y en caso afirmativo construir e imprimir un ciclo de Euler.

*Sugerencia:* para borrar una arista o un vecino de la lista de vecinos usar la función **sacar** del [ejercicio 10.12.c](#)), y para «pegar» ciclos podría usarse una construcción del tipo `ciclo1[:i-1] + ciclo2 + ciclo1[i+1:]`. 

## 15.5. Comentarios

- La presentación de la función **recorrido** y del [algoritmo 15.3](#) están basadas en la de [Papadimitriou y Steiglitz \(1998\)](#).



## Capítulo 16

# Cálculo numérico elemental

Una de las aplicaciones más importantes de la computadora (y a la cual debe su nombre) es la obtención de resultados numéricos.<sup>(1)</sup> A veces es sencillo obtener los resultados deseados pero otras —debido a lo complicado del problema o a los errores numéricos— es sumamente difícil, lo que ha dado lugar a toda un área de las matemáticas llamada *cálculo numérico*.

Sorprendentemente, al trabajar con números decimales (`float`) pueden pasar cosas como:

- `a + b == a` aún cuando `b > 0`,
- `(a + b) + c != a + (b + c)`, o sea, la suma no es asociativa.

En este capítulo empezamos mirando a estos inconvenientes, para pasar luego a ver técnicas efectivas de resolución de problemas.

### 16.1. La codificación de decimales

Como vimos al calcular  $876^{123}$  en el [ejercicio 3.17](#), Python puede trabajar con cualquier número entero (sujeto a la memoria de la compu-

---

<sup>(1)</sup> En España en vez de *computadora* se la llama *ordenador*, destacando otras aplicaciones fundamentales que consideramos en otros capítulos.

tadora) pero no con todos los números decimales. Para los últimos usa una cantidad fija de bits (por ejemplo 64) divididos en dos grupos, uno representando la *mantisa* y otro el *exponente* como se hace en la notación *científica* al escribir  $0.123 \times 10^{45}$  (0.123 es la mantisa y 45 el exponente en base 10, pero la computadora trabaja en base 2).

Python usa una estructura especial que le permite trabajar con enteros de cualquier tamaño, si es necesario fraccionándolos primero para que la computadora haga las operaciones y luego rearmándolos, procesos que toman su tiempo.

Por cuestiones prácticas, se decidió no hacer algo similar con los decimales, que también mediante fraccionamiento y rearmado podrían tener tanta precisión como se quisiera. Esto es lo que hace el módulo estándar *decimal*, que no veremos en el curso.

Es decir, la computadora trabaja con números decimales que se expresan exactamente como suma de potencias de 2, y sólo unos pocos de ellos porque usa un número fijo de bits. Así, si usamos 64 bits para representar los números decimales, tendremos disponibles  $2^{64} \approx 1.845 \times 10^{19}$ , que parece mucho pero ¡estamos lejos de poder representar a todos los racionales!

Podríamos representar a números racionales de la forma  $a/b$  como un par  $(a, b)$  (como en el [ejercicio 8.14](#) o el módulo estándar *fractions* que no vemos), pero de cualquier forma nos faltan los irracionales, a los que sólo podemos aproximar.

Peor, un mismo número tiene distintas representaciones y entonces se representan menos de  $2^{64}$  números. Por ejemplo, pensando en base 10 (y no en base 2), podemos poner  $0.001 = 0.001 \times 10^0 = 1.0 \times 10^{-3} = 100.0 \times 10^{-5}$ , donde los exponentes son 0,  $-3$  y  $-5$ , y las mantisas son 0.001, 1.0 y 100.0, respectivamente.

Se hace necesario, entonces, establecer una forma de normalizar la representación para saber de qué estamos hablando. Cuando la parte entera de la mantisa tiene (exactamente) una cifra no nula, decimos que la representación es *normal*, por ejemplo  $1.0 \times 10^{-3}$  está en forma normal, pero no  $100.0 \times 10^{-5}$  ni 0.001.

Los números grandes o irracionales no son los únicos que no se puedan representar. Como vemos en el siguiente ejercicio, hay muchos números «comunes» que no se pueden representar porque no son sumas (finitas) de potencias de 2.

**E 16.1.** Un número real  $x$ ,  $0 < x < 1$ , se puede escribir como suma finita de potencias de 2 si existen coeficientes  $b_1, b_2, \dots, b_n$ ,  $b_i \in \{0, 1\}$ , tales que

$$x = \sum_{i=1}^n \frac{b_i}{2^i} = a \times 2^{-n} = \frac{a}{2^n}, \quad (a \in \mathbb{N})$$

donde hemos tomado factor común  $2^{-n}$ . Por ejemplo, 0.5 se puede representar de esta forma pues  $0.5 = 1 \times 2^{-1}$ .

- Ver que 0.1, 0.2 y 0.7 no pueden ponerse como sumas finitas de potencias de 2. En particular, no pueden representarse exactamente como decimales (**float**) en Python.
- Como las representaciones no son exactas, hay errores en las asignaciones **a = 0.1**, **b = 0.2** y **c = 0.7**.

Efectivamente, **a + b + c** y **b + c + a** dan valores distintos, si bien muy parecidos. Compararlos también con **==**. 📌

La representación de números decimales mediante mantisa y exponente hace que —a diferencia de lo que sucede con los números enteros— la distancia entre un número decimal que se puede representar y el próximo vaya aumentando a medida que sus valores absolutos aumentan, propiedad que llamamos de *densidad variable*.

- 📌 En matemáticas no hay un número real (en  $\mathbb{R}$ ) que sea el siguiente de otro.
- 📌 Intuitivamente, la densidad es la cantidad de números representables que hay en un intervalo.

Para entender la densidad variable, puede pensarse que hay la misma cantidad de números decimales representados entre 1 (inclusive) y 2 (exclusive) que entre 2 y 4, o que entre 4 y 8, etc. (las sucesivas

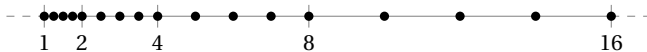


Figura 16.1: Esquema de la densidad variable en la codificación de números decimales.

potencias de 2). Por ejemplo, si hubieran sólo 4 números en cada uno de estos intervalos, tendríamos un gráfico como el de la [figura 16.1](#).

Por el contrario, hay tantos números enteros representados entre 10 (inclusive) y 20 (exclusive), como entre 20 y 30, etc. Es decir, entre 20 y 40 hay el *doble* de números enteros representados que entre 10 y 20. En este caso, la densidad es *constante*.

**E 16.2.** Trabajando con números (en  $\mathbb{R}$ ) en matemáticas, nunca puede ser  $a + 1 = a$ . La máquina piensa las cosas en forma distinta, y nuestro primer objetivo será encontrar una potencia de 2, **a**, tal que al escribirla como decimal resulta **a == 1 + a**.

Ponemos:

```
a = 1.0                # y no a = 1 !!!
while 1 + a > a:
    a = 2 * a
a
a + 1 > a
a + 2 > a
```



- Si en vez de **a = 1.0** ponemos **a = 1** inicialmente, tendremos un lazo infinito (¿por qué?).
- Encontrar  $n$  tal que el valor de **a** es  $2^n$  de dos formas: usando logaritmos y poniendo un contador en el lazo **while**.
- Ver que el valor de **a** es efectivamente el mismo de **a + 1**, pero distinto de **a - 1**: **a - 1** es el mayor decimal en Python tal que sumándole 1 da un número distinto.
- Calcular **a + i** para **i** entre 0 y 6 (inclusive). ¿Son razonables los resultados?

**E 16.3 ( $\varepsilon_{\text{máq}}$ ).** a) Esencialmente dividiendo por  $a > 0$  la desigualdad  $a + 1 > a$  en el [ejercicio 16.2](#) podemos hacer:

```
b = 1.0
while 1 + b > 1:
    b = b / 2
b = 2 * b    # nos pasamos: volver para atrás
```

☞ Ahora no importa si ponemos  $b = 1$  o  $b = 1.0$  pues la división por 2 dará un número decimal (`float`).

$b$  es el *épsilon de máquina*, que indicamos por  $\varepsilon_{\text{máq}}$  en matemáticas y por `epsmaq` en Python.

b) Al principio del ejercicio dijimos que básicamente calculábamos  $1/a$ , donde  $a$  es el valor calculado en el [ejercicio 16.2](#). ¿Es cierto que `epsmaq` es aproximadamente  $1/a$ ?, ¿qué relación hay entre estos números y  $\varepsilon_{\text{máq}}$ ?

*Sugerencia:* multiplicar  $a$  y `epsmaq`.



Podemos interpretar  $\varepsilon_{\text{máq}}$  de varias formas equivalentes:

- $\varepsilon_{\text{máq}}$  es la menor potencia (negativa) de 2 que sumada a 1 da mayor que 1,
- $1 + \varepsilon_{\text{máq}}$  es el número que sigue a 1 (para Python),
- $\varepsilon_{\text{máq}}$  es la distancia entre números en el intervalo  $[1, 2]$  (según comentamos al hablar de la densidad variable).

Desde ya que:

- Como mencionamos, en matemáticas no hay un número real que sea el siguiente de otro. La existencia de  $\varepsilon_{\text{máq}}$  refleja limitaciones de la computadora que no puede representar todos los números.
- El valor de  $\varepsilon_{\text{máq}}$  varía entre computadoras, sistemas operativos y lenguajes de programación.

**E 16.4 ( $\varepsilon_{\text{mín}}$ ).** Otro indicador importante es  $\varepsilon_{\text{mín}}$ , el *épsilon mínimo*, que es el menor número decimal positivo que se puede representar.

- ✎ Podemos pensar que *para la computadora*  $\varepsilon_{\min}$  es el decimal que sigue a 0 (y no hay otros en el medio). Ya sabemos que en los reales ( $\mathbb{R}$ ) no existe tal número.

- a) Decir por qué falla el esquema usado anteriormente:

```
x = 1.0
while x > 0:
    x = x / 2
x = 2 * x    # nos pasamos: volver al anterior
```

- b) Para conservar el último número no nulo, podemos poner:

```
x = 1.0:
while x > 0:
    x, anterior = x/2, x
# acá anterior es el menor decimal positivo
epsmin = anterior
```

Calcular el valor de  $\varepsilon_{\min}$  usando este esquema.



La función **epsi** (en el módulo **decimales**) generaliza el cálculo anterior de  $\varepsilon_{\max}$  y  $\varepsilon_{\min}$ . **epsi** toma como argumento el valor **inic**, que suponemos no negativo, y retorna la menor potencia de 2 que sumada a **inic** es mayor que **inic**. Es decir, **epsmaq** es **epsi(1.0)** y **epsmin** coincide con **epsi(0.0)**.

**E 16.5.** La función **epsi** nos ayudará a entender la densidad variable.

- a) Comparar los valores de **epsi** con argumentos  $1, 1 + 1/2, 1 + 3/4, \dots, 2 - 2^{-9}$ , y luego con argumentos  $2, 3, 3 + 1/2, \dots, 4 - 2^{-8}$ .

✎ Observar que  $2 = 2 \times 1, 3 = 2 \times (1 + 1/2), \dots, 4 - 2^{-j} = 2 \times (2 - 2^{-j-1}), \dots$

- b) Evaluar  $2^{**k} / \text{epsi}(2^{**k})$  para  $k = 0, \dots, 9$ , y ver que es una potencia de 2 (¿cuál?).
- c) Recordando que **epsmaq** es **epsi(1.0)**, ¿qué relación entre  $\varepsilon_{\max}$  y los valores del apartado anterior?



**E 16.6 (números grandes).** Tratemos de encontrar la mayor potencia de 2 que se puede representar en Python como número decimal.

- a) Recordando lo hecho en el [ejercicio 3.17](#), ponemos

```
a = 876 ** 123 # no hay problemas con enteros
float(a)        # da error
876.0 ** 123    # da error
```

obteniendo errores de *overflow* (*desborde* o *exceso*), es decir, que Python no ha podido hacer la operación pues se trata de números decimales grandes.

✎ Ya hemos visto (ejercicios [3.23](#), [5.6](#) o [8.8](#)) que `a` tiene 362 cifras en base 10.

- b) Haciendo un procedimiento similar a los que vimos en los ejercicios anteriores, y esperando tener un error (como en el [apartado a\)](#)) o que no termine nunca, cruzando los dedos ponemos:

```
x = 1.0
while 2 * x > x:
    x = 2 * x
x
```

Como en el [ejercicio 16.2](#), es crucial poner `x = 1.0` (y no `x = 1`).

☞ Cuando Python encuentra un número decimal muy grande que no puede representar o bien da *overflow* (como vimos antes) o bien lo indica con `inf` por infinito, pero hay que tener cuidado que no es el «infinito que conocemos».

- c) Usando la técnica para calcular  $\varepsilon_{\min}$ , calcular `maxpot2`, la máxima potencia de 2 que se puede representar en Python.
- d) Encontrar `n` tal que `maxpot2` es aproximadamente `2**n`.
- e) En analogía con el [ejercicio 16.3.b\)](#), averiguar la relación entre `epsmin` y `maxpot2`.



**E 16.7.** En este ejercicio miramos al número decimal más grande que se puede representar en Python (sin llegar a `inf`).

- a) Si  $x_0 = 2^n$  es la máxima potencia de 2 que se puede representar, lo que hemos llamado `maxpot2`, considerar las sumas finitas de potencias de 2

$$x_m = \sum_{j=0}^m 2^{n-j},$$

y encontrar el máximo  $m$  posible tal que  $x_m$  se representa como número decimal en Python. Llamaremos a este número `maxnum`.

✎ En mi máquina resulta  $m = 53$ ,  $x_m = 1.7976931348623157 \times 10^{308}$ .

- b) ¿Qué relación hay entre `epsi(maxpot2)` y `epsi(maxnum)`? ¶

En el módulo `decimales` incluimos los cálculos de `maxpot2` y `maxnum`.

**E 16.8.** En este ejercicio miramos el comportamiento de `inf` en las desigualdades.

- a) Usando el valor `n` obtenido en el [ejercicio 16.6.d](#)), ver que poniendo

```
| y = 2**(n + 1) - 1
```

no hay error porque es un número entero.

✎ `float(y)` no se puede representar en Python, aún cuando es suma de potencias de 2 menores que  $2^n$ ,  $y = 2^{n+1} - 1 = \sum_{j=0}^n 2^{n-j}$ . Comparar con  $x_m$  en el [ejercicio 16.7](#).

- b) Poniendo `x = maxpot2`, y considerando desde las matemáticas los valores de `n` y `y`, tendríamos (usando que  $n > 1$ ),

$$\begin{aligned} x = 2^n &< y = 2^{n+1} - 1 < 2^{n+1} = 2x \\ &< 2^{n+1} + 2^{n+1} - 2 = 2(2^{n+1} - 1) = 2y, \end{aligned}$$

pero Python piensa distinto:

```

2 * x          # -> inf
x < y          # -> verdadero
y < 2 * x      # -> verdadero
2 * x < 2 * y  # -> falso
2 * x > 2 * y  # -> verdadero

```

- c) `inf` no es un número que podemos asignar directamente (como lo son `2` o `math.pi`), para obtenerlo podemos pasar una cadena a decimal:

```

a = inf          # -> error
a = float('infinity') # o float('inf')
b = 2 * a        # -> inf
a == b           # -> verdadero

```

En conclusión:

- ☞ *Python a veces da el valor `inf` cuando hace cálculos con decimales, pero sólo números no demasiado grandes se deben comparar con `inf`.*



**E 16.9.** Recordando lo expresado al principio de la sección y como repaso de lo visto, para cada caso dar valores de `x`, `y` y `z` tales que los siguientes den verdadero:

- $x + y == x$  con `y` positivo.
- $(x + y) + z != x + (y + z)$ .
- $x + y + z != y + z + x$ .



## 16.2. Errores numéricos

Con la experiencia de la sección anterior, tenemos que ser cuidadosos con los algoritmos, sobre todo cuando comparamos por igualdad números decimales parecidos.

En el [ejercicio 16.1](#) vimos que podemos tener  $a + b + c \neq b + c + a$ , pero en ese caso los resultados eran parecidos. En los próximos ejerci-

cios vemos cómo estos pequeños errores pueden llevar a conclusiones desastrosas.

Retomemos el ejemplo del máximo común divisor y mínimo común múltiplo que vimos en la [sección 8.3](#) (página 83).

En el [ejercicio 8.14](#) propusimos una entrada compleja, dando tres datos enteros para Pablito y otros tantos para su papá, cuando en principio bastaría tomar (por ejemplo) los metros recorridos por paso para cada uno, en total, dos datos reales. La complejidad de la entrada provino de tratar de trabajar con números enteros, cuando lo natural en este problema es trabajar con decimales.

✎ Recordemos que los números reales  $a$  y  $b$  son conmensurables entre sí si existe un real positivo  $c$  y enteros  $m$  y  $n$  tales que  $a = mc$  y  $b = nc$ . Siguiendo la nomenclatura de Euclides, cuando todos los números involucrados son positivos, diríamos que  $c$  es una «medida común» para  $a$  y  $b$ .

En forma equivalente, podríamos decir que  $a$  y  $b$  son conmensurables si  $a/b$  es racional.

En las Proposiciones 1 y 2 del Libro VII de *Los Elementos*, Euclides presenta el algoritmo para calcular el máximo común divisor entre dos enteros positivos, y en el Libro X, Proposición 3, enuncia un resultado similar cuando los números (positivos pero no necesariamente enteros) son conmensurables entre sí.

**E 16.10 (de vuelta con Euclides).** En el módulo [euclides2](#) volvemos a considerar el algoritmo de Euclides, ahora con un lazo **for** del cual salimos eventualmente con **break**.


Similares a las versiones que vimos en el [capítulo 8](#) (ejercicios [8.9](#) y [8.10](#)), [mcd1](#) calcula el máximo común divisor usando restas sucesivas y terminando cuando **a** y **b** son iguales, mientras que [mcd2](#) usa restos y termina cuando **b** se anula.

- Estudiar las funciones [mcd1](#) y [mcd2](#), y ver que se obtienen resultados esperados con los argumentos [315](#) y [216](#).
- En principio no habría problemas en considerar como argumentos [3.15](#) y [2.16](#) para [mcd1](#) y [mcd2](#): los resultados deberían


ser como los anteriores sólo que divididos por 100 (es decir, 0.09).

Ver que esto no es cierto: para `mcd1` se alcanza el máximo número de iteraciones (1000) y los valores finales de `a` y `b` son muy distintos (y queremos que sean iguales), mientras que para `mcd2` los valores finales de `a` y `b` son demasiado pequeños comparados con la solución 0.09.

Explorar las causas de los problemas, por ejemplo imprimiendo los 10 primeros valores de `a` y `b` en cada función.

- c) Repetir `a)` con las entradas 6125 y 4500 y luego `b)` con las entradas 6.125 y 4.5. ¿Qué está pasando?
- d) En las funciones `mcd3` y `mcd4` evitamos las comparaciones directas, incorporando una *tolerancia* o *error permitido*. Ver que estas funciones dan resultados razonables para las entradas anteriores. 

Llegamos a una regla de oro en cálculo numérico:



*Nunca deben compararse números decimales por igualdad sino por diferencias suficientemente pequeñas.*

- ✎ Para calcular  $\varepsilon_{\text{máq}}$ ,  $\varepsilon_{\text{mín}}$  y otros números de la [sección 16.1](#) justamente violamos esta regla: queríamos ver hasta dónde se puede llegar (y nos topamos con incoherencias).
- ✎ Exactamente qué tolerancia usar en cada caso es complicado, y está relacionado con las diferencias conceptuales entre *error absoluto* y *relativo*, (a su vez relacionados con  $\varepsilon_{\text{mín}}$  y  $\varepsilon_{\text{máq}}$ , respectivamente, y más generalmente con la función `epsi`). Estas diferencias se estudian en cursos de estadística, física o análisis numérico, y no lo haremos acá.

Nos contentaremos con poner una tolerancia «razonable», generalmente del orden de  $\sqrt{\varepsilon_{\text{máq}}}$ .

**E 16.11 (problemas con la ecuación cuadrática).** Como sabemos, la ecuación cuadrática


$$ax^2 + bx + c = 0 \quad (16.1)$$

donde  $a, b, c \in \mathbb{R}$  son datos con  $a \neq 0$ , tiene soluciones reales si

$$d = b^2 - 4ac$$

no es negativo, y están dadas por

$$x_1 = \frac{-b + \sqrt{d}}{2a} \quad \text{y} \quad x_2 = \frac{-b - \sqrt{d}}{2a}. \quad (16.2)$$

- a) Definir una función que, dados  $a, b$  y  $c$ , verifique si  $a \neq 0$  y  $d \geq 0$ , poniendo un aviso en caso contrario, y en caso afirmativo calcule  $x_1$  y  $x_2$  usando las [ecuaciones \(16.2\)](#), y también  $ax_i^2 + bx_i + c$ ,  $i = 1, 2$ , viendo cuán cerca están de 0.
- b) Cuando  $d \approx b^2$ , es decir, cuando  $|4ac| \ll b^2$ , pueden surgir inconvenientes numéricos. Por ejemplo, calcular las raíces usando la función del apartado anterior, cuando  $a = 1$ ,  $b = 10^{10}$  y  $c = 1$ , verificando si se satisface la [ecuación \(16.1\)](#) en cada caso. 

Como en el caso de las sumas de Gauss ([ejercicio 10.8](#)), los dos ejercicios que siguen muestran que a veces que la teoría pueda ayudar mucho en los cálculos prácticos.

**E 16.12 (la pelota elástica).** Una pelota *muy* elástica va y viene rebotando entre dos paredes que distan un metro entre sí. Cada vez que rebota, la pelota incrementa su velocidad en 1 m/s, siendo su velocidad inicial de 1 m/s.

Suponiendo que:

- la velocidad permanece constante entre un rebote y otro,
- no hay demoras al dar la vuelta, el cambio de dirección es instantáneo,

- la velocidad de la luz es de 300 000 km/seg, y
- no tenemos en cuenta la teoría de la relatividad,

resolver:

- ¿Cuántas veces deberá rebotar la pelota para llegar a la velocidad de la luz?
- Dar una expresión matemática para el tiempo que la pelota tardará en llegar a la velocidad de la luz.
- Calcular con Python la expresión anterior de dos formas: sumando los términos de mayor a menor y sumando los términos de menor a mayor (el cálculo puede tardar un rato). ¿Dan el mismo resultado?

✎ En mi máquina, los valores son diferentes.



**E 16.13 (números armónicos).** El número que aparece en el [ejercicio 16.12.c](#)) es el *número armónico de orden  $n$* ,

$$h_n = \sum_{k=1}^n \frac{1}{k} \quad (n \in \mathbb{N}).$$

Hacia 1734, Euler demostró que a medida que  $n$  aumenta,

$$h_n \approx \log n + \gamma, \tag{16.3}$$


donde  $\gamma$  es la *constante de Euler-Mascheroni*,

$$\gamma = 0.577215664901 \dots$$

- ✎ (16.3) dice que para  $n$  grande,  $h_n$  se comporta como  $\log n$ , y por lo tanto crece muy lentamente. Por ejemplo, si queremos tener  $h_n \geq 200$ , habrá que tomar

$$n \approx e^{200} = 7.22597 \dots \times 10^{86},$$

cuando se estima que hay menos de  $10^{81}$  átomos en el universo observable.

- a) Verificar la validez de la [aproximación \(16.3\)](#) para  $n = 100, 1\,000$  y  $10\,000$ .
- b) Calcular el valor en el [ejercicio 16.12.c\)](#) usando la constante  $\gamma$ , y comparar con los resultados obtenidos allí. 

✎ El ejercicio [ejercicio 16.12](#) seguramente fue pensado por un matemático: totalmente impracticable.

Lo interesante es que muestra al desnudo el crecimiento exponencial ( $\log x$  es la inversa de  $e^x$ ): para llegar a una velocidad ridículamente alta se necesita un tiempo ridículamente corto.

## 16.3. Métodos iterativos: puntos fijos

Una de las herramientas más poderosas en matemáticas, tanto para aplicaciones teóricas como prácticas —en este caso gracias a la capacidad de repetición de la computadora— son los métodos iterativos. Casi todas las funciones matemáticas no elementales como logaritmos, exponenciales o trigonométricas, son calculadas por la computadora mediante estos métodos.

Pero, ¿qué es *iterar*?: repetir una serie de pasos. Por ejemplo, muchas calculadoras elementales pueden calcular la raíz cuadrada del número que aparece en el visor. En una calculadora con esta posibilidad, ingresando cualquier número (positivo), y apretando varias veces la tecla de «raíz cuadrada» puede observarse que rápidamente el resultado se aproxima o *converge* al mismo número, independientemente del valor ingresado inicialmente.

Hagamos este trabajo en la computadora.

### E 16.14 (punto fijo de la raíz cuadrada).


- a) Utilizando la construcción

```
y = x
for i in range(n):
    y = math.sqrt(y)
```

definir una función que tomando como argumentos  $x$  positivo y  $n$  natural, calcule

$$\underbrace{\sqrt{\sqrt{\dots\sqrt{\sqrt{x}}}}}_{n \text{ raíces}} \quad (= x^{1/2^n}),$$

imprimiendo los resultados intermedios.

- b) Ejecutar la función para distintos valores de  $x$  positivo, y  $n$  más o menos grande dependiendo de  $x$ . ¿Qué se observa? 


En el ejercicio anterior vemos que a medida que aumentamos el número de iteraciones (el valor de  $n$ ) nos aproximamos cada vez más a 1. Por supuesto que si empezamos con  $x = 1$ , obtendremos siempre el mismo 1 como resultado, ya que  $\sqrt{1} = 1$ .


Cuando un punto  $x$  en el dominio de la función  $f$  es tal que

$$f(x) = x,$$

decimos que  $x$  es un *punto fijo de  $f$* , de modo que 1 es un punto fijo de la función  $f(x) = \sqrt{x}$ .

Visualmente se pueden determinar los puntos fijos como los de la intersección del gráfico de la función con la diagonal  $y = x$ , como se ilustra en la [figura 16.2](#) (izquierda) cuando  $f(x) = \cos x$ .

**E 16.15.** Haciendo un gráfico combinado de  $f(x) = \sqrt{x}$  y de  $y = x$  para  $0 \leq x \leq 4$ , encontrar otro punto fijo de  $f$  (distinto de 1). 

**E 16.16.** Repetir los ejercicios anteriores considerando  $f(x) = x^2$  en vez de  $f = \sqrt{x}$ . ¿Cuáles son los puntos fijos?, ¿qué pasa cuando aplicamos repetidas veces  $f$  comenzando desde  $x = 0, 0.5, 1$  o 2? 

En lo que resta de la sección y el capítulo, trabajaremos con funciones *continuas*, intuitivamente funciones que «pueden dibujarse sin levantar el lápiz del papel». Ejemplos de funciones continuas son: cualquier polinomio,  $|x|$ ,  $\cos x$ , y  $\sqrt{x}$  (para  $x \geq 0$ ).



✎ Suponemos conocidas las definiciones de función continua y de función derivable, que generalmente se da en los cursos de análisis o cálculo matemático.

✎ Desde ya que hay funciones continuas que «no se pueden dibujar»:

- $1/x$  para  $x > 0$ , pues cerca de  $x = 0$  se nos acaba el papel,
- $\sin 1/x$  para  $x > 0$ , pues cerca de  $x = 0$  oscila demasiado,
- 

$$f(x) = \begin{cases} x \sin 1/x & \text{si } x \neq 0, \\ 0 & \text{si } x = 0 \end{cases}$$

pues cerca de  $x = 0$  oscila demasiado,

y hay funciones que no son continuas como

- $\text{signo}(x)$  para  $x \in \mathbb{R}$ , pues «pega un salto» en  $x = 0$ ,
- la función de Dirichlet

$$f(x) = \begin{cases} 1 & \text{si } x \in \mathbb{Q}, \\ 0 & \text{si } x \in \mathbb{R} \setminus \mathbb{Q}, \end{cases}$$

que es imposible de visualizar.

Muchas funciones de importancia teórica y práctica tienen puntos fijos con propiedades similares a la raíz cuadrada y 1.

Supongamos que  $x_0$  es un punto dado o *inicial* y definimos

$$x_1 = f(x_0), \quad x_2 = f(x_1), \quad \dots, \quad x_n = f(x_{n-1}), \quad \dots$$

y supongamos que tenemos la suerte que  $x_n$  se aproxima o *converge* al número  $\ell$  a medida que  $n$  crece, es decir,

$$x_n \approx \ell \quad \text{cuando } n \text{ es muy grande.}$$

Puede demostrarse entonces que, si  $f$  es continua,  $\ell$  es un punto fijo de  $f$ .

Por ejemplo, supongamos que queremos encontrar  $x$  tal que  $\cos x = x$ . Mirando el gráfico a la izquierda en la [figura 16.2](#), vemos que efectivamente hay un punto fijo de  $f(x) = \cos x$ , y podemos apreciar que el punto buscado está entre 0.5 y 1.

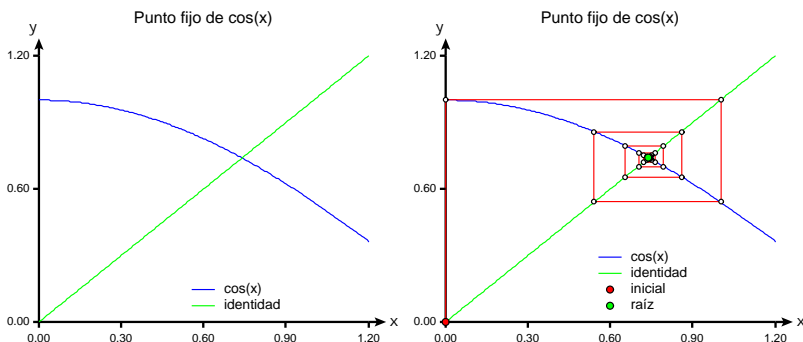


Figura 16.2: Gráfico de  $\cos x$  y  $x$  (izquierda) y convergencia a punto fijo desde  $x_0 = 0$  (derecha).

Probando la técnica mencionada, dado  $x_0 \in \mathbb{R}$  definimos

$$x_{n+1} = \cos x_n \quad \text{para } n = 0, 1, 2, \dots,$$

y tratamos de ver si  $x_n$  se aproxima a algún punto cuando  $n$  crece. A la derecha en la [figura 16.2](#) vemos cómo a partir de  $x_0 = 0$ , nos vamos aproximando al punto fijo, donde los trazos horizontales van desde puntos en el gráfico de  $f$  a la diagonal  $y = x$  y los verticales vuelven al gráfico de  $f$ .

**E 16.17 (punto fijo de  $f(x) = \cos x$ ).** Con las notaciones anteriores para  $f(x) = \cos x$  y  $x_k$  ( $k \geq 0$ ):

- Usando un lazo **for**, construir una función que dados  $x_0$  y  $n$  calcule  $x_n$ , y observar el comportamiento para distintos valores de  $x_0$  y  $n$ .
- Modificar la función para que también imprima  $\cos x_n$  y comprobar que para  $n$  más o menos grande se tiene  $x_n \approx \cos x_n$ .
- Modificar la función para hacer 200 iteraciones, mostrando los resultados intermedios cada 10. Observar que después de cierta cantidad de iteraciones, los valores de  $x_k$  no varían.

- ✎  $x_{100}$  es una buena aproximación al único punto fijo de  $f(x) = \cos x$ , aún cuando puede ser que  $x_{100} \neq \cos x_{100}$  ( $= x_{101}$ ) debido a errores numéricos.
- ✎ En realidad, las «espirales cuadradas» indican que los valores teóricos de  $x_n$  oscilan alrededor de la solución, y si trabajáramos con aritmética exacta, *nunca* obtendríamos la solución.
- ✎ También es muy posible que la solución a  $\cos x = x$  sea un número que no se puede representar en la computadora, por ejemplo si es irracional.


Abundando en lo anterior, se puede demostrar que si  $x \neq 0$  es algebraico (raíz de un polinomio con coeficientes enteros, como  $\sqrt{2}$ ) entonces  $\cos x$  es trascendente (un irracional que no es algebraico, como  $\pi$ ). En particular, el punto fijo del coseno es trascendente (y no sólo irracional).

d) Modificar la función de modo de no hacer más iteraciones si


$$|x_{k+1} - x_k| < \varepsilon,$$

donde  $\varepsilon > 0$  es un nuevo argumento (e. g.,  $\varepsilon = 0.00001 = 10^{-5}$ ), aún cuando  $k$  sea menor que  $n$ .

*Sugerencia:* usar **break** en algún lugar adecuado.

- ✎ Observar que la condición  $|x_{k+1} - x_k| < \varepsilon$  es equivalente a  $|f(x_k) - x_k| < \varepsilon$ . 

**E 16.18.** La función **puntofijo** (en el módulo *numerico*) sintetiza lo hecho en el [ejercicio 16.17](#): retorna un punto fijo de la función con una tolerancia permitida en un número máximo de iteraciones.

- a) Modificar la función de modo que el número de iteraciones máximas y la tolerancia sean argumentos.
- b) Modificar la función de modo que siempre se imprima la cantidad de iteraciones realizadas y el error obtenido. 

Cuando usamos un método iterativo para obtener una solución aproximada (como en el caso de las iteraciones de punto fijo), es tradi-

cional considerar tres *criterios de parada*, saliendo del lazo cuando se cumple algunas de las siguientes condiciones:

- la diferencia en  $x$  es suficientemente pequeña, es decir,  $|x_{n+1} - x_n| < \varepsilon_x$ ,
- la diferencia en  $y$  es suficientemente pequeña, es decir,  $|f(x_{n+1}) - f(x_n)| < \varepsilon_y$ , o, en el caso de búsqueda de ceros, si  $|f(x_n)| < \varepsilon_y$ ,
- se ha llegado a un número máximo de iteraciones, es decir,  $n = n_{\text{máx}}$ ,


donde  $\varepsilon_x$ ,  $\varepsilon_y$  y  $n_{\text{máx}}$  son datos, ya sea como argumentos en la función o determinados en ella. En la función **puntofijo** consideramos dos de ellos (el segundo es casi equivalente al primero en este caso), pero en general los tres criterios son diferentes entre sí.

La importancia de los puntos fijos es que al encontrarlos estamos resolviendo la ecuación  $f(x) - x = 0$ . Así, si nos dan la función  $g$  y nos piden encontrar una raíz de la ecuación  $g(x) = 0$ , podemos definir  $f(x) = g(x) + x$  o  $f(x) = x - g(x)$  y tratar de encontrar un punto fijo para  $f$ .


Por ejemplo,  $\pi$  es una raíz de la ecuación  $\tan x = 0$ , y para obtener un valor aproximado de  $\pi$  podemos tomar  $g(x) = \tan x$ ,  $f(x) = x - \tan x$ , y usar la técnica anterior.

**E 16.19.** Resolver los siguientes apartados con la ayuda de gráficos para ver qué está sucediendo en cada caso.

- Encontrar (sin la compu) los puntos fijos de  $f(x) = x - \tan x$ , es decir, los  $x$  para los que  $f(x) = x$ , en términos de  $\pi$ . Ver que  $\pi$  es uno de los infinitos puntos fijos.
- Usando la función **puntofijo**, verificar que con 3 o 4 iteraciones se obtiene una muy buena aproximación de  $\pi$  comenzando desde  $x_0 = 3$ .
- Sin embargo, si empezamos desde  $x_0 = 1$ , nos aproximamos a 0, otro punto fijo de  $f$ .

- d)  $f(\pi/2)$  no está definida, y es previsible encontrar problemas cerca de este punto. Como  $\pi/2 \approx 1.5708$ , hacer una tabla de los valores obtenidos después de 10 iteraciones, comenzando desde los puntos 1.5, 1.51,  $\dots$ , 1.6 (desde 1.5 hasta 1.6 en incrementos de 0.01) para verificar el comportamiento.
- e) Si en vez de usar  $f(x) = x - \tan x$  usáramos  $f(x) = x + \tan x$ , los resultados en a) no varían. Hacer los apartados b) y c) con esta variante y verificar si se obtienen resultados similares. 

**E 16.20.** Puede suceder que las iteraciones tengan un comportamiento cíclico, por ejemplo al tomar  $f(x) = -x^3$  y  $x_0 = 1$ , y también las iteraciones pueden «dispararse al infinito», por ejemplo si tomamos  $f(x) = x^2$  y  $x_0 > 1$ , o hacerlo en forma oscilatoria, como con  $f(x) = -x^3$  y  $x_0 > 1$ .

Estudiar el comportamiento en los casos mencionados haciendo unas pocas iteraciones. 

Recordar entonces que:

*Un método iterativo puede no converger a una solución, o converger pero no a la solución esperada.*

## 16.4. El método de Newton y Raphson

Los métodos iterativos y en particular la técnica de punto fijo para encontrar raíces de ecuaciones no surgieron con las computadoras.

Por ejemplo, el *método babilónico* que vemos en el [ejercicio 16.21](#) es una técnica usada por los babilonios hace miles de años para aproximar a la raíz cuadrada, y resulta ser un caso particular de otro para funciones mucho más generales que estudiamos en esta sección.

Recordemos que la derivada de  $f$  en  $x$ ,  $f'(x)$ , se define como

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h},$$

es decir, si  $h \neq 0$  y  $|h|$  es suficientemente chico,

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \quad (16.4)$$

Si la derivada existe, o sea, si los cocientes incrementales (los que están a la derecha en (16.4)) se parecen cada vez más a algo a medida que  $|h|$  se hace más y más chico, decimos que la función es derivable (en  $x$ ), pero es posible que los cocientes no se parezcan a nada.

Intuitivamente,  $f$  es derivable en  $x$  cuando podemos trazar la recta tangente al gráfico de la curva en el punto  $(x, f(x))$ . Como basta dar la pendiente y un punto para definir una recta, para determinar la tangente en  $(x, f(x))$  basta dar su pendiente, y ésta es lo que se denomina  $f'(x)$ .

Supongamos ahora que  $f$  es una función derivable en todo punto, y que  $x^*$  es un cero de  $f$ . Si  $x$  es un punto próximo a  $x^*$ , digamos  $x^* = x + h$ , «despejando» en la [relación \(16.4\)](#), llegamos a

$$f(x+h) \approx f(x) + f'(x)h,$$

y como  $h = x^* - x$ , queda

$$0 = f(x^*) \approx f(x) + f'(x)(x^* - x).$$

Despejando ahora  $x^*$ , suponiendo  $f'(x) \neq 0$ , queda

$$x^* \approx x - \frac{f(x)}{f'(x)}.$$

Esto establece un método iterativo, *el método de Newton-Raphson*, considerando la sucesión

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{para } n = 0, 1, \dots, \quad (16.5)$$

siempre que  $f'$  no se anule en los puntos  $x_n$ .

La **ecuación (16.5)** nos dice que el método, que a partir de ahora llamaremos simplemente de Newton, es un método de punto fijo, que busca un punto fijo de la función

$$g(x) = x - \frac{f(x)}{f'(x)}. \quad (16.6)$$

**E 16.21 (método babilónico).** Supongamos que no sabemos cómo calcular la raíz cuadrada de un número y queremos encontrar  $\sqrt{a}$  para algún  $a$  positivo. Como decir que  $b = \sqrt{a}$  es (por definición) lo mismo que decir que  $b^2 = a$  y  $b \geq 0$ , tratamos de encontrar un cero de la función  $f(x) = x^2 - a$ .

- a) Encontrar la función  $g$  dada en la **ecuación (16.6)** si  $f(x) = x^2 - a$ .

Ayuda:  $f'(x) = 2x$ .

Respuesta:  $g(x) = (x + a/x)/2$ .

El método babilónico para aproximar  $\sqrt{a}$ , consiste en calcular sucesivamente las iteraciones


$$x_n = \frac{1}{2} \left( x_{n-1} + \frac{a}{x_{n-1}} \right) \quad \text{para } n = 1, 2, \dots, \quad (16.7)$$

a partir de un valor inicial  $x_0$  dado ( $x_0 > 0$ ). En otras palabras,  $x_n = g(x_{n-1})$ , donde  $g$  es la función encontrada en **a)**.

- b) Definir una función **babilonico(a, it)** implementando la **iteración (16.7)** a partir de  $x_0 = 1$ , haciendo  $it$  iteraciones.
- c) Calcular **babilonico(2, it)** para  $it \in \{4, 6, 8\}$ , y comparar los resultados con **math.sqrt(2)**.
- d) Comparar los resultados de **babilonico(a, 6)** para  $a = 2$  y  $a = 200$ .

⚠ Aproximar  $\sqrt{2}$  es equivalente a aproximar  $\sqrt{200}$ , sólo hay que correr en un lugar la coma decimal en la solución, pero en la función **babilonico** no lo tenemos en cuenta.

Es más razonable considerar primero únicamente números en el intervalo  $[1, 100]$ , encontrar la raíz cuadrada allí, y luego escalar adecuadamente. Este proceso de *normalización* o *escalado* es esencial en cálculo numérico: trabajar con papas y manzanas y no con átomos y planetas, o, más científicamente, con magnitudes del mismo orden.

Cuando se comparan papas con manzanas en la computadora, se tiene en cuenta el valor de  $\varepsilon_{\text{máq}}$ , pero al trabajar cerca del 0 hay que considerar a  $\varepsilon_{\text{mín}}$ . 

Difícilmente queramos encontrar ceros de una función tan sencilla como  $x^2 - a$ . La mayoría de las veces las funciones serán más complejas, posiblemente involucrando funciones trascendentes (como trigonométricas o exponenciales) y los coeficientes no se conocerán con exactitud. De modo que en general no se implementa el método de Newton como expresado en (16.5), sino que se reemplaza la derivada de la función por una aproximación numérica, evitando el cálculo de una fórmula de la derivada.

Para aproximar la derivada usamos


$$f'(x) \approx \frac{f(x + \Delta x/2) - f(x - \Delta x/2)}{\Delta x}, \quad (16.8)$$

donde  $\Delta x$  es una constante dada, ya que es mucha mejor aproximación que la dada en (16.4).

Podemos visualizar esta propiedad en la [figura 16.3](#). La tangente a la función es la recta en verde, y queremos calcular su pendiente que es la derivada.

La recta que pasa por  $(x, f(x))$  y  $(x+h, f(x+h))$  está en marrón, y la pendiente de esta recta es la [aproximación \(16.4\)](#).

En cambio, la recta por  $(x-h/2, f(x-h/2))$  y  $(x+h/2, f(x+h/2))$  está en rojo, y su pendiente se parece mucho más a la pendiente de la recta verde.

 Podemos justificar matemáticamente la propiedad usando el desarrollo de Taylor. Tendremos:

$$f(x+h) = f(x) + f'(x)h + f''(x)h^2/2 + O(h^3),$$



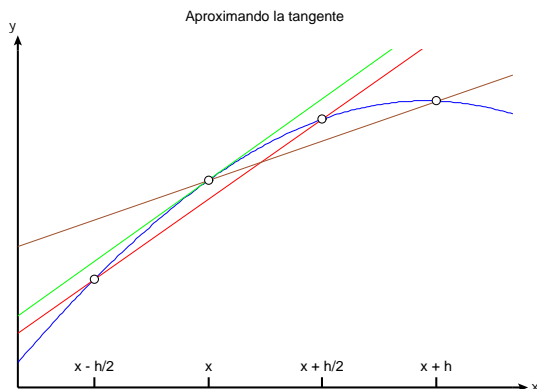


Figura 16.3: Aproximando la pendiente de la recta tangente.

de modo que el error para calcular  $f'(x)$  en (16.4) es

$$\frac{f(x+h) - f(x)}{h} - f'(x) = O(h),$$

mientras que el error en (16.8)

$$\frac{f(x+h) - f(x-h)}{2h} - f'(x) = O(h^2).$$

✎ La idea de usar derivadas aproximadas se extiende tomando  $h$  variable (acá lo tomamos fijo) dando lugar al *método secante* que es muy usado en la práctica como alternativa al de Newton, y que no veremos en el curso.

La función `newton` (en el módulo `numerico`) toma dos argumentos, la función  $f$  y el punto inicial  $x_0$ , y en ella implementamos el método de Newton, aproximando internamente la derivada de la función usando (16.8) con  $\Delta x = 10^{-7}$ .

✎ La elección  $\Delta x = 10^{-7}$  es arbitraria, y podría ser mucho más grande o chica dependiendo del problema. Ver los comentarios sobre errores absolutos y relativos en las notas de la [página 227](#).

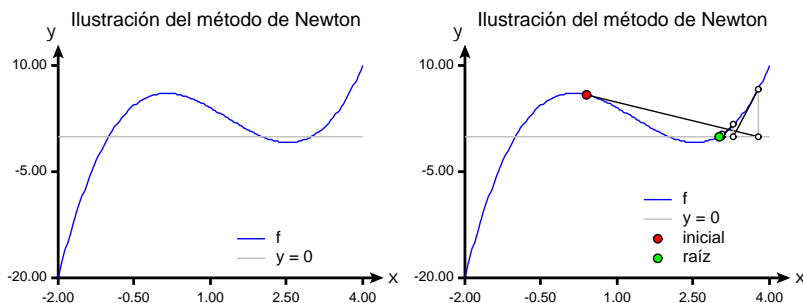


Figura 16.4: Gráfico de  $f$  en la ecuación (16.9) (izquierda) y convergencia del método de Newton desde  $x_0 = 0.4$  (derecha).




En la función `newton` no consideramos la posibilidad  $f'(x) = 0$ .

**E 16.22 (Newton con derivadas aproximadas).** Tomemos

$$f(x) = (x + 1)(x - 2)(x - 3), \quad (16.9)$$

que tiene ceros en  $-1, 2$  y  $3$  y se ilustra a la izquierda en la figura 16.4.

$f$  tiene derivadas que no son difíciles de calcular, pero de cualquier manera usamos la función `newton` tomando como punto inicial  $0.4$ , obteniendo las aproximaciones sucesivas que pueden observarse a la derecha de la figura 16.4.

Considerar otros puntos iniciales para obtener las otras dos raíces ( $-1$  y  $2$ ). 

**E 16.23.** Resolver los siguientes apartados con la ayuda de gráficos y la función `newton`.

- Encontrar soluciones aproximadas de la ecuación  $\cos x = 0$  tomando puntos iniciales  $1.0$  y  $0.0$ .
- Encontrar una solución aproximada de  $\cos x = x$  y comparar con los resultados del ejercicio 16.17.

- c) Encontrar aproximadamente todas las soluciones de las ecuaciones:

$$i) \ x^2 - 5x + 2 = 0 \qquad ii) \ x^3 - x^2 - 2x + 2 = 0.$$

Resolver también estas ecuaciones en forma exacta y comparar con los resultados obtenidos.

✎ La primera ecuación tiene 2 raíces y la segunda 3.

*Ayuda:* para las soluciones exactas usar (16.2) en el primer caso y en el segundo dividir primeramente por  $x - 1$ , ya que 1 es solución.

- d) Obtener una solución aproximada de cada una de las ecuaciones

$$i) \ 2 - \ln x = x, \qquad ii) \ x^3 \operatorname{sen} x + 1 = 0.$$

✎ La primera ecuación tiene una raíz y la segunda infinitas.



## 16.5. El método de la bisección

Supongamos que tenemos una función continua  $f$  definida sobre el intervalo  $[a, b]$  a valores reales, y que  $f(a)$  y  $f(b)$  tienen distinto signo, como la función  $f$  graficada en la [figura 16.5](#). Cuando la «dibujamos con el lápiz» desde el punto  $(a, f(a))$  hasta  $(b, f(b))$ , vemos que en algún momento cruzamos el eje  $x$ , y allí encontramos una raíz de  $f$ , es decir, un valor de  $x$  tal que  $f(x) = 0$ .

En el *método de la bisección* se comienza tomando  $a_0 = a$  y  $b_0 = b$ , y para  $i = 0, 1, 2, \dots$  se va dividiendo sucesivamente en dos el intervalo  $[a_i, b_i]$  tomando el punto medio  $c_i$ , y considerando como nuevo intervalo  $[a_{i+1}, b_{i+1}]$  al intervalo  $[a_i, c_i]$  o  $[c_i, b_i]$ , manteniendo la propiedad que en los extremos los signos de  $f$  son opuestos (que podemos expresar como  $f(a_i)f(b_i) < 0$ ), como se ilustra en la [figura 16.5](#).

✎ En los cursos de análisis o cálculo se demuestra que si  $f$  es continua en  $[a, b]$ , y tiene signos distintos en  $a$  y  $b$ , entonces  $f$  se

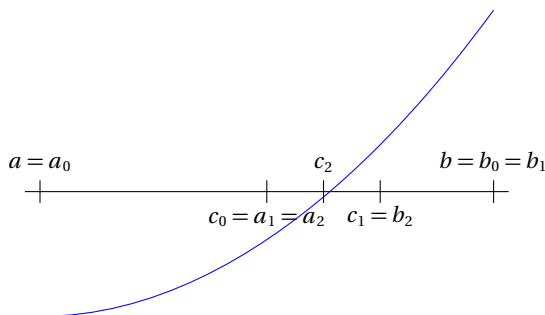


Figura 16.5: Función continua con distintos signos en los extremos.

anula en algún punto del intervalo.

Una forma de demostrar esta propiedad es con el método de la bisección usando la propiedad de completitud de los reales.

Se finaliza según algún criterio de parada (como mencionados en la [página 235](#)), por ejemplo cuando se obtiene un valor de  $x$  tal que  $|f(x)|$  es suficientemente chico o se han realizado un máximo de iteraciones.

- ✎ Recordando la filosofía de comparar papas con manzanas, la tolerancia a poner dependerá del problema que se trate de resolver.
- ✎ También en este sentido, observamos que  $2^{10} = 1024 \approx 10^3$  y  $2^{20} = 1048576 \approx 10^6$ , por lo que el intervalo inicial se divide aproximadamente en 1000 después de 10 iteraciones y en  $1000000 = 10^6$  después de 20 iteraciones. Es decir, después de 10 iteraciones el intervalo mide menos del 0.1% del intervalo original, y después de 20 iteraciones mide menos del 0.0001% del intervalo original. No tiene mucho sentido considerar muchas más iteraciones en este método, salvo que los datos originales y la función  $f$  puedan calcularse con mucha precisión y el problema amerite este cálculo.

La función **biseccion** (en el módulo [numerico](#)) utiliza el método de la bisección para encontrar raíces de una función dados dos puntos en los que la función toma distintos signos.

Observemos la estructura de la función:

1. Los extremos del intervalo inicial son *poco* y *mucho*.
2. En la inicialización, se calcula el valor de la función en el extremo *poco*, *fpoco*. Si este valor es suficientemente chico en valor absoluto, ya tenemos la raíz y salimos.
3. Procedemos de la misma forma con el extremo *mucho*, obteniendo *fmucho*.
4. Si la condición de distinto signo en los extremos no se satisface inicialmente, el método no es aplicable y salimos poniendo un cartel apropiado.
5. Arreglamos los valores *poco* y *mucho* de modo que  $fpoco < 0$ .
6. El lazo principal se realiza mientras no se haya encontrado solución:
  - Se calcula el punto medio del intervalo, *medio*, y el valor *fmedio* de la función en *medio*.
  - Si la diferencia entre *mucho* y *poco* es suficientemente chica, damos a *medio* como raíz.
  - Si el valor absoluto de *fmedio* es suficientemente chico, también damos a *medio* como raíz.
  - Si no, calculamos un nuevo intervalo, cuidando de que los signos en los extremos sean distintos.
7. El lazo principal no puede ser infinito, pues vamos reduciendo a la mitad el tamaño del intervalo en cada iteración.

**E 16.24 (método de la bisección).** Consideremos la función

$$f(x) = x(x+1)(x+2)(x-4/3),$$

ilustrada en la [figura 16.6](#) (izquierda), que tiene ceros en  $x = -2, -1, 0$  y  $4/3$ .

Usando el método de la bisección para esta función tomando como intervalo inicial  $[-1.2, 1.5]$ , obtenemos una sucesión de intervalos a

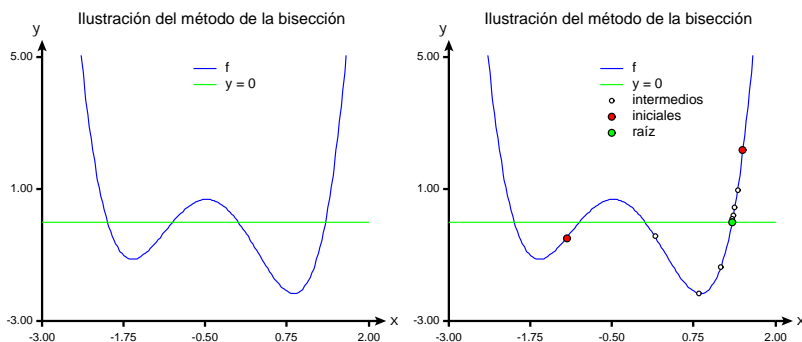


Figura 16.6: Método de bisección ([ejercicio 16.24](#)) función (izquierda) y puntos obtenidos para el intervalo inicial  $[-1.2, 1.5]$  (derecha).

partir de los puntos medios que se muestran en la misma figura a la derecha.

- a) Si hay más de una raíz en el intervalo inicial, la solución elegida por el algoritmo depende de los datos iniciales.

Verificar este comportamiento ejecutando **biseccion** sucesivamente con los valores 0.8, 1 y 1.2 para *mucho*, pero tomando *poco* = -3 en todos estos casos.

- b) ¿Por qué si ponemos *poco* = -3 y *mucho* = 1 obtenemos la raíz  $x = -1$  en una iteración?

✎ En general, *nunca* obtendremos el valor *exacto* de la raíz: para la computadora sólo existen unos pocos racionales.

- c) En **biseccion** no verificamos si *poco* < *mucho*, y podría suceder que *poco* > *mucho*. ¿Tiene esto importancia?
- d) Dividir la tolerancia en  $\varepsilon_x$  y  $\varepsilon_y$  (en vez de  $\varepsilon$ ), de modo de terminar las iteraciones si  $|\text{mucho} - \text{poco}| < \varepsilon_x$  o si  $|f(\text{medio})| < \varepsilon_y$ .
- e) Teniendo en cuenta lo expresado al principio de la sección, ¿tendría sentido agregar un criterio de modo de parar si el número de iteraciones es grande?

Si la cantidad máxima de iteraciones fuera  $n$ , ¿cuántas iteraciones deben realizarse para alcanzarla, en términos de  $\varepsilon_x$  y los valores originales de *poco* y *mucho*? ¶

El método de la bisección es bien general y permite encontrar las raíces de muchas funciones. No es tan rápido como el de Newton, pero para la convergencia de éste necesitamos que las derivadas permanezcan lejos de cero, que las derivadas segundas no sean demasiado «salvajes», y tomar un punto inicial adecuado.

Por supuesto que el método de la bisección y el de búsqueda binaria (en la [sección 12.4](#)) son esencialmente la misma cosa. Si tenemos una lista de números no decreciente  $a_0 \leq a_1 \leq \dots \leq a_n$ , uniendo los puntos  $(k-1, a_{k-1})$  con  $(k, a_k)$  para  $k = 1, \dots, n$ , tendremos el gráfico de una poligonal —y en particular una función continua— y aplicar el método de la bisección a esta poligonal es equivalente a usar búsqueda binaria, sólo que consideramos únicamente valores enteros de  $x$ , y por eso hacemos

$$\text{medio} = \left\lfloor \frac{\text{poco} + \text{mucho}}{2} \right\rfloor,$$

terminando en cuanto la diferencia en  $x$  es 1, que se traduce como

$$\text{mucho} - \text{poco} \leq 1.$$

**E 16.25.** Con la ayuda de gráficos apropiados y la función **biseccion**, usar el método de la bisección para resolver los apartados del [ejercicio 16.23](#), y comparar con las soluciones obtenidas allí. ¶

**E 16.26.** En los ejercicios [13.8](#) y [13.9](#) estudiamos funciones que relacionaban el monto inicial  $c = c_0$  de un préstamo, la tasa nominal anual  $r$  (como porcentaje) y el pago mensual  $p$ . Vimos cómo obtener  $c$  o  $p$  en términos de los otros dos parámetros cuando el número de cuotas es  $n = 12$ . En este ejercicio vamos a construir la función que nos falta: encontrar la tasa  $r \geq 0$  dados  $c$  y  $p$ .

En lo que sigue suponemos  $c = c_0 > 0$  y  $p (> 0)$  fijos, y empezamos estudiando la [ecuación \(13.12\)](#).

☞ Recordar también los comentarios en la [página 182](#).

- a) Ver que para cada  $n$  fijo (y manteniendo  $c_0$  y  $p$  fijos),  $c_n$  es creciente con  $t$ .

Por lo tanto  $c_{12}$  es creciente con  $r$ .

- b) Si  $r_{\text{crít}}$  ( $r$  crítico) es el valor de la tasa  $r$  tal que en la [ecuación \(13.12\)](#) el monto  $c_n$  es el mismo para todo  $n > 1$ , entonces

$$r_{\text{crít}} = 1200 \frac{p}{c}. \quad (16.10)$$

Dejando  $c = c_0$  ( $> 0$ ) y  $p$  ( $> 0$ ) fijos, definimos  $f(r)$  por la [expresión \(13.13\)](#) con  $n = 12$ :

$$f(r) = c_{12} = ct^{12} - p \times \sum_{j=0}^{11} t^j, \quad (16.11)$$

y observamos que, por el [apartado a\)](#),  $f$  es creciente con  $r$ .

- c) Como  $f(0) = c - 12p$ , si  $c > 12p$ , será  $f(r) > 0$  para todo  $r \geq 0$ , por lo tanto pedimos

$$c < 12p \quad \text{o sea} \quad f(0) < 0. \quad (16.12)$$

- d) Como  $f$  es creciente,  $f(0) < 0$  y  $f(r_{\text{crít}}) = c > 0$  (y  $f$  es continua), debe existir un único punto  $r$  tal que  $f(r) = 0$ .

Definir una función [tasa12\(c, p\)](#) para encontrar ese  $r$  usando el método de la bisección, y comprobar su comportamiento con los datos del [ejercicio 13.9](#). ☞

**E 16.27.** Mateo quiere actualizar su vieja computadora, y vio que en un catálogo ofrecían una que de contado cuesta \$ 13 499.10 y puede pagarse en 24 cuotas mensuales de \$ 1 554.93. ¿Cuál es la tasa nominal anual que está aplicando el comercio?

*Aclaración:* las compras en cuotas siguen el mismo mecanismo de los préstamos.

*Atención:* se trata de 24 y no de 12 cuotas. ☞



**E 16.28.** Para calcular la función **tasa12** en el **ejercicio 16.26** usamos el método de la bisección. Si bien tenemos la **fórmula explícita (16.11)** para calcular  $f(r)$  y podríamos calcular  $f'(r)$ , parece más sencillo usar directamente el método de Newton con derivadas aproximadas.

Definir una función **tasa12Newton** con estas ideas, y comparar los resultados con los obtenidos mediante **tasa12**.

✎ Un punto inicial razonable sería tomar  $r_{\text{crit}}/2$ .



## 16.6. Comentarios

- Los temas presentados en este capítulo no están tomados de ninguna fuente en particular: cualquier cita nos llevaría a cosas mucho más profundas de las que vimos.

Más aún, no hemos tocado en absoluto temas típicos de cursos de cálculo numérico como integración numérica, resolución de ecuaciones diferenciales o sistemas lineales.

- El **ejercicio 16.12** es una variante de uno que aparece en la versión inglesa de **Wikipedia**,<sup>(2)</sup> y desconozco su autor.
- El método de Newton o variantes se generaliza a varias variables (reemplazando adecuadamente la derivada unidimensional), e inclusive a variables complejas.

En cambio, es muy difícil generalizar el método de la bisección a más de una dimensión.

- Los métodos iterativos están íntimamente relacionados con los conjuntos fractales, como el de la **figura 16.7**.

Para hacer estos gráficos usamos el método de Newton buscando las raíces (complejas) del polinomio  $z^3 - 1$ , marcando con rojo (resp. azul o verde) los puntos que cuando tomados como iniciales el método converge hacia 1 (resp.  $-1/2 \pm i\sqrt{3}/2$ ).

---

<sup>(2)</sup> [https://en.wikipedia.org/wiki/Harmonic\\_series\\_\(mathematics\)](https://en.wikipedia.org/wiki/Harmonic_series_(mathematics))

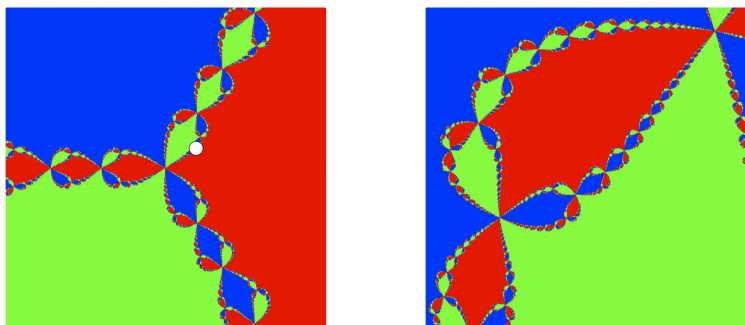


Figura 16.7: Dos imágenes del fractal asociado a la convergencia del método de Newton para  $z^3 - 1$ .

La función a iterar en este caso es

$$g(z) = \frac{2z + 1/z^2}{3},$$

que está definida para  $z \neq 0$ , y sobre la recta real todos los puntos están marcados en rojo (salvo 0).

A la izquierda de la figura vemos el gráfico en el cuadrado de centro 0 y lado 4, y a la derecha un detalle del cuadrado centrado en  $0.3765 + 0.2445i$  (punto resaltado a la izquierda en blanco) y lado 0.004.

El gráfico a la derecha es un «zoom» por un factor de 1000 del gráfico a la izquierda, lo que indica que las propiedades del gráfico no resultan de errores numéricos sino que son una propiedad intrínseca.

Este es un ejemplo extremo de lo expresado en el «cartel» de la [página 236](#), sobre cómo depende la solución obtenida del punto inicial.

- El muy difundido *conjunto de Mandelbrot*, reproducido en la [figura 16.8](#), también es un conjunto fractal relacionado con las

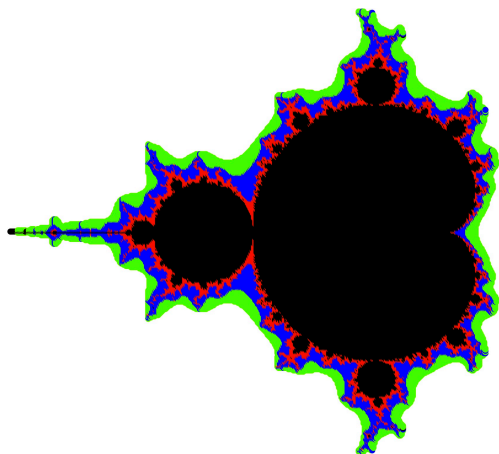


Figura 16.8: Ilustración del conjunto de Mandelbrot.

iteraciones: son los puntos  $c$  del plano complejo para los cuales tomando como punto inicial  $z_0 = 0$ , las iteraciones de  $f_c(z) = z^2 + c$  permanecen acotadas.

Más información sobre este conjunto se puede obtener en internet, por ejemplo en [Wikipedia](https://es.wikipedia.org/wiki/Conjunto_de_Mandelbrot).<sup>(3)</sup>



---

<sup>(3)</sup> [https://es.wikipedia.org/wiki/Conjunto\\_de\\_Mandelbrot](https://es.wikipedia.org/wiki/Conjunto_de_Mandelbrot)

## Capítulo 17

# Objetos combinatorios

### 17.1. Contando objetos combinatorios

Es posible dar una definición inductiva de los coeficientes binomiales, alternativa de la que vimos en (10.10):

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \text{y} \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n. \quad (17.1)$$

Recordemos que  $\binom{n}{k}$  representa la cantidad de subconjuntos con exactamente  $k$  elementos si el total tiene  $n$ . Así, si  $I_n = \{1, \dots, n\}$ , (17.1) puede interpretarse diciendo que que sólo hay un conjunto sin elementos, sólo hay un conjunto con todos los elementos, y un subconjunto de  $I_n$  que tiene  $k$  elementos ( $0 < k < n$ ), o bien contiene a  $n$  y sacándolo tenemos un subconjunto con  $k-1$  elementos de  $I_{n-1}$  o bien no contiene a  $n$  y entonces es un subconjunto con  $k$  elementos de  $I_{n-1}$ .

La última parte de la propiedad (17.1) da lugar al llamado *triángulo de Tartaglia* o *triángulo de Pascal*, que se obtiene poniendo un 1 en la primera fila, y luego en cada fila siguiente la suma de los elementos consecutivos de la fila anterior, empezando y terminando con 1, como se muestra en la figura 17.1.

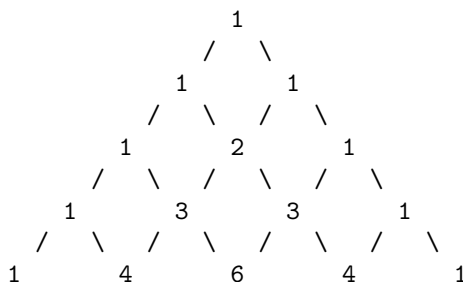


Figura 17.1: Triángulo de Tartaglia (o de Pascal) de nivel 4.

**E 17.1.** Definir una función recursiva para calcular  $\binom{n}{k}$  basada en las relaciones (17.1).

✎ Observar que en este caso *el paso base* es en realidad dos: cuando  $k = 0$  y cuando  $k = n$ . ¶

**E 17.2.** Para  $m, n \in \mathbb{N}$ , consideremos una cuadrícula rectangular de dimensiones  $m \times n$  ( $4 \times 3$  en la figura 17.2), e imaginémosnos que se trata de un mapa, donde los segmentos son calles y los puntos remarcados son las intersecciones.

Nos preguntamos de cuántas maneras podremos ir desde la esquina más hacia el sudoeste, de coordenadas  $(0, 0)$ , a la esquina más hacia el noreste, de coordenadas  $(m, n)$ , si estamos limitados a recorrer las calles únicamente en sentido oeste–este o sur–norte, según corresponda.

✎ Se trata de un grafo *dirigido*.

Para resolver el problema, podemos pensar que para llegar a una intersección hay que hacerlo desde el oeste o desde el sur (salvo cuando la intersección está en el borde oeste o sur), y por lo tanto la cantidad de caminos para llegar a la intersección es la suma de la cantidad de caminos llegando desde el oeste (si se puede) más la cantidad de caminos llegando desde el sur (si se puede). Los números en la figura 17.2 indican, para cada intersección, la cantidad de caminos para llegar allí

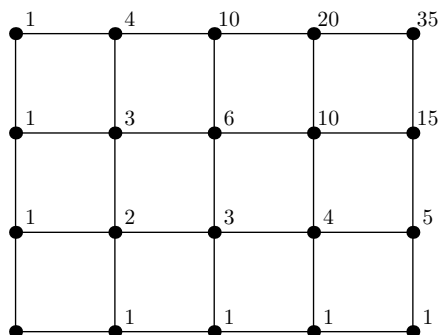


Figura 17.2: Contando la cantidad de caminos posibles.

desde  $(0, 0)$  mediante movimientos permitidos.

- a) Definir una función recursiva para calcular la cantidad  $h(m, n)$  de caminos para llegar desde  $(0, 0)$  a  $(m, n)$ , donde  $m$  y  $n$  son enteros positivos.

*Sugerencia:*  $h(m, n) = h(m, n - 1) + h(m - 1, n)$  si  $m$  y  $n$  son positivos (considerar también los casos  $m = 0$  o  $n = 0$ ).

☞ En cursos de matemática discreta se ve que  $h(m, n)$  es el número combinatorio  $\binom{m+n}{n}$ , lo que podemos apreciar comparando el rectángulo de la [figura 17.2](#) con el triángulo de la [figura 17.1](#) (rotándolos adecuadamente).

- b) Modificar la función en [a\)](#) de modo de calcular la cantidad de caminos cuando la intersección  $(r, s)$  está bloqueada y no se puede pasar por allí, donde  $0 < r < m$  y  $0 < s < n$ .

*Sugerencia:* poner  $h(r, s) = 0$ .

- c) Supongamos ahora que, al revés del apartado anterior, para ir de  $(0, 0)$  a  $(m, n)$  tenemos que pasar por  $(r, s)$  (por ejemplo, para llevar a  $(m, n)$  la pizza que compramos en la esquina  $(r, s)$ ). Definir una función para esta nueva posibilidad.

*Sugerencia:* puedo armar un camino de  $(0, 0)$  a  $(m, n)$  tomando cualquier camino de  $(0, 0)$  a  $(r, s)$  y después cualquier

camino de  $(r, s)$  a  $(m, n)$ .



**E 17.3.** Resolver el ejercicio anterior cuando se permite también ir en diagonales suroeste-noreste, es decir, pasar de  $(x, y)$  a  $(x + 1, y + 1)$  en un movimiento (cuando  $0 \leq x < m$  y  $0 \leq y < n$ ).



## 17.2. Las grandes listas: otro peligro de la recursión

Un problema muy distinto al de *contar* objetos, como hicimos con los caminos del [ejercicio 17.2](#), es *generarlos*, por ejemplo para encontrar alguno o todos los que satisfacen cierto criterio, y es muy común caer en la trampa de fabricar una «gran lista» de objetos innecesariamente.

Por ejemplo, supongamos que queremos obtener todos los subconjuntos de  $I_n = \{1, \dots, n\}$ .

Una de las formas de hacerlo es usar que o bien un subconjunto no contiene a  $n$ , y entonces es un subconjunto de  $I_{n-1}$ , o bien sí lo contiene, y entonces es un subconjunto de  $I_{n-1}$  al cual se le agrega  $n$ .

Esta idea, que puede usarse para demostrar que  $I_n$  tiene  $2^n$  subconjuntos, nos lleva a considerar:


```
def subconjuntosN0(n):
    """Lista de subconjuntos de {1,..., n}."""
    if n == 0:          # no hay elementos
        return [[]]    # sólo el conjunto vacío
    subs = subconjuntosN0(n-1)
    return subs + [s + [n] for s in subs]
```

(17.2)

✎ Aunque para conjuntos el orden no importa y no hay elementos repetidos, los representamos mediante listas.

Veamos el porqué de la señal ⚡ en esta construcción.

**E 17.4 (subconjuntos I).** Probar la función en (17.2) para  $n = 1, 2, 3$ , viendo que se obtienen los resultados esperados. Luego comprobar que para  $n = 0, 4, 8$  la cantidad de subconjuntos es  $2^n$ .

*Atención:* antes que dejar que en la terminal aparezcan  $2^8 = 256$  listas, es mejor hacer una asignación del tipo `s = subconjuntosN0(8)` y luego averiguar `len(s)`, o bien poner `len(subconjuntosN0(8))` directamente. 

El algoritmo para definir `subconjuntosN0` en (17.2) es interesante en cuanto se parece a una demostración de matemáticas, pero sólo debe usarse en condiciones extremas. Por ejemplo, si  $n = 20$  tendremos 1 048 576 subconjuntos y difícilmente necesitemos tener a todos a la vez. En general queremos encontrar algunos de ellos con ciertas propiedades, o contarlos, etc., con lo que las más de las veces bastará mirarlos en secuencia, uno a la vez.

**E 17.5.** Para entender el porqué de la insistencia de no tener a todos los objetos en una «gran lista», supongamos que  $a_n$  es la lista obtenida mediante `subconjuntosN0(n)`.

$a_n$  tiene  $2^n$  listas, y eliminando los corchetes, en total tiene  $n 2^{n-1}$  números (¿por qué?).<sup>(1)</sup>

- Suponiendo que cada número en  $a_n$  ocupa 8 bytes (64 bits), calcular la cantidad de bytes que ocupan los números en  $a_n$  para  $n = 10, 20$  y  $30$ , ¡sin construir  $a_n$ !
- Si la memoria de la computadora que usamos tiene 4 gigabytes (1 GB =  $2^{30}$  bytes,  $2^{30} \approx 10^9$ ), ¿cuál es el valor máximo de  $n$  que nos permite tener todos los números de  $a_n$  en la memoria (suponiendo que allí sólo están los números de  $a_n$ )?

*Respuesta:*  $n = 25$ .

☞ Cuando no hay más lugar en la memoria, los sistemas operativos en general usan espacio en disco moviendo los datos constantemente entre la memoria central y el disco, lo que hace que las operaciones sean mucho más lentas.

- Es posible que Python no use exactamente 8 bytes por número, o que la máquina tenga más o menos memoria, por lo que el va-

<sup>(1)</sup> *Sugerencia:* un conjunto y su complemento tienen  $n$  elementos entre los dos.



lor de  $n$  obtenido en el apartado anterior es sólo una estimación del lugar que ocupan los números.

Probar `subconjuntosN0(n)`, por ejemplo encontrando la cantidad de subconjuntos, comenzando desde  $n = 18$  y luego 19, 20, etc., hasta que los tiempos sean muy largos,<sup>(2)</sup> comprobando que en cada caso la cantidad de subconjuntos obtenidos es  $2^n$ .

Se podrá comprobar que difícilmente se pueda llegar a  $n = 24$ , ya sea por el tiempo o por la memoria. ¶

Como conclusión, cuando el número de objetos a generar es muy grande, como  $2^n$  o  $n!$ , independientemente de la eficiencia del algoritmo que usemos para generarlos:

*deben evitarse en lo posible las «grandes listas»  
como la de la función `subconjuntosN0` en (17.2).*

### 17.3. yield

Hay distintas técnicas para construir los objetos uno a la vez, evitando generar la «gran lista». Por ejemplo, podemos poner una función dentro de otra envolvente, como hicimos en la función `hanoi` del [ejercicio 14.7](#). En esta sección estudiaremos otra técnica basada en la sentencia `yield` de Python.

Una forma de calcular los primeros  $n$  números de Fibonacci (y no sólo uno), es construyendo una lista con un lazo `for`:

```
lista = []
a, b = 0, 1
for k in range(n):
    a, b = a + b, a
    lista.append(a)
```

<sup>(2)</sup> Tener en cuenta que al pasar de  $n$  a  $n + 1$ , básicamente se duplican el tiempo que se tarda y la memoria que se necesita.

Si no necesitamos la lista completa, y nos basta con mirar a cada número de Fibonacci individualmente, podemos usar la sentencia `yield`, que en este contexto podría traducirse como *producir*, *proveer* o *dar lugar a*:

```
def fibgen(n):  
    """Generar n números de Fibonacci."""  
    a, b = 0, 1  
    for k in range(n):  
        a, b = a + b, a  
        yield a
```

(17.3)

Observar que en este nuevo esquema «`yield a`» reemplaza a la instrucción «`lista.append(a)`» en el esquema anterior, evitando la construcción de la lista. Por otra parte, no es necesario hacer un lazo `for` para cada número de Fibonacci, a diferencia de lo que hicimos en el [ejercicio 13.1](#).

Estudiemos el comportamiento y propiedades de `fibgen`.

#### E 17.6.

- Definir la función `fibgen` según el [esquema \(17.3\)](#) en un módulo, y ejecutarlo.
- `fibgen` en principio es una función como cualquier otra. Pero a pesar de que no tiene un `return` explícito, su resultado no es `None` sino un objeto de tipo o clase `generator` (*generador*).

Poner en la terminal:

```
fibgen  
type(fibgen)  
fibgen(10)  
type(fibgen(10))
```

- Los generadores como `fibgen(n)` son un caso especial de «iteradores», y su comportamiento tiene similitudes con el de las secuencias cuando se usan con `for`. Evaluar:

```
[x for x in range(10)]
```

```
[x for x in fibgen(10)]
for x in fibgen(5):
    print(x)
```

d) Análogamente, pueden pasarse a lista o tupla:

```
list(fibgen(5))
tuple(fibgen(5))
```

e) A diferencia de las secuencias, podemos ir generando un elemento a la vez mediante la función `next` (*siguiente*).

Evaluar:

```
it = fibgen(3) # iterador
next(it)      # primer elemento
next(it)      # siguiente
next(it)      # siguiente
next(it)      # no hay más
```

↗ O algo así... Las secuencias pueden ponerse como iteradores con `iter` (que no veremos en el curso):

```
it = iter(['mi', 'mama', 'me', 'mima'])
next(it)
...
```



Analicemos lo que acabamos de ver.

Teniendo presente la definición en (17.3), cuando ponemos `it = fibgen(n)` la primera vez que llamamos a `next(it)` se realizan las instrucciones `a, b = 0, 1`, se entra al lazo `while`, se ejecuta `a, b = a + b, a`, y se retorna el valor de `a` (en este caso 1), como si se tratara de `return` en vez de `yield`.

A diferencia de una función con `return`, al ejecutarse `yield` los valores locales se mantienen, de modo que al llamar por segunda vez a `next(it)` se continúa desde el último `yield` ejecutado, como si no se hubiera retornado, continuando la ejecución de instrucciones hasta encontrar el próximo `yield`, y así sucesivamente.

En la definición de `fibgen(n)` el lazo `for` se realiza a lo sumo  $n$  veces, se pueden construir hasta  $n$  elementos con el iterador, y llamadas

posteriores a `next(it)` dan error. Sin embargo, el generador/iterador es lo suficientemente inteligente como para terminar cuando se lo usa en un lazo `for`, como en el [ejercicio 17.6.c](#)).

**E 17.7.** Si en la definición de la función generadora hay un lazo infinito, podemos hacer cuantas llamadas queramos.

a) Considerar la función

```
def naturales():  
    """Genera todos los naturales."""  
    k = 0  
    while True:  
        k = k + 1  
        yield k
```

y evaluar:

```
it = naturales()    # iterador  
[next(it) for i in range(5)]  
next(it)  
[next(it) for i in range(5)]  
it2 = naturales()  # otro iterador  
next(it2)          # comienza con 1  
next(it)           # sigue por su lado
```

b) Considerar ahora la función

```
def periodico(n):  
    """Repetir con período n."""  
    a = 0  
    while True:  
        if a == n: # volver al principio  
            a = 1  
        else:  
            a = a + 1  
        yield a
```

y evaluar:

```
it = periodico(7)
[next(it) for k in range(20)]
```

- c) Cambiar la definición de **fibgen** en (17.3) de modo de obtener *todos* los números de Fibonacci (y no sólo  $n$ ) con el iterador. ¶

## 17.4. Generando objetos combinatorios

Veamos cómo **yield** nos permite evitar la construcción de la «gran lista» del [ejercicio 17.4](#).

**E 17.8 (subconjuntos II).** La función **subs** en el módulo [recursion2](#) genera todos los subconjuntos de  $\{1, \dots, n\}$  para un  $n$  dado.

- a) Ejecutar el módulo y evaluar:

```
for x in subs(3):
    print(x)
```

Comparar el resultado cambiando **for s in subs(3)** por **for s in subconjuntosN0(3)**.

- b) Para contar los elementos podemos poner:

```
c = 0
for x in subs(5):
    c = c + 1
c
```

- c) La variable  $s$  dentro de la función **subs** es una lista y por lo tanto mutable, lo que puede llevar a confusión. Evaluar

```
it = subs(3)          # hay 8 subconjuntos
x = next(it)
x          # x es el primero
for k in range(4):    # hacemos 4 más
    next(it)
x          # ahora x es el 5to. y no el 1ro.
```



⇒ La lista *s* en **subs** es compartida por todas las llamadas sucesivas.

Además, *s* se comporta como una pila: empezando de la lista vacía, en *s* vamos agregando y sacando elementos de atrás. Al terminar de generar los elementos, *s* termina siendo la lista vacía.



d) Comparar con los apartados *c*) y *d*) del [ejercicio 17.6](#):

```
[x for x in subs(3)]
list(subs(3))
[list(x) for x in subs(3)]
```

⇒ Si por alguna razón necesitamos conservar resultados intermedios, tenemos que hacer copias de las listas (ver el [ejercicio 9.14](#) y siguientes).

e) Otra forma de ver el comportamiento de *s* es eliminar el renglón **s.pop()** en la definición de **subs**. En ese caso, ¿qué esperarías?



En el [ejercicio 17.8](#) vimos que el iterador asociado a **subs(n)** usa una única lista que se va modificando. Desde ya que esta lista no ocupa más de *n* lugares para guardar los números, a diferencia de la «gran lista» de **subconjuntosN0(n)** que guarda  $n \times 2^{n-1}$  números simultáneamente en la memoria. Los tiempos necesarios para recorrer todos los subconjuntos de  $I_n$  con una y otra función son bastante comparables, ya que en definitiva se basan en el mismo algoritmo, aunque la versión de la «gran lista» en general tiene que lidiar con más estructuras de datos y tarda más. La gran diferencia (¡exponencial!) radica en la cantidad de memoria que necesitan ambos.

**E 17.9.** Repetir el [ejercicio 17.5.c](#)) ahora con **subs(n)** para  $n = 18, 19, 20, \dots$ , comparando los resultados en uno y otro caso.

🔍 La versión con «gran lista» para  $n = 23$  cabe en la memoria principal de mi máquina, pero tarda entre 4 y 5 veces lo que tarda la versión con **yield**.



En el módulo `recursion2` se define la función `subs`, así como las funciones `subsnk`, que genera los subconjuntos de  $I_n$  con exactamente  $k$  elementos, y `perms`, que genera todas las permutaciones de  $I_n$ .

El algoritmo de `subsnk` se basa en que un subconjunto de  $k$  elementos de  $I_n$  o bien no tiene a  $n$ , y entonces es un subconjunto de  $I_{n-1}$  con  $k$  elementos, o bien sí lo tiene, y entonces al sacarlo obtenemos un subconjunto de  $I_{n-1}$  con  $k - 1$  elementos.

- ✎ Es una forma de pensar la identidad  $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$  para  $0 < k < n$ . Ver (17.1) y los comentarios asociados.

Para construir las permutaciones con `perms`, observamos que una permutación de  $I_n$  se obtiene a partir de una permutación de  $I_{n-1}$  insertando  $n$  en alguno de los  $n$  lugares disponibles (al principio, entremedio o al final).

- ✎ Esto da una forma de demostrar que  $n! = (n - 1)! \times n$ : cada una de las permutaciones de  $I_{n-1}$  da lugar a  $n$  permutaciones de  $I_n$ .
- ✎ Una versión *bastante menos eficiente* es reemplazar el intercambio por inserciones y borrados explícitos en cada paso, debido a la ineficiencia de estas operaciones en Python (como mencionamos otras veces, por ejemplo, en la [página 208](#)):

```
...
for p in perms(n-1):
    for i in range(n):
        p.insert(i, n)
        yield p
        p.pop(i)
...
```

**E 17.10.** Repetir los apartados del [ejercicio 17.8](#), cambiando `subs` por `subsnk(6, 3)` y `perms(4)` en los lugares adecuados.


- ☞ En `subsnk` no hay una única lista  $s$ , sino que se generan  $k + 1$  listas:  $[], [1], [1, 2], \dots, [1, 2, \dots, k]$ .

En cambio, `perms` usa siempre la misma lista  $s$ , pero no se comporta como una pila, ya que  $n$  se agrega al final y va cambiando de posición hasta salir por adelante.

**E 17.11.** a) Definir una función **cadenas**( $m$ ,  $n$ ) para generar todas las listas de longitud  $n$  con los elementos de  $\{0, \dots, m-1\}$  ( $m, n \in \mathbb{N}$ ). Por ejemplo, si  $m = 3$  y  $n = 2$ , las posibilidades son:

$[0, 0]$ ,  $[0, 1]$ ,  $[0, 2]$ ,  $[1, 0]$ ,  $[1, 1]$ ,  $[1, 2]$ ,  $[2, 0]$ ,  $[2, 1]$ ,  $[2, 2]$ .

Imprimir todas las cadenas para  $m = 2$  y  $n = 4$ , y contarlas para  $m = 5$  y  $n = 6$  (en general hay  $m^n$ ).

b) Podemos pensar que las listas del apartado anterior son los coeficientes de un número escrito en base  $m$ . Definir una función para encontrar las listas del apartado anterior recorriendo los números entre 0 y  $m^n - 1$  (inclusivos). 

**E 17.12 (camino de Hamilton).** Un *camino de Hamilton* en un grafo es un camino simple (sin vértices repetidos) que pasa por todos los vértices del grafo.

No todos los grafos tienen un camino así. El grafo debe ser al menos conexo, pero la conexión tampoco es suficiente ya que los árboles en general (salvo que se reduzcan a un camino) no tienen estos caminos.

Siguiendo un esquema como:


**para cada** permutación  $p$  de  $1, \dots, n$ :


**si**  $p$  es camino del grafo:

**retornar**  $p$  # salir

informar que no existe camino de Hamilton


definir una función **hamilton**( $n$ , **aristas**) que dado un grafo por su cantidad de vértices  $n$  y la lista de aristas, determine si el grafo tiene un camino de Hamilton y en ese caso lo exhiba, considerando todas las permutaciones de  $I_n$  posibles y terminando en cuanto se encuentre.

 Recordar el [ejercicio 15.9](#).

 Por supuesto que el algoritmo propuesto es muy brutal, y se reduce esencialmente a un barrido sobre todas las soluciones posibles, sólo que terminando en cuanto se pueda.

En este sentido, vemos la importancia de no generar la «gran lista», sino generar las permutaciones de a una.



- ✎ Dada la importancia del problema, existen algoritmos *mucho* más eficientes, aunque no se conocen algoritmos *verdaderamente* eficientes (por ejemplo, polinomiales). 

## 17.5. Ejercicios adicionales

**E 17.13.** A veces queremos combinar de todas las formas posibles objetos que pueden estar repetidos. Por ejemplo, si queremos construir todas las cadenas de caracteres posibles que se pueden hacer con las letras de «mimamamemima» usando todas las letras.

- ✎ Como el orden es importante, «mimamamemima» es distinto a «memimamimama», estos objetos son un tipo de permutaciones.
- a) En cursos de matemática discreta se demuestra que si hay  $k$  objetos distintos cada uno apareciendo  $n_i$  veces, la cantidad de permutaciones con repetición es el *coeficiente multinomial*

$$\binom{n_1 + n_2 + \dots + n_k}{n_1, n_2, \dots, n_k} = \frac{(n_1 + n_2 + \dots + n_k)!}{n_1! n_2! \dots n_k!}.$$

Por ejemplo, en 'mimamamemima' la letra 'a' aparece 3 veces, 'e' aparece 1 vez, 'i' aparece 2 veces, y 'm' aparece 6 veces, por lo que la cantidad de permutaciones con repetición es

$$\frac{(3 + 1 + 2 + 6)!}{3! \times 1! \times 2! \times 6!} = 55\,440.$$

Definir una función `multinomial(lista)` para calcular estos coeficientes, donde `lista` es de la forma `[n1, n2, ...]`.

- b) Definir una función `permsrep(lista)` para generar todas las permutaciones posibles de una lista dada que puede tener repeticiones. Por ejemplo, si la lista es `['m', 'i', 'm', 'a']`, hay 12 permutaciones posibles, como:

`['m', 'i', 'm', 'a'], ['m', 'i', 'a', 'm'],`  
`['m', 'a', 'i', 'm'], ['a', 'm', 'i', 'm'],`


['m', 'm', 'i', 'a'], ['m', 'm', 'a', 'i'], etc.

*Aclaración 1:* la lista que es argumento no debe modificarse.

*Aclaración 2:* no construir una «gran lista».

*Sugerencia:* imitar la función `perms`, finalizando el intercambio en cuanto se encuentra un elemento repetido.

- c) Aplicar la función `permsrep` (o variante) para encontrar todas las palabras distintas que se pueden escribir permutando las letras de `'mimama'` (sin espacios).

Algunas de las palabras a obtener son: `'mimama'`, `'mimaam'`, `'mimmaa'`, `'miamma'`, `'miamam'`, etc. (hay 60 posibles). 

## 17.6. Comentarios

- En este capítulo vimos que si bien recursión en general es ineficiente, se puede mejorar su desempeño con distintas técnicas.

También vimos que, de cualquier forma, siempre nos topamos con la denominada «explosión combinatoria»: la cantidad exponencial de objetos a examinar cuando usamos técnicas de barrido.

Uno de los grandes desafíos de las matemáticas es encontrar algoritmos eficientes para resolver problemas asociados, aunque —como ya mencionamos— difícilmente existan algoritmos *verdaderamente* eficientes: entre los [problemas del milenio](http://es.wikipedia.org/wiki/Problemas_del_milenio)<sup>(3)</sup> (remedando los problemas planteados por Hilbert un siglo antes) se encuentra justamente decidir si  $P \neq NP$ .

- Los generadores vía `yield` son variantes de las *corutinas*, que existen desde hace varios años en algunos lenguajes de programación, estando Simula y Modula-2 entre los primeros en usarlas.

Simula fue desarrollado en los 60 extendiendo el lenguaje Algol, del cual descende también Pascal. A su vez, Modula-2 fue desarrollado a fines de los 70 por N. Wirth como sucesor

<sup>(3)</sup> [http://es.wikipedia.org/wiki/Problemas\\_del\\_milenio](http://es.wikipedia.org/wiki/Problemas_del_milenio)

«profesional» de Pascal, que es de un carácter más «pedagógico».

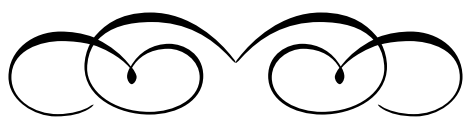
A partir de la versión 3.5 de Python se ha tratado de hacer una distinción más clara entre *generadores* y *corutinas*.

- En el módulo estándar *itertools* de Python se generan permutaciones, combinaciones, y otros objetos combinatorios. Nosotros no veremos este módulo en el curso.



## Parte III

### *Apéndices*



# Apéndice A

## Módulos y archivos mencionados

En la [página del «libro»<sup>\(1\)</sup>](#) hay copias de los archivos en este apéndice. Recordar que están codificados en utf-8.

### A.1. Módulos Python

#### ***dearchivoatterminal*** ([ejercicio 11.6](#))

```
"""Leer un archivo de datos e imprimirlo en la terminal."""  
  
entrada = input('Ingresar el nombre del archivo a leer: ')  
  
lectura = open(entrada, 'r', encoding='utf-8')      # abrirlo  
  
for renglon in lectura:  
    print(renglon, end='')  
  
lectura.close()      # y cerrarlo
```

---

<sup>(1)</sup> <http://oma.org.ar/invydoc/libro-prog.html>

**decimales (capítulo 16)**

```
"""Algunas propiedades de la representación decimal en Python.
```

- epsmaq: épsilon de máquina  
    epsmaq = epsi(1.0)
- epsmin: épsilon mínimo (menor potencia de 2 que es positiva).  
    epsmin = epsi(0.0)
- maxpot2: máxima potencia de 2 que se puede representar.
- maxnum: máximo número decimal que se puede representar.

Para curiosos:

```
>>> import sys
>>> sys.float_info
>>> sys.float_info.epsilon
```

```
>>> import math
>>> help(math.frexp)
>>> math.frexp(epsmaq)
>>> math.frexp(epsmin)
>>> math.frexp(maxpot2)
>>> math.frexp(maxnum)
"""
```

```
def epsi(x):
```

```
    """Menor potencia de 2 que sumada a x da mayor que x.
```

```
    - x debe ser float,  $0 \leq x \leq$  mayor número representable.
    """
```

```
    if x == float('inf'): # salir
        print(x, 'es demasiado grande')
        return
```

```
    x = float(x)         # para evitar lazo infinito
```

```
y = 1.0
# ver si hay que agrandar o achicar y
if x + y > x:      # achicar y
    while x + y > x:
        z, y = y, y/2
    return z
# agrandar y
while x + y == x:
    y = 2 * y
return y
```

```
epsmaq = epsi(1.0)
epsmin = epsi(0.0)
```

```
def maxpot2calc():
    """Cálculo de máxima potencia de 2 representable como float."""
    x = 1.0
    while 2 * x > x:
        x, s = 2 * x, x
    return s
```

```
maxpot2 = maxpot2calc()
```

```
def maxnumcalc():
    """Calcular el máximo número decimal representable."""
    global maxnum
    x = maxpot2
    p = maxpot2 / 2
    while x + p > x:
        s = x
        x = x + p
        p = p / 2
    return s
```

```
maxnum = maxnumcalc()
```



## **eratostenes (ejercicio 13.16)**

```
"""Versión sencilla de la criba de Eratóstenes.
```

```
Primera versión que debe mejorarse según  
los ejercicios del apunte.
```

```
"""
```

```
def criba(n):
```

```
    """Lista de primos <= n."""
```

```
    #-----
```

```
    # inicialización
```

```
    # usamos una lista por comprensión para que todos
```

```
    # los elementos tengan un mismo valor inicial
```

```
    # las posiciones 0 y 1 no se usan, pero conviene ponerlas
```

```
    esprimo = [True for i in range(n + 1)]
```

```
    #-----
```

```
    # lazo principal
```

```
    for i in range(2, n+1):
```

```
        if esprimo[i]:
```

```
            for j in range(i * i, n + 1, i):
```

```
                esprimo[j] = False
```

```
    #-----
```

```
    # salida
```

```
    # observar el uso del filtro
```

```
    return [i for i in range(2, n + 1) if esprimo[i]]
```

## **euclides2 (ejercicio 16.10)**

```
"""Complicaciones con Euclides usando decimales.
```

```
Funciones mcd1, mcd2, mcd3 y mcd4 explicadas en el apunte.
```

```
"""
```

```
# máximo número de iteraciones para lazos
maxit = 1000

def mcd1(a, b):
    """Cálculo de la medida común según Euclides.

    a y b deben ser positivos.
    """
    for it in range(maxit):
        if a > b:
            a = a - b
        elif b > a:
            b = b - a
        else:
            break
    print('    iteraciones:', it + 1)
    if it == maxit - 1:
        print('*** Máximas iteraciones alcanzadas.')
    print('    b:', b)
    return a

def mcd2(a, b):
    """Variante usando restos, a y b positivos."""
    for it in range(maxit):
        if b == 0:
            break
        a, b = b, a % b
    print('    iteraciones:', it + 1)
    if it == maxit - 1:
        print('*** Máximas iteraciones alcanzadas.')
    print('    b:', b)
    return a

# tolerancia permitida
tol = 10**(-7)

def mcd3(a, b):
    """Cálculo de la medida común según Euclides, a y b positivos.
```

```
Terminamos cuando la diferencia es chica.
"""
for it in range(maxit):
    if a > b + tol:
        a = a - b
    elif b > a + tol:
        b = b - a
    else:
        break
print('    iteraciones:', it + 1)
if it == maxit - 1:
    print('*** Máximas iteraciones alcanzadas.')
print('    b:', b)
return a

def mcd4(a, b):
    """Variante usando restos, a y b positivos."""
    for it in range(maxit):
        if b < tol:
            break
        a, b = b, a % b
    print('    iteraciones:', it + 1)
    if it == maxit - 1:
        print('*** Máximas iteraciones alcanzadas.')
    print('    b:', b)
    return a
```

### **fargumento** ([ejercicio 7.14](#))

```
"""Funciones en las que uno de los argumentos es otra función."""

def aplicar(f, x):
    """Aplica f a x."""
    return f(x)

def f(x):
    """Suma 1 al argumento."""
```

```
    return x + 1

def g(x):
    """Multiplica el argumento por 2."""
    return 2 * x
```

### **flocal** (ejercicio 7.13)

```
"""Ejemplo de función definida dentro de otra."""

def fexterna():
    """Ilustración de variables y funciones globales y locales."""

    def finterna(): # función interna, local a fexterna
        global x    # variable global
        x = 5       # se modifica acá

    x = 2           # x es local a fexterna
    print('al comenzar fexterna, x es', x)

    finterna()
    print('al terminar fexterna, x es', x)
```

### **grafos** (capítulo 15)

```
"""Funciones para grafos.

- dearistasavecinos: de lista de arista a lista de vecinos.
- devecinosaaristas: de lista de vecinos a lista de aristas.
- recorrido: recorrido lifo de un grafo a partir de una raíz.

Recordar que numeramos los vértices a partir de 1 (y no de 0).
"""

def dearistasavecinos(ngrafo, aristas):
    """Pasar de lista de aristas a lista de vecinos."""
```

```
vecinos = [[] for v in range(ngrafo + 1)]
vecinos[0] = None
for u, v in aristas:
    vecinos[u].append(v)
    vecinos[v].append(u)
return vecinos
```

```
def devecinosaaristas(vecinos):
    """Pasar de lista de vecinos a lista de aristas."""
    ngrafo = len(vecinos) - 1 # índices para vértices desde 1
    aristas = []
    for v in range(1, ngrafo): # no usamos vecinos[0]
        for u in vecinos[v]: # y u <= ngrafo
            # guardamos arista sólo si v < u para no duplicar
            if v < u:
                aristas.append([v, u])
    return aristas
```

```
def recorrido(vecinos, raiz):
    """Recorrer el grafo a partir de una raíz.
```

- Retorna los vértices «visitados».
  - Los datos son la lista de vecinos y la raíz.
  - Los índices para los vértices empiezan desde 1.
  - Se usa una cola lifo.
- ```
"""
```

```
vertices = range(len(vecinos)) # incluimos 0
padre = [None for v in vertices]
cola = [raiz]
padre[raiz] = raiz
while len(cola) > 0: # mientras la cola es no vacía
    u = cola.pop() # sacar uno (el último) y visitarlo
    for v in vecinos[u]: # examinar vecinos de u
        if padre[v] == None: # si no estuvo en la cola
```

```
        cola.append(v)      # agregarlo a la cola
        padre[v] = u        # poniendo de dónde viene
    return [v for v in vertices if padre[v] != None]
```

## **holamundo** (ejercicio 6.1)

```
"""Imprime 'Hola Mundo'.
```

Es un módulo sencillo para ilustrar cómo se trabaja con módulos propios (y también cómo se documentan).

```
"""
```

```
print('Hola Mundo')
```

## **holapepe** (ejercicio 6.2)

```
"""Ilustración de ingreso interactivo en Python.
```

Pregunta un nombre e imprime 'Hola' seguido del nombre.

```
"""
```

```
print('¿Cómo te llamas?')
```

```
pepe = input()
```

```
print('Hola', pepe, 'encantada de conocerte')
```

## **ifwhile** (capítulo 8)

```
"""Ejemplos sencillos de if y while."""
```

```
def espositivo(x):
```

```
    """Decidir si el número es positivo o no.
```

```
    El argumento debe ser un número.
```

```
    """
```

```
    if x > 0:    # si x es positivo
```

```
        print(x, 'es positivo')
```

```
    else:       # en otro caso
```

```
        print(x, 'no es positivo')
```

```
def piso(x):
```

```
    """Encontrar el piso de un número."""
```

```

y = int(x)      # int redondea hacia cero
if y < x:       # x debe ser positivo
    print('el piso de', x, 'es', y)
elif x < y:     # x debe ser negativo
    print('el piso de', x, 'es', y - 1)
else:          # x es entero
    print('el piso de', x, 'es', y)

```

```

def resto(a, b):
    """Resto de la división entre los enteros positivos a y b.

```

```

    Usamos restas sucesivas.
    """

```

```

    r = a        # al principio hay a
    while r >= b: # mientras pueda sacar b
        r = r - b # lo saco
    print('El resto de dividir', a, 'por', b, 'es', r)

```

```

def cifras(n):
    """Cantidad de cifras del entero n (en base 10)."""

```

```

    # inicialización
    n = abs(n) # por si n es negativo
    c = 0      # contador de cifras

    # lazo principal
    while True: # repetir...
        c = c + 1
        n = n // 10
        if n == 0: # ... hasta que n es 0
            break

```

```

    # salida
    return c

```

```

def mcddr(a, b):
    """Máximo común divisor entre los enteros a y b.

```

Versión usando divisiones enteras y restos.

```
mcddr(0, 0) = 0.  
"""
```

```
# nos ponemos en el caso donde ambos son no negativos  
a = abs(a)  
b = abs(b)
```

```
# lazo principal  
while b != 0:  
    r = a % b  
    a = b  
    b = r  
# acá b == 0
```

```
# salida  
return a
```

## **numerico** (capítulo 16)

```
"""Algunos métodos iterativos de cálculo numérico.
```

```
- puntofijo
```

```
- newton
```

```
- biseccion  
"""
```

```
def puntofijo(func, xinic):
```

```
    """Encontrar punto fijo de func dando punto inicial xinic.
```

```
    'func' debe estar definida como función.
```

```
    Se termina por un número máximo de iteraciones o  
    alcanzando una tolerancia permitida.
```

```
    """
```



```
# algunas constantes
maxit = 200      # máximo número de iteraciones
tol = 10**(-7)   # error permitido

# inicialización y lazo
yinic = func(xinic)
x, y = float(xinic), float(yinic)
for it in range(maxit):
    if abs(y - x) < tol:
        break
    x, y = y, func(y)

if it + 1 == maxit:
    print(' *** Iteraciones máximas alcanzadas')
    print('      el resultado puede no ser punto fijo')

return y
```

```
def newton(func, x0):
    """Método de Newton para encontrar ceros de una función.
```

```
    'func' debe estar definida como función.
```

```
    x0 es un punto inicial.
```

```
    La derivada de 'func' se estima poniendo
    un incremento bien pequeño.
```

```
    No se controla si la derivada es 0.
    """
```

```
# algunas constantes
dx = 1.0e-7      # incremento para la derivada
tol = 1.0e-7     # error permitido
maxit = 20       # máximo número de iteraciones
```

```
# estimación de la derivada de func
```

```
def fp(x):
    return (func(x + dx/2) - func(x - dx/2))/dx

# inicialización
y0 = func(x0)
x, y = x0, y0

# lazo principal
for it in range(maxit):
    if abs(y) < tol:
        break
    x1 = x - y/fp(x)
    y1 = func(x1)
    x, y = x1, y1

if it + 1 == maxit:
    print(' *** Iteraciones máximas alcanzadas')
    print('     el resultado puede no ser raíz')

return x
```

```
def biseccion(func, poco, mucho):
    """Encontrar ceros de una función usando bisección.

    'func' debe estar definida como función y
    tener signos distintos en 'poco' y 'mucho'.
    """

    # algunas constantes
    eps = 1.0e-7    # error permitido

    # inicialización
    fpoco = func(poco)
    if abs(fpoco) < eps:    # poco es raíz
        return poco

    fmucho = func(mucho)
    if abs(fmucho) < eps:    # mucho es raíz
```

```

        return mucho

# compatibilidad para el método
if fpoco * fmucho > 0:
    print('*** La función debe tener', end=' ')
    print('signos opuestos en los extremos')
    return None

# arreglamos signos de modo que fpoco < 0
if fpoco > 0:
    poco, mucho = mucho, poco

# a partir de acá hay solución si la función es continua
# lazo principal
while True:
    medio = (poco + mucho) / 2
    fmedio = func(medio)
    if abs(fmedio) < eps:          # tolerancia en y alcanzada
        break
    if abs(mucho - poco) < eps: # tolerancia en x alcanzada
        break
    if fmedio < 0:
        poco = medio
    else:
        mucho = medio

# salida
return medio

```

## **recursion1** (capítulo 14)

"""Ejemplos de funciones recursivas.

- factorial: n!
- fibonacci: el n-ésimo número de Fibonacci
- mcd: versión recursiva del algoritmo de Euclides

```
- hanoi: solución al problema de las torres de Hanoi
"""
```

```
def factorial(n):
    """n! con recursión (n entero no negativo)."""
    if n == 1:
        return 1
    return n * factorial(n-1)
```

```
def fibonacci(n):
    """n-ésimo úmero de Fibonacci con recursión."""
    if n > 2:
        return fibonacci(n-1) + fibonacci(n-2)
    return 1
```

```
def mcd(a, b):
    """Calcular el máximo común divisor de a y b.
```

Versión recursiva de la original de Euclides.

a y b deben ser enteros positivos.

```
"""
```

```
if (a > b):
    return mcd(a - b, b)
```

```
if (a < b):
    return mcd(a, b - a)
```

```
return a
```

```
def hanoi(n):
    """Solución recursiva a las torres de Hanoi.
```

- n es la cantidad de discos.

- Imprime las directivas a seguir en cada paso.

```
"""
```

```
# función interna
```

```
def pasar(k, x, y, z):
    """Pasar los discos 1,..., k de x a y usando z."""
    if k == 1:
        print('pasar el disco 1 de', x, 'a', y)
    else:
        pasar(k - 1, x, z, y)
        print('pasar el disco', k, 'de', x, 'a', y)
        pasar(k - 1, z, y, x)

# ejecución
pasar(n, 'a', 'b', 'c')
```

## **recursion2** (capítulo 17)

```
"""Funciones recursivas con yield.

- subs: subconjuntos de {1,...,n}

- subsnk: subconjuntos de {1,...,n} con k elementos.

- perms: permutaciones de {1,...,n}.
"""

def subs(n):
    """Generar los subconjuntos de {1,...,n}."""
    if n == 0:
        # no hay elementos
        yield []
        # único subconjunto
    else:
        for s in subs(n-1):
            yield s
            # subconjunto de {1,...,n-1}
            s.append(n) # agregar n al final
            yield s
            # con n
            s.pop()
            # sacar n

def subsnk(n, k):
    """Subconjuntos de {1,...,n} con k elementos."""
    if k == 0:
        # no hay elementos
        yield []
```

```
elif n == k:      # están todos
    yield list(range(1, n + 1))
else:
    for s in subsnk(n-1, k):
        yield s    # subconjunto de {1,...,n-1}
    for s in subsnk(n-1, k-1):
        s.append(n) # agregar n al final
        yield s     # con n
        s.pop()     # sacar n

def perms(n):
    """Permutaciones de {1,...,n}."""
    if n == 0:      # nada que permutar
        yield []
    else:
        for p in perms(n-1):
            p.append(n)    # agregar n al final
            yield p
            i = n - 1
            while i > 0:    # llevar n hacia adelante
                j = i - 1
                p[i] = p[j]
                p[j] = n
                yield p
                i = j
            p.pop(0)       # sacar n
```

### **sumardos** (ejercicio 6.3)

```
"""Sumar dos objetos ingresados interactivamente."""

print(__doc__)

a = input('Ingresar algo: ')
b = input('Ingresar otra cosa: ')

print('La suma de', a, 'y', b, 'es', a + b)
```

**tablaseno (ejercicio 11.7)**

```
"""Hacer una tabla del seno para ángulos entre 0 y 90 grados."""

import math

aradianes = math.pi/180

# abrir el archivo
archivo = open('tablaseno.txt', 'w', encoding='utf-8')

# escribir la tabla
for grados in range(0, 91):
    archivo.write('{0:3}   {1:<15.8g}\n'.format(
        grados, math.sin(grados*aradianes)))

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

**A.2. Archivos de texto****santosvega.txt (ejercicios 11.5 y 11.6)**

Cuando la tarde se inclina  
sollozando al occidente,  
corre una sombra doliente  
sobre la pampa argentina.  
Y cuando el sol ilumina  
con luz brillante y serena  
del ancho campo la escena,  
la melancólica sombra  
huye besando su alfombra  
con el afán de la pena.

**unilang.txt (ejercicio 11.6)**

Es un archivo de texto codificado en utf-8 tomado de

<http://www.humancomp.org/unichtm/unilang8.htm>

Es difícil de copiar a y desde el pdf: la recomendación es elegir el texto y copiarlo en una ventana de IDLE que no sea la de la terminal. También se puede encontrar directamente en

<http://oma.org.ar/invydoc/docs-libro/unilang8.txt>

外国語の学習と教授

Language Learning and Teaching

Изучение и обучение иностранных языков

Tere Daaheng Aneng Karimah

語文教學・语文教学

Enseñanza y estudio de idiomas

Изучаване и Преподаване на Чужди Езици

ᲥᲗᲗᲗᲗᲗᲗ ᲥᲗᲗᲗ ᲥᲥᲗᲗᲗᲗᲗᲗ ᲗᲗ ᲗᲗᲗᲗᲗᲗᲗᲗ

'læŋɡwɪdʒ 'lɜːr:nɪŋ ænd 'ti:tʃɪŋ

Lus kawm thaib qhia

Ngôn Ngữ, Sự học,

ללמוד וללמד את השפה

L'enseignement et l'étude des langues

말배우기와 가르치기

Nauka języków obcych

Γλωσσική Εκμάθηση και Διδασκαλία

تدریس و یادگیری زبان

Sprachlernen und -lehren

تعلم وتدریس العربية

เรียนและสอนภาษา





# Apéndice B

## Notaciones y símbolos

Ponemos aquí algunas notaciones, abreviaciones y expresiones usadas (que pueden diferir de algunas ya conocidas), sólo como referencia: deberías mirarlo rápidamente y volver cuando surja alguna duda.

### B.1. Lógica

- $\Rightarrow$  *implica o entonces.*  $x > 0 \Rightarrow x = \sqrt{x^2}$  puede leerse como *si  $x$  es positivo, entonces...*
- $\Leftrightarrow$  *si y sólo si.* Significa que las condiciones a ambos lados son equivalentes. Por ejemplo  $x \geq 0 \Leftrightarrow |x| = x$  se lee  *$x$  es positivo si y sólo si...*
- $\exists$  *existe.*  $\exists k \in \mathbb{Z}$  tal que... se lee *existe  $k$  entero tal que...*
- $\forall$  *para todo.*  $\forall x > 0, x = \sqrt{x^2}$  se lee *para todo  $x$  positivo,...*
- $\neg$  La negación lógica *no*. Si  $p$  es una proposición lógica,  $\neg p$  se lee *no  $p$* .  $\neg p$  es verdadera  $\Leftrightarrow p$  es falsa.
- $\wedge$  La conjunción lógica *y*. Si  $p$  y  $q$  son proposiciones lógicas,  $p \wedge q$  es verdadera  $\Leftrightarrow$  tanto  $p$  como  $q$  son verdaderas.

- $\vee$  La disyunción lógica o. Si  $p$  y  $q$  son proposiciones lógicas,  $p \vee q$  es verdadera  $\Leftrightarrow$  o bien  $p$  es verdadera o bien  $q$  es verdadera.

## B.2. Conjuntos

- $\in$  pertenece.  $x \in A$  significa que  $x$  es un elemento de  $A$ .
- $\notin$  no pertenece.  $x \notin A$  significa que  $x$  no es un elemento de  $A$ .
- $\cup$  unión de conjuntos.  $A \cup B = \{x : x \in A \text{ o } x \in B\}$ .
- $\cap$  intersección de conjuntos.  $A \cap B = \{x : x \in A \text{ y } x \in B\}$ .
- $\setminus$  diferencia de conjuntos.  $A \setminus B = \{x : x \in A \text{ y } x \notin B\}$ .
- $|A|$  cardinal del conjunto  $A$ . Es la cantidad de elementos de  $A$ . No confundir con  $|x|$ , el valor absoluto del número  $x$ .
- $\emptyset$  El conjunto vacío,  $|\emptyset| = 0$ .

## B.3. Números: conjuntos, relaciones, funciones

- $\mathbb{N}$  El conjunto de números naturales,  $\mathbb{N} = \{1, 2, 3, \dots\}$ . Para nosotros  $0 \notin \mathbb{N}$ .
- $\mathbb{Z}$  Los enteros,  $\mathbb{Z} = \{0, 1, -1, 2, -2, \dots\}$ .
- $\mathbb{Q}$  Los racionales  $p/q$ , donde  $p, q \in \mathbb{Z}$ ,  $q \neq 0$ .
- $\mathbb{R}$  Los reales. Son todos los racionales más números como  $\sqrt{2}$ ,  $\pi$ , etc., que no tienen una expresión decimal periódica.
- $\pm$  Si  $x$  es un número,  $\pm x$  representa dos números:  $x$  y  $-x$ .
- $\approx$  aproximadamente.  $x \approx y$  se lee  $x$  es aproximadamente igual a  $y$ .
- $\ll$  mucho menor.  $x \ll y$  se lee  $x$  es mucho menor que  $y$ .
- $\gg$  mucho mayor.  $x \gg y$  se lee  $x$  es mucho mayor que  $y$ .

- $m \mid n$   $m$  divide a  $n$  o también  $n$  es múltiplo de  $m$ . Significa que existe  $k \in \mathbb{Z}$  tal que  $n = km$ .  
 $m$  y  $n$  deben ser enteros.
- $|x|$  El *valor absoluto* o *módulo* del número  $x$ . No confundir con  $|A|$ , el cardinal del conjunto  $A$ .
- $\lfloor x \rfloor$  El *piso* de  $x$ ,  $x \in \mathbb{R}$ . Es el mayor entero que no supera a  $x$ , por lo que  $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$ . Por ejemplo,  $\lfloor \pi \rfloor = 3$ ,  $\lfloor -\pi \rfloor = -4$ ,  $\lfloor z \rfloor = z$  para todo  $z \in \mathbb{Z}$ .
- $\lceil x \rceil$  La *parte entera* de  $x$ ,  $x \in \mathbb{R}$ ,  $\lceil x \rceil = \lfloor x \rfloor$ . Nosotros usaremos la notación  $\lfloor x \rfloor$ , siguiendo la costumbre en las áreas relacionadas con la computación.
- $\lceil x \rceil$  El *techo* de  $x$ ,  $x \in \mathbb{R}$ . Es el primer entero que no es menor que  $x$ , por lo que  $\lceil x \rceil - 1 < x \leq \lceil x \rceil$ . Por ejemplo,  $\lceil \pi \rceil = 4$ ,  $\lceil -\pi \rceil = -3$ ,  $\lceil z \rceil = z$  para todo  $z \in \mathbb{Z}$ .
- $e^x$ ,  
 $\exp(x)$  La función *exponencial* de base  $e = 2.718281828459 \dots$
- $\log_b x$  El *logaritmo* de  $x \in \mathbb{R}$  en base  $b$ .  
 $y = \log_b x \Leftrightarrow b^y = x$ .  
 $x$  y  $b$  deben ser positivos,  $b \neq 1$ .
- $\ln x$ ,  
 $\log x$  El *logaritmo natural* de  $x \in \mathbb{R}$ ,  $x > 0$ , o *logaritmo en base  $e$* ,  $\ln x = \log_e x$ . Es la inversa de la exponencial,  $y = \ln x \Leftrightarrow e^y = x$ .  
Para no confundir, seguimos la convención de Python: si no se especifica la base, se sobreentiende que es  $e$ , es decir,  $\log x = \ln x = \log_e x$ .
- $\sen x$ ,  
 $\sin x$  La función trigonométrica *seno*, definida para  $x \in \mathbb{R}$ .
- $\cos x$  La función trigonométrica *coseno*, definida para  $x \in \mathbb{R}$ .

$\tan x$  La función trigonométrica *tangente*,  $\tan x = \sin x / \cos x$ .

$\arcsen x$ ,

$\arccos x$ ,

$\arctan x$  Funciones trigonométricas inversas respectivamente de  $\sin$ ,  $\cos$  y  $\tan$ .

En Python, se indican como **asin**, **acos** y **atan** (resp.).

$\text{signo}(x)$  La función *signo*, definida para  $x \in \mathbb{R}$  por

$$\text{signo}(x) = \begin{cases} 1 & \text{si } x > 0, \\ 0 & \text{si } x = 0, \\ -1 & \text{si } x < 0. \end{cases}$$

⚠ Algunos autores consideran que  $\text{signo}(0)$  no está definido.

$\Sigma$  Indica suma,  $\sum_{i=1}^n a_i = a_1 + a_2 + \cdots + a_n$ .

$\Pi$  Indica producto,  $\prod_{i=1}^n a_i = a_1 \times a_2 \times \cdots \times a_n$ .

## B.4. Números importantes en programación

$\varepsilon_{\text{mín}}$  El menor número positivo para la computadora.

$\varepsilon_{\text{máq}}$  El menor número positivo que sumado a 1 da mayor que 1 en la computadora.

## B.5. En los apuntes

¶ Señala el fin de un ejercicio, resultado o ejemplo, para distinguirlo del resto del texto.

⚠ Nota, en general más técnica. En letras más pequeñas. Muchas veces se puede omitir en una primera lectura.



Moraleja, explicación, conclusión o recomendación especialmente interesante. El texto está en cursiva para distinguirlo.



Pasaje con conceptos confusos o con información crucial: debe leerse cuidadosamente.



Cosas que hay que evitar. *En general quitan puntos en los exámenes.*

## B.6. Generales

□ Indica un espacio en blanco en entrada o salida de datos.

i. e. *es decir* o *esto es*, del latín *id est*.

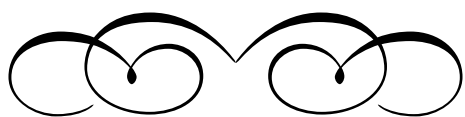
e. g. *por ejemplo*, del latín *exempli gratia*.

RAE Real Academia Española.



## **Parte IV**

### ***Índices***



# Comandos de Python que usamos

!=, 31  
", 34  
""" (documentación), 51  
+, 18  
    concatenación  
        de cadenas, 35  
        de sucesiones, 102  
-, 18  
/, 18  
//, 18  
<, 31  
<=, 31  
=, 39  
    y ==, 31  
==, 31  
>, 31  
>=, 31  
# (comentario), 53  
%, 18  
\, 37  
\n, 37  
\\, 38  
\*, 18  
    para cadenas, 37  
\*\*, 18  
' , 34  
abs, 21  
and, 31  
append, 96  
bool, 30  
break, 81  
close, 129  
continue, 82  
def, 58  
divmod, 92  
\_\_doc\_\_ (documentación), 51  
e (notación científica), 23  
elif, 74  
else, 73



- False, 30
- float, 22
- for, 104
  - en filtro, 121
  - en lista, 118
- format, 125
- global, 69
- help, 21
- if, 72
  - en filtro, 121
- import, 49
- in, 101
- input, 51, 52
- insert, 96
- int, 22
  - y `math.trunc`, 25
- key (en `sort` y `sorted`), 141
- keywords (en `help`), 43
- len, 89
  - cadenas, 35
- list, 93
- math, 24
  - `ceil` (techo), 27
  - `cos`, 24
  - `exp` ( $e^x$ ), 24
  - `e` ( $e$ ), 24
  - `floor` (piso), 27
  - `log10` ( $\log_{10}$ ), 26
  - `log2` ( $\log_2$ ), 27
  - `log` ( $\log$ ,  $\ln$ ), 24
  - `pi` ( $\pi$ ), 24
  - `sin` (sen), 24
  - `sqrt` (raíz cuadrada), 24
  - `tan`, 24
  - `trunc`, 25
- max, 108
- min, 108
- next, 258
- None, 45
- not, 31
- open, 129
  - encoding, 129
  - latin-1, 129
  - r, 129
  - utf-8, 129
  - w, 129
- or, 31
- pop, 96
- print, 36
  - end, 124
  - sep, 125
- random, 139
  - `randint`, 139
  - `random`, 139
  - `shuffle`, 140
- range, 100
- read, 130
- return, 62
- reverse, 96
- round, 21

y `math.trunc`, 25

`sep`

en `print`, 125

en `split`, 136

`sort` (clasificar), 140

`sorted` (clasificar), 140

`split`, 134

`sep`, 136

`str`, 30

`sum`, 110

`True`, 30

`tuple`, 90

`type`, 22

`while`, 77

`write`, 133

`yield`, 256

# Índice de figuras y cuadros

|       |                                                                    |     |
|-------|--------------------------------------------------------------------|-----|
| 2.1.  | Entrada, procesamiento y salida. . . . .                           | 11  |
| 2.2.  | Transferencia de datos en la computadora. . . . .                  | 12  |
| 2.3.  | Un byte de 8 bits. . . . .                                         | 13  |
| 2.4.  | Desarrollo de un programa. . . . .                                 | 15  |
| 3.1.  | Traducciones entre matemáticas y el módulo <code>math</code> . . . | 24  |
| 4.1.  | Traducciones entre matemáticas y Python. . . . .                   | 31  |
| 5.1.  | Objetos en la memoria. . . . .                                     | 40  |
| 5.2.  | Asignaciones. . . . .                                              | 42  |
| 6.1.  | Contextos global y de módulos. . . . .                             | 57  |
| 7.1.  | Variables globales y locales. . . . .                              | 67  |
| 8.1.  | Prueba de escritorio. . . . .                                      | 79  |
| 8.2.  | Pasos de Pablito y su papá. . . . .                                | 86  |
| 9.1.  | Efecto del intercambio de variables. . . . .                       | 93  |
| 10.1. | «Triángulo de estrellas» . . . . .                                 | 117 |
| 10.2. | «Arbolito de Navidad» . . . . .                                    | 118 |
| 12.1. | Ordenando por conteo. . . . .                                      | 144 |

|       |                                                          |     |
|-------|----------------------------------------------------------|-----|
| 13.1. | Cálculo de los números de Fibonacci. . . . .             | 158 |
| 13.2. | Esquema del problema de Flavio Josefo. . . . .           | 178 |
| 14.1. | Las torres de Hanoi. . . . .                             | 191 |
| 15.1. | Grafo con 6 vértices y 7 aristas. . . . .                | 198 |
| 15.2. | Asignando colores a los vértices de un grafo. . . . .    | 204 |
| 15.3. | Esquema del algoritmo <i>recorrido</i> . . . . .         | 207 |
| 15.4. | Recorrido <i>lifo</i> de un grafo. . . . .               | 210 |
| 15.5. | Recorrido <i>fifo</i> de un grafo. . . . .               | 213 |
| 15.6. | Laberintos. . . . .                                      | 214 |
| 16.1. | Esquema de la densidad variable. . . . .                 | 220 |
| 16.2. | Gráfico de $\cos x$ y $x$ . . . . .                      | 233 |
| 16.3. | Aproximando la pendiente de la recta tangente. . . . .   | 240 |
| 16.4. | Método de Newton. . . . .                                | 241 |
| 16.5. | Función continua con distintos signos. . . . .           | 243 |
| 16.6. | Método de bisección. . . . .                             | 245 |
| 16.7. | Fractal asociado al método de Newton. . . . .            | 249 |
| 16.8. | Ilustración del conjunto de Mandelbrot. . . . .          | 250 |
| 17.1. | Triángulo de Tartaglia (o de Pascal) de nivel 4. . . . . | 252 |
| 17.2. | Contando los caminos posibles. . . . .                   | 253 |

# Bibliografía

- T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST Y C. STEIN, 2009. *Introduction to Algorithms*. The MIT Press, 3.<sup>a</sup> ed. (pág. 8)
- A. ENGEL, 1993. *Exploring Mathematics with your computer*. The Mathematical Association of America. (págs. 7 y 183)
- E. GENTILE, 1991. *Aritmética elemental en la formación matemática*. Red Olímpica. (pág. 7)
- C. A. R. HOARE, 1961. Algorithm 65: find. *Commun. ACM*, 4(7):321–322. (pág. 149)
- B. W. KERNIGHAN Y D. M. RITCHIE, 1991. *El lenguaje de programación C*. Prentice-Hall Hispanoamericana, 2.<sup>a</sup> ed. (págs. 7 y 8)
- D. E. KNUTH, 1997a. *The art of computer programming*. Vol. 1, *Fundamental Algorithms*. Addison-Wesley, 3.<sup>a</sup> ed. (pág. 8)
- D. E. KNUTH, 1997b. *The art of computer programming*. Vol. 2, *Seminumerical algorithms*. Addison-Wesley, 3.<sup>a</sup> ed. (pág. 8)
- D. E. KNUTH, 1998. *The art of computer programming*. Vol. 3, *Sorting and searching*. Addison-Wesley, 2.<sup>a</sup> ed. (págs. 8 y 153)
- M. LITVIN Y G. LITVIN, 2010. *Mathematics for the Digital Age and Programming in Python*. Skylight Publishing. <http://www.skylit.com/mathandpython.html>. (págs. 8 y 38)

- C. H. PAPADIMITRIOU Y K. STEIGLITZ, 1998. *Combinatorial Optimization, Algorithms and Complexity*. Dover. (pág. 216)
- R. SEDGEWICK Y K. WAYNE, 2011. *Algorithms*. Addison-Wesley, 4.<sup>a</sup> ed. (págs. 8 y 153)
- N. WIRTH, 1987. *Algoritmos y Estructuras de Datos*. Prentice-Hall Hispanoamericana. (págs. 7, 8, 138 y 153)
- S. WOLFRAM, 1988. *Mathematica - A System for Doing Mathematics by Computer*. Addison-Wesley, 1.<sup>a</sup> ed. (pág. 7)

# Índice alfabético

$\varepsilon_{\text{máq}}$ , 221, 291

$\varepsilon_{\text{mín}}$ , 221, 291

$\pi$ , véase también [math](#), 24

$e$ , véase también [e](#) y [math](#), 24, 112

acumulador, 109

algoritmo, 72

de la división, 43, 80

anidar

estructuras, 82

árbol, 199

raíz, 199

archivo de texto, 128

asignación, 39

barrido (técnica), 171

base (cambio de), 169

binomio, 112

bit, 13

bucle (lazo), 77

byte, 13

cadena (de caracteres), 30

clasificación, 140

concatenar, 35

lectura de archivo, 130

subcadena, 102

vacía, 36

y cifras, 170

camino

cantidad de, 252

en grafo, 198

cerrado, 198

entre vértices, 198

longitud, 198

simple, 198

carpeta, véase directorio

ciclo

de Euler, 212, 215

en grafo, 198

cifra, 223

algoritmo, 27, 44, 81

significativa, 127

y cadena, 170

clasificación, 140

estable, 142

código, 15

seudo, 81

- ul style="list-style-type: none;">
- coeficiente
  - binomial, 112
  - multinomial, 264
- comentario
  - en código, 53
- componente
  - de grafo, 199
- concatenar, 35
- conexión
  - de grafo, 198
- contexto
  - de función, 64
  - de módulo, 56
- copia
  - playa y profunda, 98
- CPU, 11
- criba, 175
  - de Eratóstenes, 175
- csv (formato), 136
- densidad (representación)
  - constante, 220
  - variable, 219
- dígito (número), 14
- directorío, 50
- Dirichlet
  - función, 232
- documentación, 53
- ecuación
  - cuadrática, 228
  - diofántica, 170
- editor de textos, 15
- enlace, *véase* referencia
- espacio, *véase también* contexto
- Euclides
  - algoritmo para mcd, 83, 84, 190, 226
  - recursivo, 187
- Euler
  - ciclo, 212, 215
- Euler-Binet
  - fórmula, 159
- exponente, 218
- Fibonacci
  - número, 157, 188, 190
- filtro
  - de lista, 121
- Flavio Josefo
  - problema, 177
- formato, 124
- fractal, 248
- función, 58
  - contexto, 64
  - continua, 231
  - generadora, 257
  - marco, 187
  - piso, 27
  - signo, 75
  - techo, 27
- fusión
  - de listas ordenadas, 149
- Gauss
  - sumas de, 111, 186
- generador
  - función, 257



- ul style="list-style-type: none;">
- grafo, 196
  - arista, 196
  - camino, 198
  - ciclo, 198
  - componente, 199
  - conexo, 198
  - dirigido, 197
  - recorrido, 206
  - simple, 197
  - vértice, 196
- Hanoi (torres), 190
- Horner
  - regla, 167
- identificador, 39
- índice
  - de sucesión, 89
- inmutable, 95
- iterable, véase [cap. 10](#), 132
- iterador, 257
- lazo, 77
- lenguaje, 14
- link, véase referencia
- lista
  - copia, 98
  - profunda, 99
- lista ([list](#)), 88, 93
  - elementos repetidos, 145
  - filtro, 121
  - fusión, 149
  - por comprensión, 118
  - purga, 145
- longitud
  - de camino, 198
- mantisa, 218
- marco, véase contexto
  - de función, 187
- math (módulo), 24
- máximo, véase también [max](#)
  - común divisor, véase mcd
- mcd, 83
  - algoritmo, 226
- mcm, 86
- media, véase también promedio
  - indicador estadístico, 149
- mediana
  - indicador estadístico, 149
- memoria, 11
- método
  - de clase, 96
  - de Newton (y Raphson), 236
- mínimo, véase también [min](#)
  - común múltiplo, véase mcm
- moda
  - indicador estadístico, 149
- módulo, 48
  - estándar, 48
  - math, 24
  - propio, 48
- multiplicación
  - de número y cadena, 37
- mutable, 95
- MVPQC, 130
- MVQSYNQF, 20

- ul style="list-style-type: none;">
- notación
  - científica, 218
- número
  - armónico, 229
  - combinatorio, 112
  - de Fibonacci, 157
  - decimal (float), 18
  - entero (int), 18
  - primo, 84
- objeto (de Python), 39
- palabra
  - clave, 43
  - reservada, 43
- parámetro
  - de función, 64
  - formal, 64
  - real, 64
- Pascal
  - triángulo, 251
- piso (función), 27
- potencia (cálculo de), 179
- precedencia
  - de operador, 19
- primo, 84, 180
- producto interno o escalar, 161
- programa, 14
  - corrida, 14
  - ejecución, 14
- RAE, 292
- raíz
  - de árbol, 199
  - rango (range), 88, 100
  - referencia, 40
  - regla
    - de Horner, 167
    - de oro, 227
  - representación
    - de grafo, 200
    - normal de número, 218
  - sección (de sucesión), 89
  - sección
    - de sucesión, 89
  - secuencia, véase sucesión
  - signo (función), 75
  - sistema operativo, 14
  - sucesión, 88
  - suma
    - acumulada, 113
    - promedio, 109
  - Tartaglia
    - triángulo, 251
  - techo (función), 27
  - técnica
    - de barrido, 171
  - teorema
    - números primos, 176
  - tipo (type)
    - cambio, 23, 102
    - de datos, 29
  - triángulo
    - de estrellas, 117
    - de Pascal, 251
    - de Pitágoras, 20

- de Tartaglia, [251](#)
- tupla ([tuple](#)), [88](#), [90](#)
  - clasificación, [140](#)
- variable, [40](#)
  - global, [56](#), [64](#)
  - local, [56](#), [64](#)
- vértice
  - aislado, [197](#)
- vínculo, *véase* referencia
- von Neumann
  - arquitectura, [13](#)

