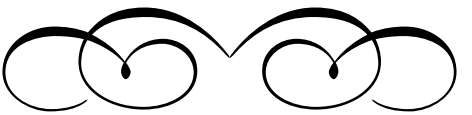
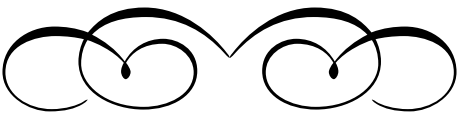


Matemáticas y programación

con Python

N. Aguilera

Febrero de 2014



Contenidos

Capítulo 1

Preliminares

Estos son apuntes de un curso introductorio a la resolución de problemas matemáticos usando la computadora y el lenguaje de programación Python.

La primera parte es el núcleo del curso y la mayoría de sus contenidos son comunes a otros cursos iniciales de programación. Comenzamos viendo tipos de datos, funciones y estructuras de control, pasamos luego al manejo de secuencias, terminando con el manejo elemental de archivos de texto.

En la segunda parte vemos algunos temas más directamente relacionados con matemáticas: números aleatorios y simulación, clasificación y búsqueda, resolución iterativa de ecuaciones de una variable, números enteros, grafos, terminando con recursión.

A diferencia de los cursos tradicionales de programación, veremos 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».

Trataremos de no apartarnos demasiado de las estructuras disponibles en lenguajes procedimentales, aunque será inevitable mencionar algunas particularidades de Python.

En este sentido, es conveniente tener siempre presente que:

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

Por otra parte hay muy poca teoría, que se habrá visto o se verá en otros cursos: lo esencial aquí son los ejercicios.

En casi todos los temas habrá algunos ejercicios rutinarios y otros que no lo son tanto. Algunos pensarán que el material presentado es excesivo, y habrá otros que querrán resolver más ejercicios o ejercicios más avanzados, y para ellos en algunos capítulos se incluye una sección de *ejercicios adicionales*.

1.1. 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 del libro está diseñada para que pueda ser leída sin muchos inconvenientes en «tabletas», y ofrece la ventaja de vínculos (*links*) remarcados en azul.

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 y lo que responde la computadora. En este caso, escribimos en el renglón donde está `>>>` y la respuesta de la computadora aparece sin esos 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 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.

En el apéndice B hay una síntesis de notaciones, convenciones o abreviaturas. En la sección B.5 se explican los «garabatos» como ¶, ✍, ❦... ¡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.

1.2. ¿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 en los últimos años, y es difícil conseguir versiones recientes para las distintas plataformas. De cualquier forma, tiene dos inconvenientes mayores: no tiene un generador nativo de números aleatorios, y carece de elementos gráficos, casi impensado en estos días.

Actualmente no hay lenguajes destinados a la enseñanza de la programación desde un enfoque matemático, que sean gratuitos y de amplia difusión.

Entre los lenguajes más populares hoy,[1] hemos elegido Python (pronunciado *páizon*), `http://www.python.org`, creado por G. van Rossum hacia 1990.

La elección de Python se debe a varios motivos, entre ellos:

- Es fácil hacer los primeros programas, y se puede comenzar con

[1] Ver `http://www.tiobe.com/index.php/content/paperinfo/tpci/`.

un modo prácticamente interactivo escribiendo en una ventana tipo terminal, característica compartida por sistemas como *Mathematica*, *Matlab* o *Maple*.

- Los algoritmos se pueden implementar rápidamente usando funciones predefinidas, y en especial listas con filtros, característica compartida también con, por ejemplo, *Mathematica*.
- Las funciones pueden ser argumentos de otras funciones.
- No tiene límite para el tamaño de enteros.
- Es gratis y 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 incluye al módulo *tkinter* con el que se puede acceder a las facilidades gráficas de Tcl/Tk, permitiendo un entorno gráfico independiente de la plataforma. Estas facilidades son usadas por IDLE, de modo que la integración es estrecha y seguida muy de cerca.
- La distribución oficial incluye una amplia variedad de extensiones (módulos), entre los que nos interesan los de matemáticas (*math* y *random*).

Pero también tiene sus desventajas para la enseñanza:

- No tiene un conjunto pequeño de instrucciones.
- Es posible hacer una misma cosa de varias formas distintas, lo que confunde a los principiantes.
- 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 lista y `reversed` da un iterador.
  - 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.

1.2.1. Censura, censura, censura

Para organizarnos mejor y no enloquecernos ponemos algunas restricciones para el uso de Python. Por ejemplo:

> *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.

Esto nos trae a que hay muchas cosas de programación en general y de Python en particular que no veremos. Entre otras:

- No estudiaremos diccionarios (`dict`) o conjuntos (`set`).

  ✍ Usaremos diccionarios sin mayores explicaciones (apretaremos botones) en los módulos gráficos.

- No veremos cómo definir clases, ni programación orientada a objetos en general, ni —claro— 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 en funciones, aunque los usaremos en las funciones predefinidas como `print` o `sort`.
- Veremos muy poco de generadores (sólo funciones definidas usando `yield` en el capítulo 19).
- No veremos construcciones de programación funcional como `filter`, `map`, y `zip`, aún cuando son comunes a otros lenguajes.

En la página 329 hay un resumen de los comandos de Python que vemos, señalando la primera página donde aparecen. Es posible que falten algunos, pero en general son los únicos admitidos y no deben usarse otros.

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

Terminamos destacando una convención muy importante.

Un párrafo con el símbolo 💣 contiene construcciones de Python que, aunque tal vez válidas, hay que evitar a toda costa. Son construcciones confusas, o absurdas en matemáticas, o ineficientes computacionalmente, o no se usan en otros lenguajes, etc.:

> *Las construcciones señaladas con 💣 no deben usarse en este curso.*

1.3. La versión 2013

Estas notas se van modificando en base a la experiencia que se gana en el dictado del curso, de modo que en el transcurso de los años hay temas que cambian, otros se eliminan, otros se agregan, y otros... ¡reaparecen!

Esta versión 2013 no es diferente. Entre los cambios más destacados:

- Las notas se dividieron en dos partes: una más básica, común a casi cualquier primer curso de programación, y otra de tópicos más cercanos a matemáticas.
- El capítulo 11 intenta unificar conceptos dispersos en versiones anteriores, tratando de llamar la atención de los alumnos. Uno de los hilos conductores es la regla de Horner, que antes estaba en el capítulo de cálculo numérico.
- Dejamos el estudio de grafos con pesos, pues son técnicas difíciles de aplicar directamente a otros problemas.
- Desaparecieron muchas instrucciones de Python (`del`, `remove`, `join`, `extend` y otras), tratando de reducir la cantidad de instrucciones disponibles y reforzar el aprendizaje del uso de índices.
- En cambio, apareció `yield` (y `next`) en la generación de objetos combinatorios.

1.4. Comentarios

- Los temas y formas de abordarlos están inspirados en Wirth (1987), Kernighan y Ritchie (1991), y los tres primeros volúmenes de Knuth (1997a; 1997b; 1998) en lo referente a programación, y de 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 las presentaciones de los libros de *Mathematica* (Wolfram, 1988, y siguientes ediciones), y el mismo *tutorial de Python*.

- El libro de Litvin y Litvin (2010) trabaja con Python con una filosofía similar a la nuestra, pero a nivel de secundaria y está en inglés.

  Otro libro con una filosofía parecida es el de Sedgewick y Wayne (2011). Es algo más avanzado y toca temas que no vemos (y al revés). También está en inglés y trabaja con Java.
- A los que quieran aprender más les sugerimos los libros ya mencionados de Wirth (1987) y Sedgewick y Wayne (2011) y, para un estudio más profundo, los de Knuth (1997a; 1997b; 1998).
- Hemos tratado de incluir referencias bibliográficas sobre temas y problemas particulares, pero muchos pertenecen al «folklore» y es difícil determinar el origen.
- La mayoría de las referencias históricas están tomadas de MacTutor History of Mathematics, Wolfram MathWorld y WikipediA.
- A quienes les haya «picado el bichito» les sugerimos tomar problemas o participar en las competencias estudiantiles de la ACM (Association for Computing Machinery) en las que estudiantes de nuestro país participan exitosamente desde hace varios años.

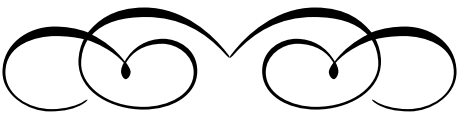
1.5. Agradecimientos

Agradezco a los muchos docentes y alumnos que influyeron en los cambios y la detección de errores (tipográficos o conceptuales).

Especialmente quiero agradecer a Luis Bianculli, Jorge D'Elía, María Fernanda Golobitsky, Egle Haye, Alberto Marchi, Martín Marques y Marcela Morvidone, con quienes he tenido el placer de trabajar.

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

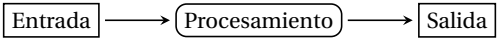


Figura 2.1: Esquema de entrada, procesamiento y salida.

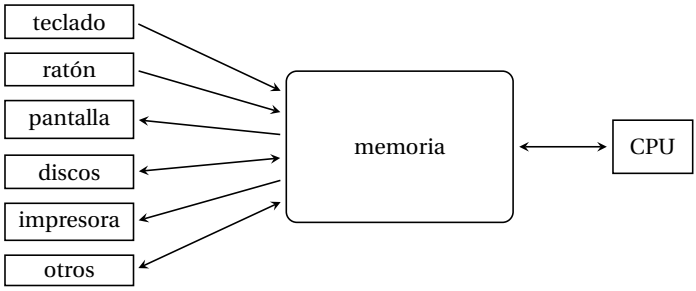


Figura 2.2: Esquema de transferencia de datos en la computadora.

computadora.

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.

Aunque las computadoras modernas suelen tener más de una CPU, en nuestro modelo consideraremos que hay una sola. Del mismo modo, consideraremos que la CPU lee y escribe los datos de la memoria *secuencialmente* —uno a la vez— a pesar de que las computadoras actuales pueden ir leyendo y escribiendo simultáneamente varios datos.

Finalmente, las instrucciones que sigue la CPU forman parte de los datos en la memoria que se procesarán. Es decir, la CPU lee instrucciones en la memoria que le dicen qué otros datos (que pueden ser más instrucciones) tomar de la memoria y qué hacer con ellos.

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

- Hay una única CPU,
- que sólo intercambia datos con la memoria,

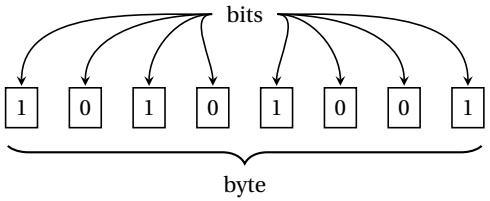


Figura 2.3: Un byte de 8 bits.

- este intercambio es secuencial, y
- las instrucciones que recibe la CPU forman parte de los datos en la memoria.

❦ *El modelo, con pocos cambios, se debe a John von Neumann (1903–1957), quien se interesó inicialmente en lógica, teoría de conjuntos, de la medida, y mecánica cuántica, tocando luego temas de análisis funcional, teoría ergódica, siendo fundador de la teoría de juegos.*

En sus últimos años también tuvo influencia decisiva en ecuaciones en derivadas parciales y en teoría de autómatas, en la que sintetizó sus conocimientos e ideas de lógica y grandes computadoras electrónicas.

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, 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.

Ejercicio 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,...

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.

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 ristras de ceros y unos. Las instrucciones para la máquina se escriben como sentencias —de acuerdo a las reglas 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 *fuente* o *código fuente* o smplemente *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

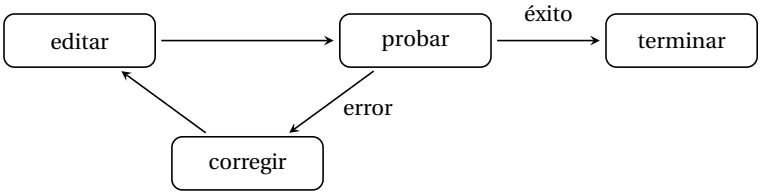


Figura 2.4: Esquema del desarrollo de un programa.

(casualmente) *python* hace la traducción de humano a binario e indica a la CPU que realice la tarea. Esto se hace a través de otro programa llamado *terminal* en sistemas Unix: allí escribimos las instrucciones y luego le pedimos al programa python que las ejecute.

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 programación 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 (3.3 al escribir estas notas), que puede obtenerse del sitio oficial de Python, donde hay instalaciones disponibles para los principales sistemas operativos.

Algunas observaciones:

- La versión que usaremos no es compatible con versiones anteriores como la 2.7, y hay que tener cuidado cuando se hacen instalaciones o se consulta documentación (ejemplos, apuntes, libros, etc.) de internet.
- Muchos sistemas Unix tienen Python preinstalado, pero posiblemente se trate de una versión anterior. Para verificar si la versión 3.3 está instalada, puede ponerse en la terminal de Unix:

  ```
  which python3
  ```

  y si no aparece la ubicación, debe instalarse.

En algunos casos, IDLE se ofrece como instalación separada (que debe instalarse). Además, IDLE usa el programa Tcl de ActiveState, y habrá que tener instaladas versiones de Python, IDLE y Tcl compatibles.

La forma de iniciar IDLE depende del sistema operativo y la instalación, pero finalmente debe aparecer la terminal de IDLE con un cartel similar a

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 13:52:24)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

Es importante verificar que aparezca anunciado `Python 3.3` *en alguna parte: versiones menores seguramente 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

```
>>> 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 rapto 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` (previous o *previo*) o `alt-n` (next o *siguiente*) 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 de IDLE para ver qué otros atajos hay o modificarlos a gusto.

Con `ctrl-d` (fin de datos) 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

Ejercicio 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 «123×45»),

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.

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

Ejercicio 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`. ¶

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

a) `12 / 0` *b*) `34.5 / 0` *c*) `67 // 0` ¶

Ejercicio 3.4. ¿Cuánto es 0^0 (cero a la cero) según las matemáticas? ¿Y según Python? ¶

Ejercicio 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` *c*) `4 - 3 + 2` ¶

Cuando no usamos paréntesis, tenemos que tener cuidado con la *precedencia* (cuál se aplica primero) de los operadores. Por ejemplo, si en matemáticas ponemos $2 + 3 \times 4$, sabemos que tenemos que calcular primero 3×4 y a eso agregarle 2, o sea, « × » tiene mayor precedencia que « + ». Si en cambio tenemos $2 - 3 + 4 - 5$, evaluamos de izquierda a derecha. La cosa se complica si consideramos $3/4 \times 5$: en matemáticas no queda claro si nos referimos a $(3/4) \times 5$ o a $3/(4 \times 5)$.

Python sigue reglas similares, aunque evalúa `3 / 4 * 5` de izquierda a derecha, como en el caso de sumas y restas. En el curso trataremos de evitar este tipo de construcciones agregando paréntesis aunque sean redundantes.

✍ Volvemos a este tema en el ejercicio 4.9.

Ejercicio 3.6. 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)` ¶

Ejercicio 3.7. 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.

Ejercicio 3.8. Analizar los resultados de:

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

Ejercicio 3.9. 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)`. ¶

Ejercicio 3.10. Python puede trabajar con enteros de cualquier tamaño, pero los tamaños de los decimales es limitado.

a) Poner `123 ** 456` viendo que obtenemos un número muy grande.
b) Poner `123.0 ** 456.0` viendo que obtenemos un error (el mensaje de error indica `Result too large` o *resultado demasiado grande*.)
c) ¿Qué pasará si ponemos `float(123) ** float(456)`?
d) Probar también con `123.0 ** 456` y `123 ** 456.0`.

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

Más adelante estudiaremos con más cuidado estos temas. ¶

Matemáticas:	π	e	$\sqrt{x}$	sen	cos	tan	log
Python:	`pi`	`e`	`sqrt(x)`	`sin`	`cos`	`tan`	`log`

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

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 *conestasletras*.

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 $\pi = 3.14159\ldots$ y $e = 2.71828\ldots$

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, π se escribe `math.pi` en Python.

Hay otras formas de «importar» módulos o parte de sus contenidos, pero:

> *nosotros sólo usaremos la forma*
>
> ```
> import nombre_del_módulo
> ```
>
> *importando siempre todo el contenido del módulo.*

Ejercicio 3.11. Poner `import math` en la terminal, y luego realizar los siguientes apartados:

a) Poner `help(math)`, para ver las nuevas funciones disponibles.

b) Evaluar `math.sqrt(2)`. ¿Qué pasa si ponemos sólo `sqrt(2)` (sin `math`)?

c) Desde las matemáticas, ¿es $\sqrt{3}$ entero?, ¿y $\sqrt{4}$? ¿De qué tipo son `math.sqrt(3)` y `math.sqrt(4)`? ¶

Ejercicio 3.12. Usando el módulo *math*:

a) Ver qué hace `math.trunc` usando `help(math.trunc)`.

b) ¿Cuál es la diferencia entre `round` y `math.trunc`? Encontrar valores del argumento donde se aprecie esta diferencia.

c) ¿Cuál es la diferencia entre `int(x)` y `math.trunc(x)` cuando `x` es un número decimal? ¶

Ejercicio 3.13 (funciones trigonométricas). Como en matemáticas avanzadas, en Python las funciones trigonométricas toman los argumentos en radianes. En todo caso podemos pasar de grados a radianes multiplicando por $\pi/180$. Así, $45° = 45° \times \pi/180° = \pi/4$ (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:

a) Evaluar `math.pi`.

b) Sin usar calculadora o compu, ¿cuáles son los valores de $\cos 0$ y $\operatorname{sen} 0$? Comparar con la respuesta de Python a `math.cos(0)` y `math.sin(0)`.

c) Calcular $\operatorname{sen} \pi/4$ usando Python.

d) Calcular $\tan 60°$ usando `math.tan` con argumento $60 \times \pi/180$.

e) Calcular seno, coseno y tangente de $30°$ usando Python (las respuestas, por supuesto, deberían ser aproximadamente $1/2$, $\sqrt{3}/2$ y $\sqrt{3}/3$).

f) ¿Cuánto es $\tan 90°$?, ¿y según Python?

✍ Python no puede evaluar $\tan 90°$ exactamente: tanto `math.pi`

como `math.tan` son sólo aproximaciones a π y tan (respectivamente). ¶

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

Ejercicio 3.14 (exponenciales y logaritmos). En la terminal de IDLE, poner `import math` y realizar los siguientes apartados:

a) Calcular `math.e`.

b) Usando `help`, averiguar qué hace la función `math.log` y calcular `math.log(math.e)`. ¿Es razonable el valor obtenido?

c) 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?

d) Calcular $a = \log_{10} 5$ aproximadamente, y verificar el resultado calculando 10^a (¿cuánto debería ser?).

e) Ver qué hace `math.exp`. Desde la teoría, ¿que 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., ±1, ±10, ±100). ¶

Ejercicio 3.15. 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 ±1, ±2.3 y ±5.6.

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

Ejercicio 3.16 (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 3.10 y la nota al final de éste.

b) Encontrar la cantidad de cifras en base 2 de 2^{64} y de 1023. ¶

3.3. Comentarios

- La presentación como calculadora es típica en sistemas interactivos como *Matlab* o *Mathematica*. En particular, aparece en el *tutorial de Python*.
- Más adelante —en el capítulo 15— veremos cómo hacer para que Python se comporte como una calculadora gráfica también.

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, para poder trabajar. 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 construidos

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».

Ejercicio 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 ≠ 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 veremos en el capítulo 5.*

Ejercicio 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)`. ¶

Ejercicio 4.3. Conjeturar y verificar en Python:

a) `not True` *b*) `True and False`
c) `True or False` *d*) `False and (not True)` ¶

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

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

son de tipo `bool`. ¶

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

Ejercicio 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` ¶

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

a) `123` con `123.0`,

b) `123` con `123 + 1.0e-10`,

✍ `1.0e-10` es la versión de Python de lo que en matemática es 1.0×10^{-10}, y no está relacionado directamente con el número $e = 2.71828\ldots$.

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, muchas veces 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 4.9.

Python también usa esta convención:

Ejercicio 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 (pues las operaciones lógicas de conjunción y disyunción son conmutativas), aceptaremos el uso de «cortocircuitos» porque su uso está muy difundido en programación.* ¶

Ejercicio 4.9 (precedencias).

a) Evaluar `3 / 4 * 5` y decidir si corresponde a `(3 / 4) * 5` o a `3 / (4 * 5)`.

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

c) Conjeturar el valor de `True or False and False`, luego evaluar la expresión en Python y decidir si corresponde a `(True or False) and False` o a `True or (False and False)`.

Decidir si en Python hay precedencias entre `and` y `or` o si se evalúan de izquierda a derecha (como `3 / 4 * 5`).

✍ En matemáticas no hay precedencias entre la conjunción y la disyunción: siempre deben ponerse paréntesis al mezclarse.

☞ *En todos los ejemplos anteriores* hay *que poner paréntesis.*

☞ *Más vale poner algunos paréntesis redundantes antes que dejar algo de significado dudoso.* ¶

Otra diferencia importante con las matemáticas es que a los efectos de evaluaciones lógicas, Python considera que todo objeto tiene un valor verdadero o falso. Recíprocamente, para operaciones aritméticas `False` y `True` tienen valores numéricos (0 y 1 respectivamente).

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'`.

Ejercicio 4.10. Ingresar en la terminal `'Ana Luisa'` y averiguar su tipo poniendo `type('Ana Luisa')`. ¶

Ejercicio 4.11. Ver que para Python `'Ana Luisa'` y `"Ana Luisa"` son lo mismo usando `==`. ¶

Ejercicio 4.12. 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'` ¶

Ejercicio 4.13. Como vimos en el caso de `int` y `float` (ejercicio 3.8), 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 3.8.

d) A veces el resultado puede llevar a error:

```
bool(str(False))
```

✍ Esto es particularmente fastidioso cuando se está leyendo de un archivo de texto. ¶

Ejercicio 4.14 (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')` ¶

Ejercicio 4.15 (concatenación). *Concatenar* es poner una cadena a continuación de otra, para lo cual Python usa «`+`», el mismo símbolo que para la suma de números.

a) Evaluar:

i) `'mi' + 'mama'` *ii*) `'mi' + '␣' + 'mama'`

✍ Para evitar confusiones, a veces ponemos ␣ para indicar un espacio en blanco.

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. ¶

Ejercicio 4.16 (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 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. ¶

Ejercicio 4.17.

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'`.

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

c) 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` ¶

Ejercicio 4.18 (`isinstance` I). Para cerrar el tema de los tipos de datos, en este ejercicio vemos la función `isinstance`, que de alguna manera es la inversa de `type`.

a) Usando `help(isinstance)`, ver qué hace esta función.

b) Evaluar

i) `isinstance(1, int)`

ii) `isinstance(1.0, int)`

iii) `isinstance('mama', float)`

iv) `isinstance(True, int)` ¶

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 12.

4.4. `print` (imprimir)

`print` nos permite imprimir en la terminal expresiones que pueden involucrar elementos de distinto tipo.

Ejercicio 4.19.

a) Al ingresarlos en la terminal, ¿cuál es la diferencia entre `123`, `print(123)`, `'123'` y `print('123')`?

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

i) `print` *ii*) `print()`

iii) `print('')` *iv*) `print('␣')`

c) También 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)
```

d) Evaluar y observar las diferencias entre:

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

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

Ejercicio 4.20. Aunque matemáticamente no tiene mucho sentido, la multiplicación de un entero por una cadena que vimos en el ejercicio 4.17 es útil para imprimir.

Evaluar:

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

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

Ejercicio 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')` ¶

Ejercicio 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. ¶

Como vimos, `\` se usa para indicar que el renglón continúa, o, en la forma `\n`, para indicar que debe comenzarse un nuevo renglón.

Si queremos que se imprima «\», tenemos que poner `\\` dentro del argumento de `print`.

Ejercicio 4.23.

a) Comparar `print('\')` con `print('\\')`.

b) Conjeturar y luego verificar los resultado de `print('/\')` y de `print('//\\')`.

c) Imprimir una «casita» usando `print`:

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

¶

4.5. Comentarios

- El ejercicio 4.23 está tomado de Litvin y Litvin (2010).

Capítulo 5

Variables y asignaciones

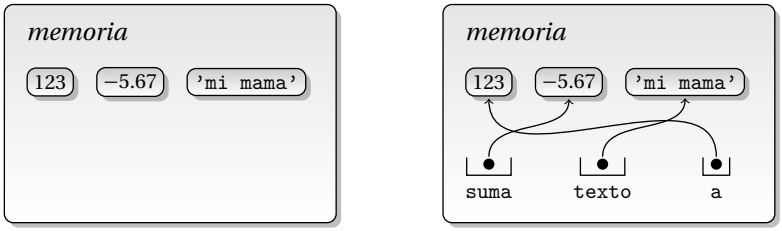
Frecuentemente queremos usar un mismo valor varias veces en los cálculos, o guardar un valor intermedio para usarlo más adelante, o simplemente ponerle un nombre sencillo a una expresión complicada para referirnos a ella en adelante. A estos efectos, muchas calculadoras tienen memorias para conservar 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
```



a) Objetos en la memoria. *b*) Asignaciones.

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 ristras 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* (*link*) en una página de internet.

Ejercicio 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 hubiera. 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:

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

Ejercicio 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:

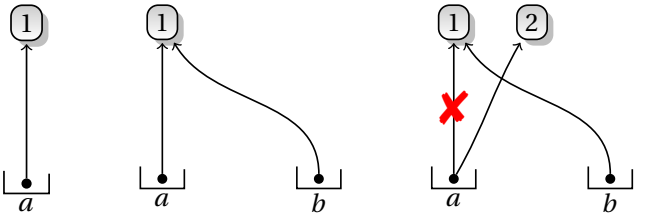


Figura 5.2: Ilustración de asignaciones.

a)
```
a = 1
a
```

b)
```
a = 1
a = a + 1
a
```

c)
```
a = 1
a = a + a
a
```

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

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

f)
```
a = 3
b = 2
a = a + b
b = a - b
a
b
```
¶

Los identificadores 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 3.6).

Ejercicio 5.3.

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

i) `PepeGrillo` *ii*) `Pepe_Grillo` *iii*) `Pepe_Grillo`
iv) `Pepe-Grillo` *v*) `Pepe\Grillo` *vi*) `Pepe/Grillo`
vii) `Grillo,Pepe` *viii*) `Pepe12` *ix*) `34Pepe`

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

Sugerencia: asignarlos a valores distintos y comprobar que son diferentes. ¶

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

a) Para encontrarlas, ponemos en la terminal `help()` y a continuación `keywords` (*keyword* = palabra clave).

b) Esto quiere decir que no podemos poner `and = 5`: verificarlo.

c) 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:

```
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`. ¶

Ejercicio 5.5. Dados los enteros a y b, $b > 0$, el *algoritmo de la división* encuentra enteros q y r, $0 \le r < b$ tales que $a = qb + r$, aún cuando a no sea positivo, y esto se traduce en Python poniendo

```
q = a // b      y      r = a % b.
```
(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.

a) ¿Qué pasa si `b` es 0?, ¿y si `a` y `b` son ambos 0?

b) ¿Qué hace Python cuando `b` es entero pero negativo? (o sea: ¿qué valores de `q` y `r` se obtienen?).

c) 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).

d) 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.

e) Dar ejemplos «de la vida real» donde tenga sentido considerar:

i) `a` decimal y `b` entero positivo,

ii) `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.

f) ¿Qué hace Python cuando `b` es negativo y decimal? ¶

Ejercicio 5.6 (cifras II). En el ejercicio 3.16 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.

a) 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 3.16. ¶

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.

Ejercicio 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 distinguirlas 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 que nos permite guardar grupos de instrucciones. En Python estos archivos se llaman *módulos*, y para distinguirlos en estas notas indicamos sus nombres `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.

Ejercicio 6.1 (Hola Mundo).

a) Abrir una ventana nueva en IDLE distinta de la terminal (menú *File* → *New Window*), escribir en ella `print('Hola Mundo')` en un único renglón, y guardar en un archivo con nombre *elprimero.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`.
- ✍ Para evitar problemas, guardaremos *todos* nuestros módulos en el mismo directorio (veremos más adelante por qué).

  Por prolijidad es mejor que el directorio no sea el directorio principal. Por ejemplo, podrían ponerse en un directorio *Python* dentro del directorio *Documentos* (o similar) del usuario.

b) 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: *más vale que sobre y no que falte, lo que abunda no daña,...*
- ✍ El uso de `"""` es similar al de las comillas simples `'` y dobles `"` para encerrar cadenas de caracteres, con algunas diferencias. Por ejemplo, no es necesaria la barra invertida « `\` » al final de un renglón para indicar que el texto continúa en el siguiente.
- ✍ Recordar de no poner más de 80 caracteres por renglón.
- ✍ La *guía de estilos de Python* sugiere que la documentación sea un renglón corto, seguido eventualmente de otros separados por renglones en blanco. Si hay más de un renglón, se sugiere

dejar un renglón en blanco antes del `"""` 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`.

✍ Usando `os.getcwd()` se puede determinar el *directorio de trabajo*. Nosotros no veremos el módulo estándar *os*. ¶

6.2. Ingreso interactivo de datos

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

Ejercicio 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) Al ingresar el nombre no es necesario poner comillas: Python toma cualquier entrada como cadena de caracteres.

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 4.15).

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. ¶

Ejercicio 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, viendo si se obtiene el resultado esperado.

b) ¿Qué resultado esperarías si las entradas fueran `2` y `3`?

Ejecutar el módulo, viendo 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 usar el símbolo «`#`»: Python ignora este símbolo y todo lo que le sigue en ese renglón. Esta acción se llama *comentar* el texto.

Ejercicio 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 llamás?')`. ¿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.

Ejercicio 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 se pierden las asignaciones anteriores.* ¶

Ejercicio 6.6. Resolver los siguientes apartados en una nueva sesión de IDLE.[(1)]

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).

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)`.

☞ *Se puede importar cualquier cantidad de módulos, pero*

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

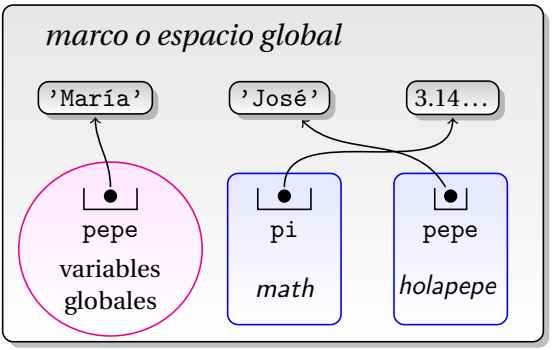


Figura 6.1: Marco global y marcos de módulos en el ejercicio 6.7.

ejecutar sólo 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 *espacio* (o *contexto* o *marco*) determinado por el módulo, y que son *locales* a él. Este espacio tiene el mismo nombre del módulo (como *math*), y por eso se denomina *espacio de nombre* o *nombrado*. Los objetos creados fuera de estos espacios se dicen *globales*.

Veamos cómo es esto, ayudándonos con la figura 6.1.

Ejercicio 6.7 (espacios de nombres).

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.

↬ `math.pi` *es una variable en el espacio* 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 espacio 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*. 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

Ejercicio 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 como en los módulos (ejercicio 6.7).*

> ✍ El tema se explica con más detalle en la sección 7.4. ¶

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

Ejercicio 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 hubiera).

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. ¶

Ejercicio 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).

Ejercicio 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`. ¶

Ejercicio 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`*.* ¶

Ejercicio 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 ± 1, ± 2 y ± 3. ¶

7.2. Funciones numéricas

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

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

a) Expresar en Python la ecuación que relaciona c y f.
b) En la terminal de IDLE, usar la expresión anterior para encontrar `c` cuando `f` es `0`, `10`, `98`.
c) 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*). ¶

Ejercicio 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, $12° \; 34' \; 56.78''$ son 0.21960 . . . radianes. ¶

Ejercicio 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 3.16.

b) Usando las ideas del ejercicio 5.6. ¶

7.3. Python y variables lógicas

En esta sección tratamos de deshacer los entuertos de Python con las variables lógicas, apoyándonos en la función `isinstance` (ejercicio 4.18).

Ejercicio 7.10.

a) Definir una función `esbool` que determine si el argumento ingresado es una variable lógica o no *desde las matemáticas* (no de Python), retornando verdadero o falso. Por ejemplo:

argumento	`'mi mama'`	`1`	`1.2`	`1.0`	`True`
debe dar	`False`	`False`	`False`	`False`	`True`

b) Definir una función `esnumero` que determine si el argumento ingresado es un número. Por ejemplo:

argumento	`'mi mama'`	`1`	`1.2`	`1.0`	`True`
debe dar	`False`	`True`	`True`	`True`	`False`

Ayuda: usar el apartado anterior y operadores lógicos como `and` y `not`.

c) Definir una función `esentero` que determine si el argumento ingresado es un número entero. Por ejemplo:

argumento	`'mi mama'`	`1`	`1.2`	`1.0`	`True`
debe dar	`False`	`True`	`False`	`True`	`False`

Atención: ver el resultado de `int(True) == True`.

d) Definir una función `esnatural` que determine si el argumento ingresado es un número entero y positivo. ¶

7.4. 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 espacio o marco, 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 marco de una función hay objetos como instrucciones y variables, que son locales a la función.

Los argumentos (si los hubiera) 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 *espacios* definidos por módulos),...
- ... salvo que se declare como *global* con `global`, y en este caso será... ¡global!

Ejercicio 7.11. 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

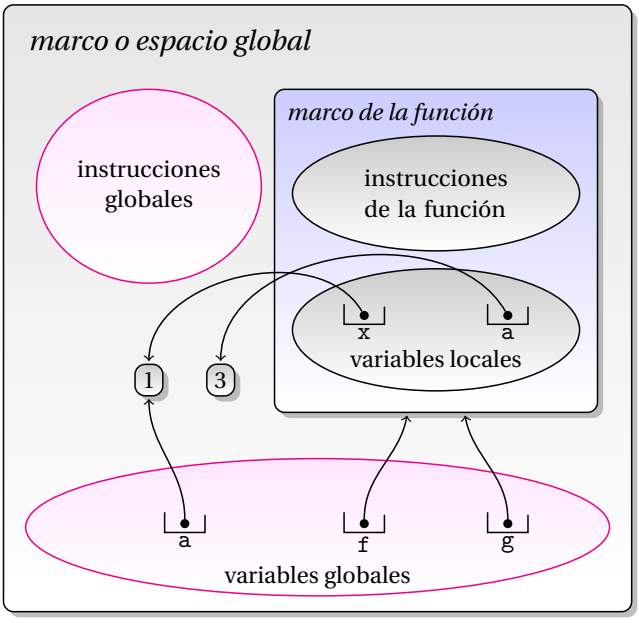


Figura 7.1: Variables globales y locales.

```
a = 2
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 7.11 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 `2`, 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.

Ejercicio 7.12.

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)
```
a = f(1234)
a == 1234
x
```

ii)
```
a = 1234
b = f(a)
a == b
```

iii)
```
x = 12
y = f(34)
x == y
```

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

```
def f(x):
    """Usa una variable global a."""
    return x + a  # a es global, no asignada en f
```

y explicar los resultados de los siguientes:

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

ii)
```
a = 1
f(2)
```

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

☞ *Si* `a` *no tiene valor definido, la llamada a* `f(a)` *no tiene sentido.*

✍ Debemos tener presente que cuando definimos `f` mediante

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

`a` es local a la estructura, y puede o no haber otra variable con el mismo identificador `a` fuera de `f`.

Ver también el ejercicio 7.11 y el apartado *h*).

c) Reiniciar la terminal, poner

```
def f(x):
    """Trata de cambiar la variable a."""
    a = 5   # a es local porque se asigna
    return x + a
```

y explicar los resultados de los siguientes:

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

ii)
```
a = 2
f(1)
a
```

d) Reiniciar la terminal, poner

```
def f(x):
    """Usa una variable global a."""
    b = a       # b es local y a es global
    return x + a + b
```

y volver a resolver las preguntas del apartado anterior.

e) Repetir el apartado *c*) con

```
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)                    ?
```

¿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)
```
f(1)
a
```

ii)
```
a = 2
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. ¶

Ejercicio 7.13. 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. ¶

Ejercicio 7.14. 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? ¶

Ejercicio 7.15. 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))` ¶

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 *está relacionada con el nombre de Abu Ja'far Muhammad ibn Musa al-Khwarizmi' (~780–850). A su vez, la palabra* álgebra *está relacionada con su obra.*

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.

Ejercicio 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*).

a) Estudiar la construcción y probar la función con distintos argumentos, numéricos y no numéricos.

b) Cambiar la definición de la función de modo que *retorne* verdadero o falso en vez de imprimir, cambiando `print` por `return` adecuadamente.

c) 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
```
¶

Ejercicio 8.2 (piso y techo). Recordando las definiciones de *piso* y *techo* (en el ejercicio 3.15), la función `piso` del módulo *ifwhile* imprime los valores correspondientes al piso de un número real usando una estructura `if... elif... else`.

a) Estudiar las instrucciones de la función y comparar los resultados de la función `piso` con los de la función `math.floor` para ±1, ±2.3 y ±5.6.

b) Cambiando `print` por `return` (en lugares apropiados), modificar `piso` de modo de *retornar* el valor del piso del argumento.

c) Construir una función `techo` para retornar el techo de un número real. ¶

Ejercicio 8.3 (signo de un número real). La función signo : $\mathbb{R} \to \mathbb{R}$ se define como:

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

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

✍ Observar que $|x| = x \times \operatorname{signo} x$ para todo $x \in \mathbb{R}$, y un poco arbitrariamente definimos $\operatorname{signo} 0 = 0$.

✍ En este y otros ejercicios similares, si no se explicita «retornar» o «imprimir», puede usarse cualquiera de las dos alternativas (o ambas). ¶

Ejercicio 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. ¶

Ejercicio 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.

Ejercicio 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
```

a) Estudiar las instrucciones de la función (sin ejecutarla).

b) 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 >= b`: verdadero			
3	`r = r - b`			7
4	`r >= b`: verdadero			
5	`r = r - b`			4
6	`r >= b`: verdadero			
7	`r = r - b`			1
8	`r >= 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 los que hemos visto hasta ahora, ni demasiado complicada como varios de los 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*).

c) 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).

d) Ejecutar la función, verificando que coinciden los resultados de la función y de las pruebas de escritorio.

e) 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? ¶

Ejercicio 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 $a = q\,b + r$.

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 mantenemos el invariante $a = qb + r$ pues $a = 0b + a = 1b + (a - b) = \cdots$

b) Extender la función anterior para considerar también el caso en que a sea negativo (siempre con $b > 0$ y usando sólo sumas y restas).
Sugerencia:

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

¶

Ejercicio 8.8 (cifras III). En los ejercicios 3.16 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 *seudo código*: 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 « ← » 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.

a) Estudiar las instrucciones de la función, y probarla con distintas entradas enteras (positivas, negativas o nulas).

b) ¿Qué pasa si se elimina el renglón `n = abs(n)`?

c) ¿Habría alguna diferencia si se cambia el lazo principal por

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

?

d) Ver que los resultados de los ejercicios 3.16 y 5.6 coinciden con el obtenido al usar `cifras`.

e) 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 $\mathrm{mcd}(a, b)$, se define[1] como el máximo elemento del conjunto de divisores comunes de a y b:

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

✍ 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 mín $\{a, b\}$), por lo que $\mathrm{mcd}(a, b)$ está bien definido.

✍ La denominación *máximo común divisor* es la de uso tradicional en matemáticas. Más recientemente en las escuelas de nuestro país se la ha cambiado a *divisor común máximo*: me reservo mi opinión al respecto.

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

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

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

Cuando $\mathrm{mcd}(a, b) = 1$ 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 ± 1 y $\pm p$. Los primeros primos son 2, 3, 5, 7 y 11.

Algunos autores consideran que -2, -3, etc. también son primos, pero nosotros los consideraremos siempre mayores que 1.

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:

[1] ¡Como su nombre lo indica!

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 su obra* Los elementos*, Euclides de Alejandría (alrededor de 325–265 a. C.) considera a los números como longitudes de segmentos (no existía el* 0 *ni los números negativos), y de allí que tratara dentro de la geometría cuestiones que hoy consideraríamos como de teoría de números.*

✍ 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 *conmensurabilidad* 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.

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

```
mientras a ≠ b:
    si a > b:
        a ← a − b
    en otro caso:    # acá es b > a
        b ← b − a
# acá es a = b
retornar a
```
(8.1)

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
```

¶

Ejercicio 8.10 (algoritmo de Euclides II). Invirtiendo el proceso que hicimos 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 `mcd2` 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 `mcd2` 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`? ¶

Ejercicio 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 $m \in \mathbb{Z}$ y $n \in \mathbb{N}$ de modo que $\frac{p}{q} = \frac{m}{n}$ y $\mathrm{mcd}(m, n) = 1$.

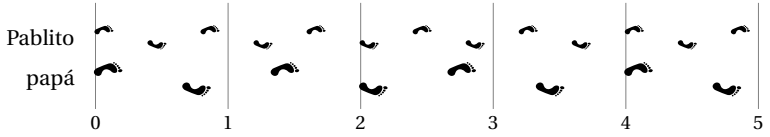


Figura 8.2: Pasos de Pablito y su papá.

✍ ¡Atención con los signos de p y q! ¶

Ejercicio 8.12. El *mínimo común múltiplo* de $a, b \in \mathbb{N}$, $\mathrm{mcm}(a, b)$, 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

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

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

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

Ejercicio 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?

✍ 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.

✍ El ejercicio no requiere programación, lo que postergamos para el ejercicio adicional 8.14. ¶

8.4. Ejercicios adicionales

Ejercicio 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

Ayuda: pongamos $m_b = n_b/d_b$ y $m_p = n_p/d_p$ para simplificar. Si Pablito hace h_b pasos, la cantidad de metros recorridos será

$$h_b \times \frac{m_b}{p_b},$$

y de modo similar para su papá. Como ambos deben realizar un número par de pasos y recorrer la misma cantidad de metros, ponemos $h_b = 2k_b$, $h_p = 2k_p$ y buscamos los valores positivos más chicos posibles de modo que

$$2 \times k_b \times \frac{m_b}{p_b} = 2 \times k_p \times \frac{m_p}{p_p}. \tag{8.2}$$

Como $m_b = n_b/d_b$, $m_p = n_p/d_p$, simplificando llegamos a

$$k_b \times n_b \times d_p \times p_p = k_p \times n_p \times d_b \times p_b \tag{8.3}$$

donde las incógnitas son k_b y k_p que deben ser enteros positivos lo más chicos posibles, y por lo tanto las cantidades en (8.3) deben coincidir con

$$\text{mcm}(n_b \times d_p \times p_p, n_p \times d_b \times p_b),$$

a partir del cual se pueden encontrar k_b y k_p, y luego la cantidad de metros recorridos usando la ecuación (8.2). ¶

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 14.2 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*.

Ejercicio 9.1 (índices de sucesiones). Si la sucesión `a` tiene n elementos, éstos pueden encontrarse individualmente con `a[i]`, donde i es un *índice*, $i = 0, \ldots, n-1$. Índices negativos cuentan desde el final hacia adelante ($i = -1, \ldots, -n$), 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]` ¶

Ejercicio 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 continuo.

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 4.15).

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 «`)`».

Ejercicio 9.3 (tuplas).

a) Poner

```
a = 123, 'mi mamá', 456
```

en la terminal, y resolver los siguientes apartados.

i) Encontrar el valor, el tipo (con `type`) y la longitud (con `len`) de `a`.

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
```

¶

Ejercicio 9.4 (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
```

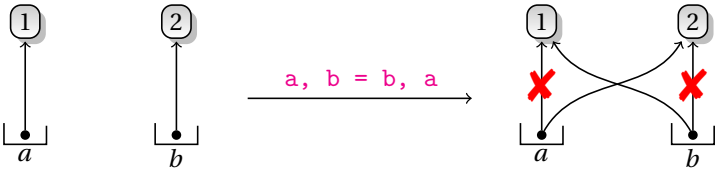


Figura 9.1: Efecto del intercambio de variables.

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.

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 `mcd2` (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? ¶

Ejercicio 9.5. Hay veces que es conveniente tener más de un valor como resultado.

Por ejemplo, en el algoritmo de la división (que ejercicios 5.5 y 8.7) a veces 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). ¶

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 9.4, para hacer asignaciones múltiples como en el ejercicio 9.3.*d*), y ocasionalmente como argumentos o resultados de funciones como `isinstance` o `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)]]`.

Ejercicio 9.6 (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 9.4).

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`. ¶

Ejercicio 9.7 (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.

a) Poner en la terminal `a = [1, 2, 3]` y verificar el valor de `a`.

b) Poner `a[0] = 5` y volver a verificar el valor de `a`.

c) Poner `b = a` y verificar el valor de `b`.

d) Poner `a[0] = 4` y verificar los valores de `a` y `b`.

e) 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`*.*

f) 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 9.14.

Ejercicio 9.8. 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.

Ejercicio 9.9 (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) `a.append('b')` *ii*) `a.pop()`
iii) `a.pop(0)` *iv*) `a.insert(3, 's')`
v) `a.insert(-1, 'x')` *vi*) `a.reverse()`

✍ 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.

Ejercicio 9.10. 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, podemos decir que

`lista.append(x)`	equivale a	`a[-1:] = [a[-1], x]`
`lista.insert(i, x)`	equivale a	`a[i:i+1] = [x, a[i]]`[1]
`lista.pop(i)`	equivale a	`a[i:i+1] = []`

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

Ejercicio 9.11. 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 de `a` al final de cada una de ellas. ¶

Ejercicio 9.12. La mutabilidad hace que debamos ser cuidadosos cuando las listas son argumentos de funciones.

Por ejemplo, definamos

[1] Supuesto que `a[i]` exista.

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

Poniendo

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

vemos que `b` se ha modificado, y ahora es `[2, 1]`. ¶

Ejercicio 9.13. 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 9.12, de modo de *retornar* una copia de `a` a la que se le ha agregado 1, *sin modificar* la lista `a`. ¶

Ejercicio 9.14. En realidad la cosa no es tan sencilla como sugiere el ejercicio 9.13. 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 9.7 y 9.12.

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*). ¶

Dada la mutabilidad de las listas, debemos recordar:

> *Cuando pasamos una lista 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 mutablitis aguda.

Ejercicio 9.15 (problemas con asignaciones múltiples). Las asignaciones múltiples e intercambios pueden ser convenientes (como en los ejercicios 9.3 y 9.4), pero debemos tener cuidado cuando hay listas involucradas porque son mutables.

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. ¶

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.

Ejercicio 9.16 (`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.

Ejercicio 9.17 (`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. ¶

Ejercicio 9.18. La concatenación que vimos en el ejercicio 4.15 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)` ¶

Ejercicio 9.19 (`isinstance` II). Con `isinstance` (ejercicio 4.18) podemos determinar si un objeto es de cierto tipo. Si queremos determinar si es de uno entre varios tipos posibles, ponemos el segundo argumento de `isinstance` como tupla.

a) Si queremos determinar si `x` es un número (entero o real), podríamos poner

```
isinstance(x, (int, float))
```

Probar este esquema cuando `x` es:

i) `1` *ii*) `1.2` *iii*) `'mi mama'` *iv*) `False`

✍ La última respuesta no es satisfactoria si queremos que los valores lógicos no sean considerados como enteros (ver el ejercicio 7.10).

b) Definir una función `essecuencia` que determine si su argumento es o no una secuencia (cadena, tupla, lista o rango). ¶

Ejercicio 9.20 (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 10.8. 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 es propio de Python, y 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. Esta estructura es distinta a la de lista y no está implementada en Python.

  En este curso será suficiente emular arreglos usando listas porque los ejemplos son de tamaño muy reducido.

  El módulo no estándar *numpy* implementa arreglos en Python eficientemente, pero no lo usaremos en el curso.
- 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,[1] pero normalmente se tiene algo un poco al revés:

[1] 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 9.17) 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 12.

Miremos algunos ejemplos sencillos.

Ejercicio 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 *b*) son equivalentes a:

```
long = 0
for i in range(len(a)):
```

```
    long = long + 1  # la acción no depende de i
long
```

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`. ¶

La similitud de las construcciones

```
for x in a:     # para cada elemento
    ...
```

y

```
for i in range(len(a)):  # para cada índice
    ...
```

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`.

Ejercicio 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
```
¶

Ejercicio 10.3. En el ejercicio 9.17 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
for y in a:
    if y == x:
        t = True
        break
t
```
(10.2)

a) Comprobar la validez del esquema probándolo para los valores de `a` y `x` del ejercicio 9.17.

b) 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 9.17. ¶

Ejercicio 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, por ejemplo. ¿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)`? ¶

Ejercicio 10.5 (`index` y `count`). Además de `in`, para saber si determinado elemento está en la sucesión, podemos usar `index` para encontrar la primera posición o `count` para ver cuántas veces aparece.

a) Averiguar qué hacen estos métodos poniendo, por ejemplo,

```
help(list.count)
help(list.index)
```

b) Poniendo `a = 'mi mama me mima'` (sin tildes), hacer los siguientes:

i) Contar las veces que aparece la letra `'a'` en `a` «manualmente», y verificar que `a.count('a')` da el mismo resultado.

ii) Encontrar la primera posición de la letra `a` en `a` «manualmente», y luego usar `a.index('a')`.

iii) La letra `'p'` no está en `a`: ver los resultados de

```
'p' in a
a.index('p')
a.count('p')
```

c) Expresar `index` en términos de la función `posiciones` del ejercicio 10.4, es decir, encontrar una expresión que involucre `posiciones(a, x)` que sea equivalente a `a.index(x)`.

Repetir para `count` (en vez de `index`).

d) Definir una función `sacar1(a, x)` que elimina la primera aparición del objeto `x` en la lista `a` (modificándola) si es que `x` está en `a` usando sólo `if`, `count`, `index` y `pop`. ¶

Ejercicio 10.6 (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) `maxpos([1, 2, 3])` *ii*) `maxpos((1, 2, 3))`
iii) `maxpos(1, 2, 3)` *iv*) `maxpos('mi mama')`
v) `maxpos([1, 'a'])` *vi*) `maxpos([])`

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. ¶

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

$$1 + 2.1 = 3.1, \quad 3.1 + 3.5 = 6.6, \quad 6.6 - 4.7 = 1.9. \tag{10.5}$$

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 = 0           # valor inicial de s
s = s + 1       # s -> 1
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 la ecuación (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)

Ejercicio 10.7 (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 10.6, 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.

Nos gustaría definir un valor conveniente para la suma de sucesiones vacías como `''` o `[]`, y arbitrariamente decidimos que en esos casos pondremos un cartel avisando de la situación y retornaremos la sucesión original.

Tendríamos algo como:

```
if len(a) == 0:     # nada para sumar
    print('*** Atención: Sucesión vacía')
    return a        # 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
```

Ejercicio 10.8. 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)` ¶

Ejercicio 10.9 (factorial). Recordemos que para $n \in \mathbb{N}$, el factorial se define por

$$n! = 1 \times 2 \times \cdots \times n. \tag{10.7}$$

a) Definir una función `factorial(n)` para calcular $n!$ usando la ecuación (10.7), usando el esquema:

```
f = 1
for i in range(1:n+1):
    f = f * i
```
(10.8)

✍ En realidad, tanto en la ecuación (10.7) como en en el esquema (10.8), podríamos empezar multiplicando por 2 en vez de por 1. Se empieza desde 1 como regla mnemotécnica.

☞ *Cuando queremos calcular la* suma *de varios términos, ponemos inicialmente el valor de la suma en* 0.

En cambio, cuando queremos calcular el producto *de varios términos, ponemos el valor inicial del producto en* 1.

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[(2)] expresa que

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}, \tag{10.9}$$

donde

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k!} \tag{10.10}$$

(2) A veces llamado *teorema del binomio de Newton*.

son los *coeficientes binomiales* o *números combinatorios* ($0 \le k \le n$).

✍ Cuando $x = y = 1$, (10.9) se reduce $2^n = \sum_{k=0}^{n} \binom{n}{k}$, 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.

Ejercicio 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.

a) Volcar esta idea en una función para calcular el coeficiente binomial $\binom{n}{k}$ donde los argumentos son n y k ($0 \le k \le n$).

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 $k \le n/2$. ¶

Ejercicio 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 $a = (a_1, a_2, \ldots, a_n)$, la lista de acumuladas es la lista de sumas parciales,

$$s = (a_1, a_1 + a_2, a_1 + a_2 + a_3, \ldots, a_1 + \cdots + a_n),$$

o, poniendo $s = (s_1, s_2, \ldots, s_n)$, más formalmente tenemos

$$s_k = \sum_{i=1}^{k} a_i, \qquad k = 1, \ldots, n. \tag{10.11}$$

✍ 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.

b) Una posibilidad es construir las sumas

```
[sum(a[:k]) for k in range(1, len(a))]
```

pero es terriblemente ineficiente porque volvemos a empezar cada vez.

c) A partir de (10.11), podemos poner

$$s_1 = a_1, \qquad s_k = s_{k-1} + a_k \quad \text{para } k > 1,$$

y de aquí que

$$a_1 = s_1, \qquad a_k = s_k - s_{k-1} \quad \text{para } k > 1. \tag{10.13}$$

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.

Los que 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.

Indicando por ◊ al operador, en todos estos casos usamos un esquema como:

```
s = valor_inicial   # valor inicial del acumulador
for x in secuencia:
    s = s ◊ x
s
```

Por ejemplo:

- Para la suma ponemos inicialmente `s = 0` y en cada paso `s = s + x`.
- Para el producto 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 (ver también 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 10.8 para la suma (de números u otros objetos) y en el ejercicio 10.6 para el máximo,

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.

Ninguno 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.

💣 **Ejercicio 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:

```
b = [1]
for x in b:
    b.append(1)
```

b) 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 para arreglar la función: recorrer la lista desde atrás hacia adelante.

✍ Ver también los ejercicios 10.5.*d*), 10.21 y 14.5. ¶

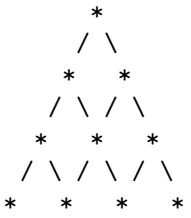


Figura 10.1: «Arbolito de Navidad» con 4 estrellas en la parte inferior.

Ejercicio 10.13. Construir una función `navidad(n)` que imprima un «arbolito de Navidad» como se muestra en la figura 10.1 para $n = 4$. El árbol 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 4.15), 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`. ¶

10.2. Listas por comprensión

Los conjuntos se pueden definir por *inclusión* como en $A = \{1, 2, 3\}$, o por *comprensión* como en $B = \{n^2 : n \in A\}$. En este ejemplo particular, también podemos poner en forma equivalente la definición por inclusión $B = \{1, 4, 9\}$.

De modo similar, las listas también pueden definirse por inclusión, 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.

Ejercicio 10.14.

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 19. ¶

Ejercicio 10.15. 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? ¶

Ejercicio 10.16. 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
```
¶

Ejercicio 10.17. 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
```

a) Probar los tres esquemas anteriores, viendo que dan los mismos resultados.

b) 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.

c) ¿Podrías demostrar la fórmula? ¶

Ejercicio 10.18. Comparar los valores de la función `factorial(n)` (ejercicio 10.9) con los de `math.factorial(n)` para $n = 1, \ldots, 10$:

a) Construyendo dos listas y viendo si son iguales:

```
uno = [factorial(n) for n in range(1, 11)]
dos = [math.factorial(n) for n in range(1, 11)]
uno == dos
```

b) Usando un lazo `for`

```
t = True
for n in range(1, 11):
    if factorial(n) != math.factorial(n):
        t = False
        break
t
```

✍ 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 (pág. 113). ¶

10.3. 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):                                    (10.16)
        lista.append(f(x))
lista
```

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) for x in lista]` tiene la misma longitud de `lista`, `[f(x) for x in lista if p(x)]` puede tener longitud menor o inclusive ningún elemento.

Ejercicio 10.19. Dando definiciones por comprensión con filtros, encontrar:

a) Los números positivos en `[1, -2, 3, -4]`.

b) Las palabras terminadas en «o» en

```
['Ana', 'Luisa', 'y', 'Mateo', 'Adolfo']
```

Ayuda: el índice correspondiente al último elemento de una sucesión es −1. ¶

Ejercicio 10.20. 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 = 'mi mama me mima'` (sin tildes) y `x` es:

a) `'m'` *b*) `'i'` *c*) `'a'` *d*) `'A'` ¶

Ejercicio 10.21. ¿Qué hace la función

```
def haceralgo(a, x):
    return [y for y in a if y != x]  ?
```

✍ En el ejercicio 10.12.*b*) estudiamos una versión usando lazos donde la lista *se modifica*. Ver también el ejercicio 10.5.*d*). ¶

Ejercicio 10.22. 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` y la otra usando filtros.

Si ponemos `indices = posiciones(lista, x)`, ¿cuál será el resultado de `sublista(lista, indices)`? ¶

10.4. Ejercicios adicionales

Ejercicio 10.23. Python tiene el método `reverse` que «da vuelta» una lista modificándola (ver el ejercicio 9.9). Esta acción se puede hacer eficientemente con un lazo `while`, tomando dos índices que comienzan en los extremos y se van juntando:

```
i, j = 0, len(a) - 1
while i < j:
    a[i], a[j] = a[j], a[i]     # intercambiar
    i, j = i + 1, j - 1
```

Definir una función `darvuelta` siguiendo este esquema y probarla con distintas listas. ¶

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*). En realidad, el esquema con `while` de ese ejercicio es ligeramente más eficiente en Python que el correspondiente lazo `for`.
- 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 descriptas (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

Cuatro variaciones sobre un tema

En este capítulo estudiamos algunas aplicaciones sencillas que parecerían no estar relacionadas entre sí, pero que desde las matemáticas pueden verse como variantes del mismo tema.

11.1. Reloj... no marques las horas

> Las horas de la locura las mide el reloj, pero ningún reloj puede medir las horas de la sabiduría.
>
> *Proverbios del Infierno* (1792)
> William Blake (1757–1827)

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:

$$\begin{aligned} 12\text{ hs} = 12 \times 3600\text{ s} &= 43200\text{ s} \\ 34\text{ m} = 34 \times 60\text{ s} &= 2040\text{ s} \\ 56.78\text{ s} &= 56.78\text{ s} \\ \hline \text{total} &= 45296.78\text{ s} \end{aligned} \tag{11.1}$$

Pero parece más sencillo primero pasar horas a minutos y luego minutos a segundos:

$$\begin{gathered} 12\text{ h} = 12 \times 60\text{ m} = 720\text{ m}, \\ 720\text{ m} + 34\text{ m} = 754\text{ m}, \\ 754\text{ m} = 754 \times 60\text{ s} = 45240\text{ s}, \\ 4524\text{ s} + 56.78\text{ s} = 45296.78\text{ s}, \end{gathered}$$

que podríamos poner en un único renglón:

$$(12 \times 60 + 34) \times 60 + 56.78 = 45296.78. \tag{11.2}$$

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):

```
t = a * 365 + d      # días
t = t * 24 + h       # horas
t = t * 60 + m       # minutos
t = t * 60 + s       # segundos
```
(11.3)

Aunque no es estrictamente necesario, suponemos que

$$\begin{gathered} a, d, h \text{ y } m \text{ son enteros, } s \text{ puede ser decimal,} \\ 0 \le a,\ 0 \le d \le 364,\ 0 \le h \le 23,\ 0 \le m \le 59,\ 0 \le s < 60. \end{gathered} \tag{11.4}$$

Si los datos están en la lista `tiempo = [a, d, h, m, s]` y las unidades de las conversiones en la lista `c = [365, 24, 60, 60]`, podríamos poner:

```
n = len(tiempo)     # len(c) es n - 1
t = tiempo[0]
for i in range(1, n):                                    (11.5)
    t = t * c[i - 1] + tiempo[i]
t       # tiempo pasado a segundos
```

Ejercicio 11.1. Tomando como base el esquema (11.5), 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]` ¶

Consideremos ahora el problema inverso: tenemos un tiempo expresado en segundos y queremos pasarlo a años, días, horas, minutos y segundos, bajo las restricciones (11.4).

Recorriendo de abajo hacia arriba el esquema (11.3), si t es el tiempo expresado en segundos y lo dividimos por 60, el cociente es el tiempo expresado en minutos y el resto son los segundos. Quedaría:

```
s, m = t % 60, t // 60    # segundos y minutos
m, h = m % 60, m // 60    # minutos y horas
h, d = h % 24, h // 24    # horas y días
d, a = d % 365, d // 365  # días y años
```

Este esquema tiene el problema de que si `t` no es entero, tampoco lo serán `t % 60` y `t // 60`:

```
>>> 45296.78 % 60, 45296.78 // 60
(56.779999999998836, 754.0)
```

lo que contradice la suposición (11.4).

Como sólo los segundos pueden ser decimales, podríamos separar la parte entera de la decimal, y después seguir trabajando sólo con enteros.

Recordando que el tiempo final será una lista de longitud 5 y la completaremos desde atrás hacia adelante, si la entrada es `t` y `c = [365, 24, 60, 60]`, podemos poner:

```
n = int(t)      # parte entera
f = t - n       # parte fraccionaria
tiempo = [0 for i in range(5)]
for i in range(4, -1, -1):                          (11.6)
    n, tiempo[i] = divmod(n, c[i - 1])
tiempo[-1] = tiempo[-1] + f
tiempo
```

Observar cómo el lazo `for` aquí es el inverso del lazo `for` en el esquema (11.5).

Ejercicio 11.2. Tomando como base el esquema (11.6), definir una función `desegs(tiempo)` que dado un número no negativo de segundos lo convierte a años, días, horas, minutos y segundos respetando las condiciones (11.4).

Comprobar que `desegs` es efectivamente la inversa de `asegs`, tomando como entrada los resultados del ejercicio 11.1. ¶

11.2. Algo de matemática financiera

En esta sección estudiamos una de las tantas herramientas matemáticas usadas en las finanzas, viendo cómo se va actualizando el capital cuando sometido a interés compuesto.

Empecemos con cosas sencillas.

Ejercicio 11.3.

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?

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

inflación indicada.

b) El kilo de pan hoy me cuesta $ 12 y (exactamente) dos años atrás me costaba $ 7.50. ¿Qué estimación podríamos dar sobre la inflación promedio en cada uno de estos dos años? ¶

Supongamos que hacemos un certificado a plazo fijo depositando un capital inicial d_0, con una tasa de interés de r_0 % nominal anual con capitalización mensual.

Al fin del primer mes nuestro capital se ve incrementado por el interés que es proporcional al capital inicial,

$$\textit{interés} = d_0 \times \textit{tasa mensual}.$$

Para calcular la tasa mensual, usamos que r_0 es la tasa nominal anual expresada como porcentaje y que el año se ha dividido en 12 períodos, por lo que en un mes la tasa de interés es $r_0/(100 \times 12)$, y el interés es

$$d_0 \times \frac{r_0}{100 \times 12}.$$

Una vez computados los intereses, nuestro capital al vencimiento del primer período será

$$v_1 = \textit{capital inicial} + \textit{interés} = d_0 \times \left(1 + \frac{r_0}{100 \times 12}\right) = d_0 \times t_0,$$

donde

$$t_0 = 1 + \frac{r_0}{100 \times 12}.$$

Teniendo la opción de renovar el certificado, supongamos que agregamos una cantidad d_1 (que será negativa si extraemos en vez de depositar), es decir, el monto inicial para el próximo certificado será

$$c_1 = v_1 + d_1.$$

Continuando de esta forma, vemos que si

- c_{n-1} es el monto al principio del mes n,

- r_{n-1} es la tasa (nominal anual en porcentaje) correspondiente,
- $t_{n-1} = 1 + \dfrac{r_{n-1}}{12 \times 100}$,
- v_n es el monto al vencer el n-ésimo certificado,
- d_n es lo que agregamos para el certificado en el mes $n+1$,

entonces para $n = 1, 2, \ldots$:

$$c_n = v_n + d_n, \quad v_n = c_{n-1}\, t_{n-1}. \tag{11.7a}$$

Por otro lado, para $n = 0$ tenemos

$$c_0 = d_0. \tag{11.7b}$$

Por ejemplo, al final del tercer período el capital será:

$$\begin{aligned} v_3 &= c_2 \times t_2 = (v_2 + d_2) \times t_2 \\ &= (c_1 \times t_1 + d_2) \times t_2 = ((v_1 + d_1) \times t_1 + d_2) \times t_2 \\ &= ((d_0 t_0 + d_1) \times t_1 + d_2) \times t_2, \end{aligned} \tag{11.8}$$

que es bien parecida a (11.2).

Las ecuaciones (11.7) nos llevan a imitar lo hecho en el esquema (11.5):

```
c = d[0]  # capital inicial = depósito inicial
for i in range(1, n+1):
    t = 1 + r[i - 1] / 1200                    (11.9)
    v = c * t       # monto al vencimiento
    c = v + d[i]    # nuevo capital
```

donde

- `r = [r[0], r[1],...]` son las tasas de interés (nominal, anual, en porciento), y
- `d = [d[0], d[1],...]` son los depósitos mensuales.

Ejercicio 11.4. Tomando como base el esquema (11.9), definir una función `vencimiento(n, d, r)` que determine el monto del certificado al vencimiento del mes n, antes de hacer el depósito (o extracción) correspondiente a ese mes. ¶

Ejercicio 11.5. Al menos para períodos cortos, en muchas ocasiones podemos considerar que tanto la tasa de interés como los depósitos/extracciones mensuales permanecen fijos.

Modificando adecuadamente la función del ejercicio 11.4, definir una función `queda(n, c, r, d)` para determinar el monto al vencimiento después de n períodos, donde c es el capital inicial, r es la tasa de interés (fija), y d es el monto (fijo) de los $n-1$ depósitos que hicimos (después del inicial c).

Aclaración: `c`, `r` y `d` son números (que pueden ser decimales). ¶

Ejercicio 11.6. El banco de Gabi ofrece una tasa de 2 % nominal anual en certificados a dos meses, permitiendo una renovación automática con la misma tasa.

a) Si Gabi hace un certificado por $ 1000 con renovación automática, ¿cuánto dinero puede retirar al cabo de un año (o sea al vencimiento del sexto certificado)?

Aclaración: suponemos que no hay otros gastos como estampillados o impuestos.

b) Si como en el apartado anterior, Gabi retira $ 100 en cada vencimiento, con tasa constante, ¿cuánto dinero queda al final de los 6 períodos (antes de retirar los $ 100)?

c) ¿Y si deposita (en vez de retirar) $ 100 mensuales (y de esta forma aumenta el capital). ¶

11.3. Polinomios

Los polinomios son expresiones de la forma

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n \\ &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0, \end{aligned} \tag{11.10}$$

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 primera tarea será evaluar un polinomio dado.

Ejercicio 11.7 (regla de Horner). Construir una función que dada una lista de coeficientes (reales) $(a_0, a_1, a_2, \ldots, a_n)$ y un número $x \in \mathbb{R}$, evalúe el polinomio $P(x)$ en la ecuación (11.10).

Hacerlo de tres formas:

a) Calculando la suma de términos como se indica en la ecuación (11.10), calculando x^k para cada k.

✍ Corresponde al esquema (11.1).

b) Como el anterior, pero las potencias en la forma $x^{k+1} = x^k \times x$, guardando x^k en cada paso.

c) Usando la *regla de Horner*,

$$((\cdots((a_n x + a_{n-1})x + a_{n-2}) + \cdots)x + a_1)x + a_0. \qquad (11.11)$$

✍ Es la versión correspondiente a la ecuación (11.2) para el caso del tiempo, y a (11.8) para el caso del certificado, suponiendo que los depósitos son constantes.

✍ Observar que primero se multiplica por a_n, luego por a_{n-1}, etc., es decir, recorremos la lista al revés de lo que hacemos en las ecuaciones mencionadas. ¶

En las dos primeras versiones del ejercicio 11.7 se hacen n sumas, n productos y se calculan $n-1$ potencias, que en la primera versión representan otros $1+2+\cdots+(n-1) = n(n-1)/2$ 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.

Es por eso que para pasar de horas, minutos y segundos a segundos, intuitivamente elegimos usar la ecuación (11.2) antes que el procedimiento del esquema (11.1).

❤ *En la mayoría de los casos es preferible usar la regla de Horner para evaluar polinomios, y sólo en algunos pocos casos particulares debe usarse explícitamente potenciación.*

11.4. Escritura en base entera

Ejercicio 11.8. Dado una base b entera, $b > 1$, y un entero $n \in \mathbb{N}$, nos gustaría encontrar coeficientes $a_0, a_1, \ldots, a_k$ de modo que

$$n = \sum_{i=0}^{k} a_i b^i, \quad \text{donde } a_i \in \mathbb{Z} \text{ y } 0 \le a_i < b. \tag{11.12}$$

Siguiendo las ideas de la regla de Horner (ver la ecuación (11.11)), ponemos

$$\begin{aligned} n &= \sum_{i=0}^{k} a_i b^i \\ &= a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b + a_0 \\ &= \left(a_k b^{k-1} + a_{k-1} b^{k-2} + \cdots + a_1\right) \times b + a_0 \\ &= \left(\left(a_k b^{k-1} + a_{k-1} b^{k-2} + \cdots + a_1\right) \times b + a_0 \right. \\ &= \left(\left(\cdots\left(\left(a_k b + a_{n-1}\right) \times b + a_{k-2}\right) + \cdots\right) \times b + a_1\right) \times b + a_0. \end{aligned}$$

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.

✍ Comparar con el esquema (11.6). También usamos este proceso en el ejercicio 8.8.

a) Definir una función que dados n y b encuentre los coeficientes $(a_0, a_1, \ldots, a_k)$ de la expresión (11.12), 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) 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 (11.12), ¿cómo se relacionan k y $\log_b n$ (si $a_k \neq 0$)?

✍ Recordar también el ejercicio 3.16. ¶

11.5. Ejercicios adicionales

Ejercicio 11.9 (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

$$x^n = x^{a_0+2a_1+2^2a_2+\cdots+2^{k-1}a_{k-1}+2^k a_k} =$$
$$= x^{a_0} \cdot (x^2)^{a_1} \cdot (x^4)^{a_2} \cdots (x^{2^{k-1}})^{a_{k-1}} \cdot (x^{2^k})^{a_k}.$$

Observando que las potencias de x son x, x^2, x^4,... , 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
```

a) Definir una función para el cálculo de potencias «puras» con estas ideas.

b) Hacer una función para calcular x^n usando un lazo `for`.

c) Comparando las funciones definidas en los apartados anteriores: ¿cuántas operaciones aritméticas (productos y divisiones) se realizan en cada caso?

d) Calcular 456^{789} de tres formas: usando `x**n`, y con la función definidas en los dos primeros apartados.

❦ *Debido a su alta eficiencia, una técnica similar se usa para encriptar mensajes usando números primos de varias decenas de cifras.* ¶

Capítulo 12

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.

Como en otros casos, las sentencias que vemos son particulares a Python pero tienen su correlato en otros lenguajes.

12.1. Formatos

Empezamos estudiando cómo imprimir secuencias en forma prolija, especialmente cuando se trata de encolumnar la información en tablas. Por supuesto que la presentación puede ser mejorada enormemente con otro tipo de aplicaciones como los procesadores de texto o las de diseño de página. Como además no hay mucha matemática involucrada, lo que vemos en esta sección no forma parte esencial del curso, si bien será útil muchas veces.

Ejercicio 12.1 (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 (en un módulo que se ejecutará):

```
for x in a[:-1]:
    print(x, end='')
print(a[-1])
```

¶

Ejercicio 12.2 (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*[1] y en el curso nos restringiremos al método `format` usando llaves (« `{` » y « `}` »).

Veamos cómo funciona con un ejemplo:

[1] Sí, es una palabra permitida por la RAE.

```
for n in range(1, 11):
    print('{0:2d} {1:4d}'.format(n, n**2))
```

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) Observar también que dejamos un espacio entre los dos juegos de llaves: modificarlos a ningún espacio o varios para ver el comportamiento.

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

```
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) Si queremos hacer una tabla del seno donde la primera columna sean los ángulos entre 0° y 45° en incrementos de 5 podríamos poner:

```
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. ¶

✍ 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*).

Ejercicio 12.3. Las instrucciones siguientes construyen la tabla de verdad para la conjunción ∧ («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)))
```

a) Modificarlas para obtener la tabla de verdad para la disyunción $\vee$ (el «o» lógico).

b) Modificarlas para que aparezcan las cuatro columnas p, q, $p \wedge q$, $p \vee q$. ¶

Ejercicio 12.4. Repetir el ejercicio 10.13 usando formatos y `print(..., end = '')` en vez de concatenación o multiplicación de cadenas de caracteres. ¶

12.2. 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.

Para nosotros será conveniente que el archivo sea *de texto*, de modo de poder leerlo con un *editor de textos* como IDLE.

- ✍ En general, los archivos de texto se guardan con extensión *.txt*, pero podrían tener cualquier otra como *.dat* o *.sal*, o ninguna (en MS-Windows la cosa puede ser distinta...).
- ✍ 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 de texto para guardar la información y *leer* esos archivos para recuperarla. Empecemos por la escritura.

Al escribir el programa, tenemos que relacionar el nombre del archivo en el disco de la computadora, llamémoslo *externo*, con un nombre que le daremos en el programa, llamémoslo *interno*.

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'`, `'Mateo Adolfo'` o cualquier otro.

Ejercicio 12.5 (guardando en archivo). 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 12.2.*e*).

Observamos las siguientes novedades:

- La variable `archivo` es el nombre interno (dentro del módulo) que se relaciona con el nombre externo *tablaseno.txt* mediante la instrucción `open` (*abrir* en inglés).
- 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.
- El tercer argumento de `open` es la codificación (*encoding*) necesaria para leer correctamente el archivo cuando hayamos terminado. En nuestro caso ponemos la codificación utf-8 con `encoding='utf-8'`.

  ✍ Es posible que la codificación por defecto sea utf-8 (y no sea necesario explicitarla) pero esto depende de la configuración del sistema operativo: más vale prevenir...
- 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.

Ayuda: recordar `input` (ejercicio 6.2). ¶

Ejercicio 12.6 (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')
```

Similar al caso de escritura, esta instrucción indica a Python que en lo que sigue estamos asociando la variable `archivo`, que es interna a Python, con el nombre *santosvega.txt*, que es el nombre del archivo en el disco de la computadora.

El segundo argumento en `open` es ahora `'r'` (por read, *leer*), indicando que vamos a leer el archivo.

✍ No es necesario poner `'r'` pues se supone por defecto. Para no complicarla, lo incluimos.

Como en el caso de escritura, el tercer argumento de `open` es la codificación (`encoding`).

b) Averiguar el valor y el tipo de `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).

✍ 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) 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)
```

✍ Es conveniente cerrar (`close`) antes de volver a abrir (`open`) el archivo.

✍ El método `readline` también nos permite leer un renglón: `archivo.readline()`. Nosotros no lo veremos.

f) El resultado del apartado anterior es que se deja otro renglón en blanco después de cada renglón. Podemos usar la técnica del ejercicio 12.1 poniendo

```
archivo = open('santosvega.txt', 'r',
               encoding='utf-8')
for renglon in archivo:
    print(renglon, end='')
```

Comprobar el resultado de estas instrucciones.

g) A pesar del lazo `for` en el apartado *e*), `archivo` no es una sucesión de Python. Probarlo poniendo

```
archivo = open('santosvega.txt', 'r',
               encoding='utf-8')
archivo[0]
len(archivo)
```

✍ No es una sucesión pero es un *iterable*, como lo son las sucesiones. Veremos algo más sobre el tema en el ejercicio 12.10.

h) 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*).

i) Cuando el archivo no se va a usar más, es conveniente liberar los recursos del sistema, *cerrando* el archivo, como hicimos en el ejercicio 12.5.

Poner

```
archivo = open('santosvega.txt', 'r',
               encoding='utf-8')
archivo.close()
```

```
archivo.read()
```

viendo que la última instrucción da error.

j) En el módulo *dearchivoaconsola* hacemos una síntesis de lo que vimos: preguntamos por el nombre del archivo de texto a imprimir en la consola, abrimos el archivo, lo imprimimos y finalmente lo cerramos.

Probarlo ingresando los nombres *santosvega.txt* y *dearchivoaconsola.py*. ¶

Ejercicio 12.7. Tomando partes de los módulos *tablaseno* y *dearchivoaconsola*, 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). ¶

Abriendo el archivo *tablaseno.txt* 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.

Ejercicio 12.8 (`split`).

a) Ver qué hace el método `split` poniendo `help(str.split)`.

b) Ver el resultado de:

```
a = 'mi␣␣␣mamá␣me...␣␣␣␣imima!'
a
a.split()
```

donde en `a` ponemos arbitrariamente espacios entremedio.

c) Como el anterior, poniendo

```
a = 'mi\nmama␣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()`? ¶

Ejercicio 12.9. Usando `split`, imprimir —una por renglón— las palabras con 5 o más letras en el archivo *santosvega.txt*. ¶

Ejercicio 12.10 (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 12.5.

b) Usando `for` para construir una lista por comprensión (en forma similar al ejercicio 12.6.*e*)), ver el resultado de las siguientes instrucciones:

```
a = [x for x in archivo]
a
a[0]
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.

✍ Será conveniente primero cerrar el archivo y luego volver a abrirlo. ¶

12.3. Ejercicios adicionales

Ejercicio 12.11. La posibilidad de leer y escribir archivos con Python nos brinda una interfaz —bastante elemental— para comunicarnos con otro software.

Por ejemplo, MS-Excel y planillas de cálculo similares también tienen la posibilidad de leer y escribir archivos de texto, y en este ejercicio trataremos de intercambiar datos entre ellos y Python.

a) En una planilla de cálculo crear una tabla del seno: la columna de la izquierda representando grados entre 0° y 90° en incrementos de 10° y en la columna de la derecha el seno del ángulo correspondiente.

Guardar la planilla como archivo de texto «delimitado por tabulaciones».

b) De modo similar, construir una planilla de tres columnas y cuatro filas, donde la primer columna represente apellidos, la segunda nombres y la tercera edades. Guardar también esta planilla como archivo de texto delimitado por tabulaciones.

c) Modificando adecuadamente el módulo *dearchivoaconsola*, construir una función en Python para leer archivos de texto e imprimirlos, y luego probarla con los archivos construidos en los apartados anteriores.

✍ Si los nombres o apellidos tienen tildes, como en «María Sánchez», pueden haber problemas en la lectura.

En estos casos tenemos las opciones de guardar el archivo de texto desde MS-Excel o similar con la codificación utf-8, o bien abrir el archivo con Python usando la codificación apropiada, por ejemplo, poniendo `encoding = 'latin_1'`.

d) Modificar la función del apartado anterior para construir una lista de Python: las filas serán las filas de la planilla y las entradas

las columnas correspondientes, donde las tabulaciones « `'\t'` » se reemplazan por separaciones de lista « `,` » con `split`.

Comprobar el funcionamiento usando los archivos construidos en *a)* y *b)*.

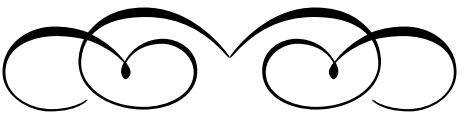
e) Definir una nueva función, variante de la del apartado anterior, para pasar de cadena de caracteres a número cada entrada de la lista (matriz) resultante. Verificar el resultado leyendo el archivo construido en *a)*.

f) Recíprocamente, modificando el módulo *tablaseno*, construir un archivo de texto con la tabla del coseno para ángulos entre 0° y 90° en incrementos de 10° y luego abrirlo con MS-Excel (o similar):

- separando grados y los senos con tabulaciones (`'\t'`),
- cada fila con nuevo renglón `'\n'`.

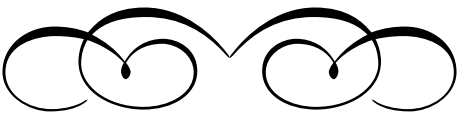
Ayuda: usar `'{0}\t{1}\n'.format(...)`.

✍ Valen las observaciones hechas en *c)*. ¶



Parte II

Popurrí



Capítulo 13

Números aleatorios

> Cualquiera que considere métodos aritméticos para producir dígitos aleatorios está, por supuesto, en estado de pecado.
>
> John von Neumann (1903–1957)

Muchas veces se piensa que en matemática «las respuestas son siempre exactas», olvidando que las probabilidades forman parte de ella y que son muchas las aplicaciones de esta rama.

Una de estas aplicaciones es la simulación, técnica muy usada por físicos, ingenieros y economistas cuando es difícil llegar a una fórmula que describa el sistema o proceso. Así, simulación es usada para cosas tan diversas como el estudio de las colisiones de partículas en física nuclear y el estudio de cuántos cajeros poner en el supermercado para que el tiempo de espera de los clientes en las colas no sea excesivo. La simulación mediante el uso de la computadora es tan difundida que hay lenguajes de programación (en vez de Python o C) especialmente destinados a este propósito.

Cuando en la simulación interviene el azar o la probabilidad, se usan números generados por la computadora que reciben el nombre de *aleatorios*, o más correctamente, *seudo-aleatorios*, porque hay un

algoritmo determinístico que los construye.

En este capítulo miramos propiedades de estos números y algunas de las facilidades que ofrece Python para su uso.

13.1. Funciones de números aleatorios en Python

Como en el caso de funciones matemáticas como seno o logaritmo para las que usamos el módulo *math*, para trabajar con números aleatorios en Python usamos el módulo *random* (*aleatorio* en inglés).

Ejercicio 13.1. Veamos ejemplos de las funciones en *random* que nos interesan.

✍ Hay muchas más que no veremos: usar `help(random)`.

Por supuesto, arrancamos con

```
import random
```

a) `random.random` es la función básica en *random*. Genera un decimal (`float`) aleatoria y uniformemente en el intervalo $[0,1)$.

```
random.random()
random.random()
```

b) En general, los números aleatorios se obtienen a partir de un valor inicial o *semilla* (*seed* en inglés). Un método eficaz para obtener distintos sucesiones de números aleatorios es cambiar la semilla de acuerdo a la hora que indica el reloj de la computadora, lo que hace Python automáticamente.

Para obtener los mismos números siempre, podemos usar la misma semilla con `random.seed`.

```
random.seed(0)
random.random()
random.random()
random.seed(0)
```

```
random.random()
```

c) 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)]
```

✍ Python tiene otra función similar, `random.randrange`, que no veremos en el curso para no confundir.

d) `random.choice` nos permite elegir un elemento de una sucesión no vacía.

```
random.choice(range(5)) # = random.randint(0, 4)
random.choice('mi mama me mima')
random.choice([2, 3, 5, 7, 11, 13, 17])
```

e) Si queremos elegir varios elementos aleatoriamente, lo que se llama una *muestra* aleatoria, podemos usar `random.sample`.

```
a = [2, 3, 5, 7, 11, 13, 17]
random.sample(a, 4)
a   # no cambia
random.sample('mi mama', 3)
random.sample((1, 5, 8, -1), 2)
random.sample(range(10), 4)
random.sample(range(10), 20)    # da error
random.sample([1 for i in range(20)], 5)
```

f) `random.shuffle` genera una permutación aleatoria «in situ» de una lista.

```
a = [2, 3, 5, 7, 11, 13, 17]
random.shuffle(a)
a
random.shuffle(a)
a
random.shuffle('mi mama')   # da error
```

```
random.shuffle((1, 5, 8))   # da error
```

g) ¿Cómo podría obtenerse una permutación aleatoria de los caracteres en `'mi mama me mima'`?

Sugerencia: una posibilidad es combinar `random.shuffle`, la función `sublista` del ejercicio 10.22 y la función `sumar` del ejercicio 10.8, pero hay muchas otras posibilidades. ¶

Ejercicio 13.2. Es posible usar sólo unas pocas funciones del módulo *random* y emular con ellas el resto. Veamos algunos ejemplos.

a) Construir una función `mirandint(a, b)` (donde `a` y `b` son enteros) para emular el comportamiento de `random.randint` usando sólo la función `random.random`.

Sugerencia: cuando $a = 0$ y $b = n - 1$, `mirandint(0, n-1)` podría definirse como `int(random.random() * n)`, y en general podríamos poner `a + int(random.random() * n)` donde $n = b - a + 1$.

b) Usando `mirandint` o `random.randint`, construir una función `michoice(s)` (donde `s` es una sucesión) para emular el comportamiento de `random.choice`.

✍ Es bastante más complicado emular el comportamiento de las funciones `random.shuffle` y `random.sample`. ¶

13.2. ¿Qué son los números aleatorios?

Aunque no es nuestra intención hacer una descripción rigurosa de qué son o cómo se obtienen (lo que nos llevaría un curso o dos), miremos un poco más las propiedades de los números aleatorios.

De alguna forma, no tiene sentido hablar de un número aleatorio: ¿es 2 un número aleatorio?

Más bien, uno piensa en una sucesión de números con ciertas propiedades, fundamentalmente la de *independencia*, o sea que el valor del n-ésimo número no depende de los $n - 1$ valores anteriores (ni de

los posteriores), y la de *distribución*, es decir, cómo se van repartiendo o distribuyendo los valores.

Para ilustrar estas propiedades, fabriquemos una lista de $n = 200$ (o 100 o 1000) números aleatorios:

```
import random
n = 200     # o 100 o 1000
lista = [random.random() for i in range(n)]
```

Dado que no nos interesa mirar uno por uno a los números sino tener una idea global, vamos a mirar a la lista de números como puntos en el plano, poniendo como primera coordenada al índice y como segunda al número original.

Como vamos a hacer varios gráficos, definimos una función:

```
def pts2d(lista):
    """Pasar números a puntos del plano."""
    return [(i, lista[i]) for i in range(len(lista))]
```

Hacemos el gráfico usando el módulo *graficar*, obteniendo algo similar a la figura 13.1:

```
import graficar
graficar.puntos(pts2d(lista))
graficar.titulo = str(n) + ' números aleatorios'
graficar.mostrar()
```

Como vemos en el gráfico, los valores están completamente desparramados, no habiendo un patrón claro de que haya alguna relación entre ellos.

Para obtener algún orden, vamos a clasificar los puntos usando `sort` (que modifica la lista, de modo que trabajamos sobre una copia).

```
clasif = lista[:]  # copiar lista para no cambiarla
clasif.sort()      # ordenar la nueva lista
graficar.reinicializar()   # borrar objetos gráficos
graficar.puntos(pts2d(clasif))
graficar.titulo = 'Los números clasificados'
graficar.mostrar()
```

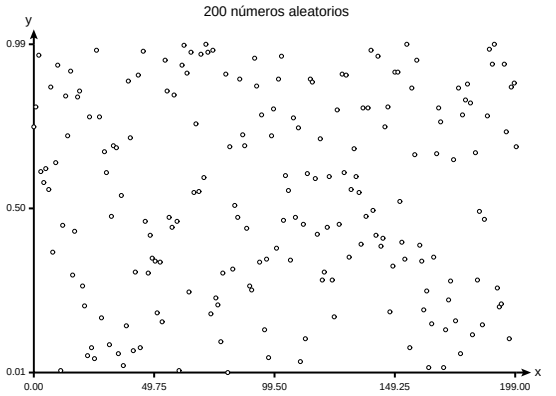


Figura 13.1: 200 números aleatorios entre 0 y 1 representados en el plano.

✍ En vez de copiar la lista en `clasif` y luego ordenarla, podríamos haber puesto directamente `clasif = sorted(lista)`, como veremos en el ejercicio 14.1.

✍ No es demasiado importante usar la misma lista en todos los pasos que vamos haciendo, pero por coherencia la preservamos, lo que podemos hacer de dos formas: o bien generando siempre la misma lista usando una única semilla en `random.seed`, o bien usando `graficar.reinicializar` para no reinicializar IDLE (como hacemos acá).

Obtenemos un gráfico como el de la figura 13.2, donde vemos que los puntos están aproximadamente sobre una recta, representando la *distribución uniforme* de ellos.

Decimos que los números están uniformemente distribuidos pues la probabilidad de que uno de los números esté en un subintervalo no depende de la ubicación del subintervalo, sino sólo de su longitud. Es decir, podemos pensar que intervalos de igual longitud reciben aproximadamente la misma cantidad de puntos. En otras palabras, como los números están entre 0 y 1, en un intervalo $[a, b] \subset [0, 1]$

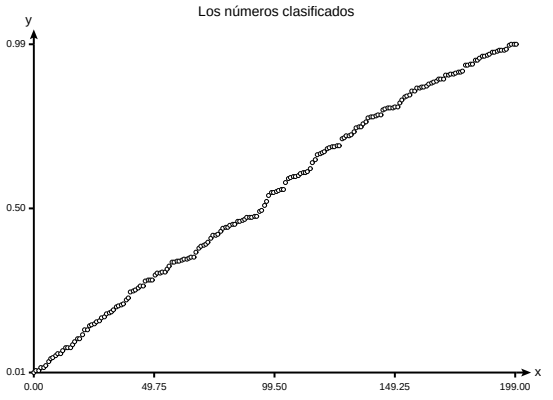


Figura 13.2: Los números clasificados.

habrá aproximadamente $n\,(b-a)$ puntos:

```
a = [x for x in lista if 0.1 < x < 0.2]
len(a), 0.1 * n
a = [x for x in lista if 0.55 < x < 0.65]
len(a), 0.1 * n
```

El que la lista ordenada tuviera como gráfico prácticamente una recta es un efecto de la «uniformidad» y no de ser «números aleatorios», pues para obtener una recta con n puntos podríamos haber tomado directamente la lista `[x/n for x in range(n)]` $(= 0, 1/n, 2/n, \ldots)$ que no parece muy aleatoria que digamos, pero sí que su gráfico está sobre una recta.

En cambio, el hecho de ser aleatoria y los números obtenidos en forma *independiente*, se refleja en que si tomamos una sublista de, digamos, $n/2$ números, en general ésta tendrá el mismo comportamiento que la original.

Ejecutar los siguientes grupos:

- ```
  n2 = n // 2
  lista2 = lista[:n2]
  ```
  ```

```
graficar.reinicializar()
graficar.puntos(pts2d(lista2))
graficar.titulo = 'La mitad de los números'
graficar.mostrar()
```

- ```
  a = [x for x in lista2 if 0.1 < x < 0.2]
  len(a), 0.1 * n2
  a = [x for x in lista2 if 0.85 < x < 0.95]
  len(a), 0.1 * n2
  ```

- ```
  lista2.sort()  # no importa que la lista cambie
  graficar.reinicializar()
  graficar.puntos(pts2d(lista2))
  graficar.titulo = 'Mitad de los números ordenados'
  graficar.mostrar()
  ```

Ejercicio 13.3. Repetir los pasos anteriores tomando:

- sólo los últimos $\lfloor n/2 \rfloor$ números,
- los que estén en posición par,
- eligiendo $\lfloor n/2 \rfloor$ puntos con `random.sample`,

comprobando que no hay diferencias cualitativas. ¶

En cambio, si tomamos los cuadrados, el comportamiento es diferente: los números se agrupan más cerca de 0, y la distribución deja de ser uniforme.

```
graficar.reinicializar()
lista3 = [x**2 for x in lista]
graficar.puntos(pts2d(lista3))
graficar.titulo = 'Los números elevados al cuadrado'
graficar.mostrar()
```

Por intervalos, vemos que en intervalos cercanos a 0 hay más números que cuando tomamos el intervalo cerca de 1:

```
a = [x for x in lista3 if 0.1 < x < 0.2]
len(a), 0.1 * n
```

```
a = [x for x in lista3 if 0.85 < x < 0.95]
len(a), 0.1 * n
```

Clasificando, la curva se parecerá a una parábola:

```
graficar.reinicializar()
lista3.sort()
graficar.puntos(pts2d(lista3))
graficar.titulo = 'Los números al cuadrado ordenados'
graficar.mostrar()
```

13.3. Aplicaciones

Ejercicio 13.4. La función `dado1` (en el módulo *dados*) simula tirar un dado mediante números aleatorios, retornando un número entero entre 1 y 6 (inclusivos).

a) Hacer varias «tiradas», por ejemplo construyendo una lista.

b) Modificar la función para simular tirar una moneda con resultados «cara» o «ceca». ¶

Ejercicio 13.5. La función `dado2` (en el módulo *dados*) hace una simulación para encontrar la cantidad de veces que se necesita tirar un dado hasta que aparezca un número prefijado.

✍ Observar que si el argumento es un número menor que 1 o mayor que 6, el lazo no termina nunca.

a) Ejecutar la función varias veces, para tener una idea de cuánto tarda en aparecer un número.

b) Modificar el lazo de modo de no usar `break` en el lazo poniendo `veces = 1` inicialmente.

Sugerencia: cambiar también la condición en `while`.

c) Corriendo la función 1000 veces, calcular el promedio de los tiros que tardó en aparecer el número predeterminado.

✍ Recordar que el promedio generalmente es un número decimal.

✍ Se puede demostrar que el promedio debe ser aproximadamente 6.

d) Modificar la función a fin de simular que se tiran simultáneamente *dos* dados, y contar el número de tiros necesarios hasta obtener un resultado entrado como argumento (entre 2 y 12). ¶

Ejercicio 13.6. Definir una función que diga cuántas veces debió tirarse un dado hasta que aparezca el seis *k* veces *consecutivas*, donde *k* es argumento.

Sugerencia: poner un contador `c` inicialmente en 0, y dentro de un lazo el contador se incrementa en 1 si salió 6 y si no se vuelve a 0. ¶

En los ejercicios anteriores surge la duda de si el programa terminará alguna vez, dado que existe la posibilidad de que *nunca* salga el número prefijado, o que *nunca* salga *k* veces consecutivas un mismo número. Suponiendo que el generador de números aleatorios es correcto, se puede demostrar matemáticamente que la probabilidad de que esto suceda es 0.

Trabajando con tamaños chicos no hay inconvenientes, pero puede suceder que haya problemas en la práctica, ya que ningún generador es perfecto (siendo determinístico). Por ejemplo, si en vez de dados consideramos permutaciones de *n*, y *n* es más o menos grande (alrededor de 2100), es posible que nunca consigamos una permutación dada con el generador de Python.

Ejercicio 13.7. Mari quiere jugar al Quini y como está haciendo el curso de programación, se le ocurrió usar la compu para encontrar los números, eligiendo aleatoriamente 6 números distintos entre 0 y 45 (inclusivos). El profe le dijo que podría usar `random.sample`, `random.shuffle`, o sólo `random.randint` y `pop`. ¿Podrías ayudarla a usar estas tres variantes? ¶

Ejercicio 13.8. Recordando lo hecho en la sección 13.2:

a) Desarrollar una función para hacer una lista de *r* números enteros, elegidos «aleatoria y uniformemente» entre 1 y *s*, donde $r, s \in \mathbb{N}$ son argumentos.

b) Modificar la función de modo que al terminar imprima la cantidad de veces que se repite cada elemento.

✍ Debido a la distribución uniforme, las cantidades de las apariciones deberían ser muy similares, aproximadamente r/s cada uno, cuando $r \gg s$. ¶

Ejercicio 13.9.

a) Definir una función que elija aleatoriamente el número 1 aproximadamente el 45 % de las veces, el número 2 el 35 % de las veces, el 3 el 15 % de las veces y el 4 el 5 % de las veces.

Sugerencia: considerar las sumas acumuladas .45, .45 + .35,... (recordar el ejercicio 10.11).

b) Generalizar al caso en que en vez de las frecuencias .45, .35, .15 y .5, se dé una lista $(f_1, \ldots, f_n)$ de números no negativos tales que $\sum_{i=1}^{n} f_i = 1$. Probar para distintos valores y verificar que las frecuencias son similares a las deseadas. ¶

Ejercicio 13.10 (dos con el mismo cumpleaños). Mucha gente se sorprende cuando en un grupo de personas hay dos con el mismo día de cumpleaños: la probabilidad de que esto suceda es bastante más alta de lo que se cree normalmente.

a) ¿Cuántas personas te parece que debería haber para que haya dos con el mismo cumpleaños (en general, en promedio)?

Supongamos que en una sala hay n ($n \in \mathbb{N}$) personas y supongamos, para simplificar, que no hay años bisiestos (no existe el 29 de febrero), de modo que podemos numerar los posibles días de cumpleaños $1, 2, \ldots, 365$.

b) ¿Para qué valores de n se garantiza que haya al menos dos personas que cumplen años el mismo día?

Sugerencia: recordar el principio de Dirichlet.

✍ El principio de Dirichlet (o del casillero o del palomar) dice que si hay $n + 1$ objetos repartidos en n casillas, hay al menos una casilla con 2 o más objetos.

c) Si la sala es un cine al cual van entrando de a una las personas, ¿cuántas personas, en promedio, entrarán hasta que dos de ellas tengan el mismo día de cumpleaños? Responder esta pregunta escribiendo una función que genere aleatoriamente días de cumpleaños (números entre 1 y 365) hasta que haya dos que coincidan, retornando la cantidad de «personas» necesarias. Hacer varias corridas para tener una idea más acabada.

✍ Además de suponer que no hay años bisiestos, para la simulación suponemos también que existe la misma probabilidad de nacer en cualquier día, lo que no es estrictamente cierto.

d) Si en tu curso hay 30 compañeros, ¿apostarías que hay dos que cumplen años el mismo día?

✍ El valor teórico para el apartado *c*) es aproximadamente 24.61658.

❦ *Aunque parece trivial, el problema se relaciona con temas tan diversos como métodos para estimar el total de una población o con las* funciones de hash *que usa Python para el manejo eficiente de listas (ver comentarios al final del capítulo 14).* ❡

Ejercicio 13.11. Un problema que causó gran revuelo hacia 1990 en Estados Unidos es el siguiente:

- En un show televisivo el locutor anuncia al participante que detrás de una de las tres puertas que ve, hay un auto o km, no habiendo nada detrás de las otras dos (el auto se coloca detrás de una de las puertas aleatoriamente, con la misma probabilidad para cada puerta).
- El locutor le pide al participante que elija una de las puertas.
- Sin abrir la puerta elegida por el participante, el locutor (quien conoce dónde está el auto) abre otra puerta mostrándole que no hay nada detrás de ésta, y le da la opción al participante de cambiar su elección, pudiendo optar entre mantener su primera elección o elegir la tercera puerta.

¿Debe el participante cambiar su elección?

El revuelo surgió porque mucha gente opinaba que era lo mismo cambiar la elección que no cambiarla, mientras que otros pensaban que era mejor cambiar.

a) ¿Qué te parece (intuitivamente)?

b) Para tener una idea de la solución al problema, definir una función para simular la situación bajo las siguientes suposiciones:

   i) las puertas están numeradas 1, 2 y 3;

   ii) la compu pone aleatoriamente el auto detrás de alguna de las puertas (con igual probabilidad);

   iii) nosotros elegimos siempre la puerta número 1;

   iv) el «locutor» elige una puerta detrás de la cual no está el auto ni es la que elegimos. Si tiene dos posibilidades, elige una aleatoriamente.

   La función debe imprimir la puerta detrás de la cual está el auto, la puerta elegida por el locutor, y retornar `True` si ganamos el auto y `False` en otro caso.

c) Repetir el apartado anterior pero donde siempre cambiamos nuestra elección (en vez de mantenerla).

d) Modificando adecuadamente las funciones de los apartados *b*) y *c*) (por ejemplo, comentando la impresión), hacer varias corridas (1 000 o 10 000) con estas funciones obteniendo el promedio de las veces que ganamos.

   En base a los resultados obtenidos, ¿qué estrategia seguirías, cambiarías o no tu elección? ¶

13.4. Métodos de Monte Carlo

Existen muchos métodos, llamados genéricamente *de Monte Carlo*, para aproximar cantidades determinísticas mediante probabilidades, y en esta sección vemos algunos ejemplos de esta técnica.

❦ *Uno de las primeras aplicaciones fue la* aguja de Buffon *(ejercicio 13.14), pero el método comenzó a aplicarse sistemáticamente cuando S. Ulam (1909–1984) y J. von Neumann (quien eligió el nombre de* Monte-Carlo *en honor al casino famoso) lo usaron hacia 1946 en estudios de física atómica.*

Ejercicio 13.12. Definir una función para simular una máquina que emite números al azar (uniformemente distribuidos) en el intervalo $[0, 1)$ uno tras otro hasta que su suma excede 1. Comprobar que al usar la máquina muchas veces, la cantidad promedio de números emitidos es aproximadamente $e = 2.71828\ldots$ ❡

Ejercicio 13.13. Definir una función para aproximar π tomando n pares de números aleatorios (a, b), con a y b entre -1 y 1, contar cuántos de ellos están dentro del círculo unidad (es decir, $a^2 + b^2 < 1$). El cociente entre este número y n (ingresado como argumento), es aproximadamente el cociente entre las áreas del círculo de radio 1 y el cuadrado de lado 2. ❡

13.5. Ejercicios adicionales

Ejercicio 13.14 (aguja de Buffon). Una aguja fina (teóricamente un segmento) de longitud $\ell \leq 1$, es arrojada sobre un tablero dividido por líneas paralelas donde la distancia entre dos consecutivas es 1. Mostrar, mediante una función adecuada, que el promedio de veces que la aguja cruzará una de las líneas es aproximadamente $2\ell/\pi$.

❦ *Georges Louis Leclerc, Conde de Buffon (1707–1788), planteó este problema en 1733, dando origen a la teoría de* probabilidad geométrica. ❡

13.6. Comentarios

- Hay distintos algoritmos para generar números aleatorios, y los resultados pueden ser muy distintos según el algoritmo, a dife-

rencia de lo que sucede con el seno o el logaritmo (que pueden calcularse con diferentes algoritmos pero los resultados son similares).

El libro de Knuth (1997b, vol. 2) es una excelente referencia para quienes quieran aprender sobre qué debe pedirse a un generador de números aleatorios, y distintos tipos de algoritmos para construirlos.

- Muchos de los sistemas operativos y las CPU tienen sus propios generadores, no necesariamente iguales al que usa Python.

  En general, los algoritmos para construir números aleatorios trabajan con números enteros. Por cuestiones prácticas se pone un límite sobre el tamaño, forzando a los números «aleatorios» a repetirse cíclicamente.

  El algoritmo que usa Python tiene un ciclo de $2^{19937} - 1$, que tiene unas 13820 cifras en base 10, ¡ampliamente suficiente para nosotros!

- Python tiene varias funciones para números aleatorios, inclusive usando distintas distribuciones probabilísticas, pero nosotros vamos a usar siempre la distribución uniforme. El *manual de la biblioteca* tiene información completa sobre el módulo *random*.

- Los ejemplos del método de Monte Carlo que dimos (ejercicios 13.12, 13.13 y 13.14) son más que nada de interés teórico e introductorio al tema, pues las aproximaciones respectivas son lentas.

Capítulo 14

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).[1] De cualquier forma, vamos a ver el *método de conteo* en el ejercicio 14.4 que extiende técnicas ya vistas, y es muy sencillo y rápido (usando del orden de n pasos) en ciertas circunstancias.

Análogamente, con `in`, `index` y `count` (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 mejo-

[1] El interesado puede consultar el excelente libro de Wirth (1987).

rar la eficiencia si la lista está ordenada usando búsqueda binaria, lo que hacemos en la sección 14.3.

14.1. Clasificación

Ejercicio 14.1. Para clasificar una lista (modificándola) podemos usar el *método* `sort` en Python. 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` que da una lista.

Verificar los resultados de los siguientes en la terminal de IDLE, donde usamos la función `sumar` definida en el ejercicio 10.8:

a)
```
import random
a = list(range(10))
random.shuffle(a)
a
b = sorted(a)
b
a                       # no se modificó
a.sort(reverse=True)  # clasificar al revés
a
```

b)
```
a = 'mi mamá me mima'   # con 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 « á » (con tilde) no aparezca en el lugar correcto. Para que lo haga hay que cambiar algunas cosas que nos desviarían demasiado. Nuestra solución será sencillamente no usar tildes o diéresis al clasificar caracteres.

c)
```
a = (4, 1, 2, 3, 5)
```

```
sorted(a)                # da una lista
sorted(a, reverse=True)  # orden inverso
```
¶

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.

Ejercicio 14.2 (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', 'Pedro',
           '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. ¶

Ejercicio 14.3. Definir una función que tome como argumento una cadena de caracteres y escriba los caracteres ingresados y la cantidad de apariciones, ordenados decrecientemente según la cantidad de apariciones, y alfabéticamente ante igualdad de apariciones.

Por ejemplo, si argumento es `'mece'`, se debe imprimir algo como:

```
Caracter            cantidad
   e                   2
   c                   1
   m                   1
```

Aclaración: suponemos que la cadena tiene sólo letras y números, pero no letras con tildes ni signos de puntuación.

✍ Ver también el ejercicio 14.6. ¶

Ejercicio 14.4 (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],..., a[n-1]]
```

son enteros que satisfacen `0 <= a[i] < m` para $i = 1, 2, \ldots, n$, 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 14.1 (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:

```
# al principio, las cajas están vacías
cuenta = [0 for i in range(m)]

# ponemos cada bolita en su caja,
```

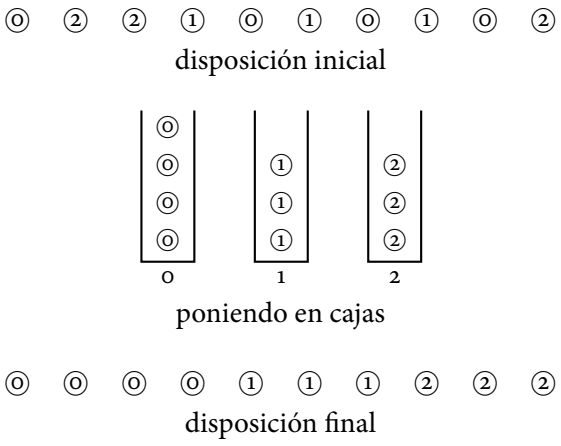


Figura 14.1: Ordenando por conteo.

```
# 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). Luego comparar con `sort` cuando `a` es una lista de 1000 números enteros aleatorios entre 1 y 10.

b) ¿Cómo podríamos encontrar `m` si no se conoce de antemano?

Aclaración: suponemos dada una lista con números enteros no negativos.

c) ¿Se podría cambiar el lazo `for k in a...` por una función o método de Python?

✍ 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,[2] *y se usa en muchas situaciones. Por ejemplo, una forma de clasificar cartas (naipes) es ordenarlas primero según el «palo» y después ordenar cada palo.* ¶

14.2. Listas como conjuntos

A veces es conveniente tratar a una sucesión como un conjunto, es decir, no nos interesan los elementos repetidos ni el orden. En estos casos, es conveniente eliminar los elementos repetidos, procedimiento que llamamos «purga».

✍ 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.

Ejercicio 14.5 («purgar» una lista). En este ejercicio vamos a purgar una lista dada, eliminando los elementos repetidos pero conservando el orden original 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

[2] *Bucket*: balde o cubo, *bin*: caja, *sort*: clasificación.

```
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
a[m:] = []  # considerar sólo a[0],..., a[m-1]
```

en el apartado anterior, empezando desde 1 y no desde 0? ¶

Ejercicio 14.6.

a) Definir una función `cuentas(a)` dada una lista `a` retorna una nueva lista `[[x, p], [y, q],...]` donde `x`, `y`,... son los elementos de `a` *sin repeticiones*, y `p`, `q`,... es la cantidad de apariciones.

Por ejemplo, si `a = [1, 3, 1, 4, 3, 1, 4, 2]`, el resultado debe ser `[[1, 3], [3, 2], [4, 2], [2, 1]]`.

b) Modificar la función anterior para retornar la lista ordenada decrecientemente según la cantidad de apariciones (a igual cantidad, ordenada crecientemente).

✍ Comparar con el ejercicio 14.3. ¶

Ejercicio 14.7 («purgar» una lista ordenada). En este ejercicio queremos eliminar elementos repetidos de la lista `a` que está ordenada de menor a mayor. Por ejemplo, si `a = [1, 2, 2, 5, 6, 6, 9]`, queremos obtener `[1, 2, 5, 6, 9]`.

Podríamos usar directamente el ejercicio 14.5, pero una variante del apartado *b*) es más eficiente:

```
m = 0                # a[0],...,a[m - 1]
                     # son los elementos sin repetir
for i in range(1, n):   # n es la longitud de a
    if a[m] < a[i]:     # incluir a[i] si no es a[m]
        m = m + 1
        a[m] = a[i]
a[m+1:] = []         # considerar sólo a[0],..., a[m]
```

Construir una función `purgarordenado` que modifica su argumento siguiendo este esquema.

✍ La eficiencia se refiere a que el método del ejercicio 14.5 en general hace del orden de n^2 comparaciones, mientras que el presentado acá usa del orden de n. ¶

Ejercicio 14.8 (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]` (`a` no debe modificarse).

✍ En general (no siempre) es más eficiente primero clasificar y después purgar (según el ejercicio 14.7) que purgar primero (según el ejercicio 14.5) y después clasificar. ¶

Ejercicio 14.9 («unión» de listas). En este ejercicio queremos construir la «unión» de las listas `a` y `b`, es decir, una 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`).

a) Construir una función poniendo una lista a continuación de la otra:

```
c = a + b
```

y luego clasificando y purgando (con `sort` y `purgarordenado` respectivamente) la lista `c`.

b) 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 a y b ya están clasificadas y purgadas,
# con longitudes n y m respectivamente
i, j = 0, 0     # índices para a y b
c = []          # no hay elementos copiados
while (i < n) and (j < m):
    if a[i] < b[j]:
        c.append(a[i])  # copiar en c
        i = i + 1       # y avanzar en a
    elif a[i] > b[j]:
        c.append(b[j])  # copiar en c
        j = j + 1       # y avanzar en b
    else:               # a[i] == b[j]
        c.append(a[i])  # copiar en c
        i = i + 1       # y avanzar en
```

```
        j = j + 1          # ambas listas
c = c + a[i:]              # copiar el resto de a
c = c + b[j:]              # copiar el resto de b
```

✍ Este procedimiento de combinar listas ordenadas se llama «fusión» (*merge*).

✍ En general, este método es más eficiente que el del apartado anterior. ¶

Ejercicio 14.10 («intersección» de listas). Del mismo modo, podemos considerar la «intersección» de las listas `a` y `b`, es decir, una lista ordenada `c` con todos los elementos que están en `a` y `b` simultáneamente (pero aparecen en `c` una sola vez). Por ejemplo, si `a = [3, 1, 4, 2, 3, 1]` y `b = [2, 3, 1, 5, 2]`, debe ser `c = [1, 2, 3]`. Si `a` y `b` no tienen elementos comunes, `c` es la lista vacía. Como en la «unión», la «intersección» de `a` con `a` son los elementos de `a` clasificados y sin repeticiones.

Construir una función copiando las ideas del ejercicio 14.9.*b*).

Sugerencia: eliminar algunos renglones en el esquema, copiando en `c` sólo si el elemento está en ambas listas. ¶

Ejercicio 14.11 (estadísticos). Cuando se estudian estadísticamente los datos numéricos $(a_1, a_2, \ldots, a_n)$, se consideran (entre otros) tres tipos de «medidas»:

***La media o promedio*:** $\dfrac{1}{n}\sum_{i=1}^{n} a_i$.

***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_ℓ 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 14.4. ¶

14.3. Búsqueda binaria

Ejercicio 14.12 (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». ¶

Ejercicio 14.13. 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 14.12.

Ejercicio 14.14 (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] <= a[1] <=...<= 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:
        poco = medio
    else:
        mucho = medio
# 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

$$mucho = poco + 1, \tag{14.1}$$

como $medio = \lfloor (poco + mucho)/2 \rfloor$ obtenemos

$$medio = poco. \tag{14.2}$$

i) Verificar que (14.1) implica (14.2).

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, 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 14.12)? 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*). ¶

14.4. Ejercicios adicionales

Ejercicio 14.15 (generala). En el juego de la generala se tiran 5 dados y se tiene:

- «full» si hay 3 dados con un mismo número y 2 con otro número,
- «póker» cuando hay 4 dados iguales (pero no 5),
- «escalera» cuando hay 5 números consecutivos, y
- «generala» cuando los 5 dados son iguales.

Definir una función para simular este juego, imprimiendo alguna de las posibilidades anteriores si salió, o si no salió ninguna. ¶

14.5. 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.
- Python usa funciones *hash*[3] 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.

[3] Normalmente traducidas como *transformaciones de llave (o clave)*.

Capítulo 15

El módulo *graficar* y funciones numéricas

En este capítulo vemos cómo Python nos puede ayudar en el estudio de funciones $f : A \to \mathbb{R}$, (donde A es un intervalo o eventualmente todo $\mathbb{R}$) como las que se ven en cálculo o análisis matemático, visualizándolas con el módulo *graficar*.

Este módulo no estándar nos permite hacer gráficos en dos dimensiones de puntos y curvas, aprovechando las facilidades gráficas del módulo *tkinter* de Python, y se puede bajar de la página del libro.

graficar es muy elemental, muy lejos de las posibilidades gráficas de, por ejemplo, *Mathematica*, pero es adecuado para nuestros propósitos, no tenemos que instalar programas adicionales, y es independiente del sistema operativo pues está escrito (claro) en Python.

✍ El módulo no estándar *matplotlib* tiene muchas más posibilidades. Lamentablemente no está disponible para todos los sistemas operativos en las versiones más recientes de Python.

En el curso usaremos *graficar* como «caja negra», como hacemos con *math*, sólo usaremos algunos comandos y no estudiaremos las instrucciones en él. En este capítulo nos concentramos en los gráficos

de funciones, pero —como ya hemos visto en el capítulo 13— *graficar* también nos permite agregar otros elementos, como puntos, rectas, tangentes, poligonales, texto y otros más.

Ejercicio 15.1.

a) Copiar el módulo *graficar* en el directorio donde se guardan nuestros módulos, y poner `help(graficar)` y después, por ejemplo, `help(graficar.funcion)` (`funcion` sin tildes).

✍ Recordar que antes de usar `help` hay que colocarse en el directorio correcto y usar `import`.

b) Ejecutar el módulo *grseno* para hacer un gráfico del seno entre 0 y π, con los valores de las opciones por defecto.

En el contenido de *grseno*, observamos que:

- El primer argumento de `graficar.funcion` es la función, que debe estar definida con anterioridad, y los otros dos son los extremos del intervalo (cerrado) donde hacer el gráfico.
- *graficar* espera que primero se definan los elementos del gráfico, y luego lo construye con `graficar.mostrar()`.
- El gráfico no conserva la escala 1 : 1 entre los ejes, sino que —por defecto— pone el gráfico en el tamaño de la ventana disponible.
- Los ejes x y y se dibujan en el borde, dejando más «limpio» el gráfico, y pueden no cortarse en $(0, 0)$.

c) Podemos cambiar o agregar algunos elementos del gráfico.
Por ejemplo, si ponemos en el módulo *grseno*, antes de `graficar.mostrar()`:

i) `graficar.titulo = 'Seno en [0, pi]'`, el título de la ventana cambiará acordemente.

✍ Sabiendo la forma de introducir el símbolo π (que depende del sistema operativo y teclado), no hay problema en reemplazar `pi` por π en el título pues usamos utf-8.

Siempre se pueden ingresar caracteres utf-8 con códigos especiales (independientes del sistema operativo o teclado), pero para lo que hacemos no vale la pena meterse en ese lío: en todo caso dejamos `pi`.

ii) `graficar.ymin = -0.2` y `graficar.ymax = 1.2` hacen que el gráfico tenga un poco más de espacio en el eje y.

iii) En cambio, «rebanamos» el gráfico con `graficar.ymin = 0.2` y `graficar.ymax = 0.8`.

iv) Con `graficar.aspecto = True` hacemos que los ejes tengan la misma escala. En este caso, el alto se determina a partir del ancho, aunque algunos elementos gráficos pueden no quedar bien. ¶

Ejercicio 15.2. Graficar:

a) $|x|$ en el intervalo $[-2, 2]$.

b) $\cos x$ en el intervalo $[-\pi, 2\pi]$.

c) $\sqrt{x}$ en el intervalo $[0, 10]$. ¶

Ejercicio 15.3. *graficar* permite hacer el gráfico de varias funciones simultáneamente, alternando los colores automáticamente. En estos casos conviene poner leyendas para distinguirlas, como hacemos en el módulo *grexplog*, donde graficamos las funciones e^x entre -2 y 5, $\log x$ entre (casi) 0 y 5,[(1)] y las comparamos con la función identidad, $\mathrm{Id}(x) = x$, puesto que siendo inversas una de la otra, los gráficos de e^x y $\log x$ son simétricos respecto de la diagonal $y = x$.

a) El gráfico no refleja bien la simetría respecto de la diagonal $y = x$, debido a que las escalas de los ejes son distintas. Ver si poniendo `graficar.aspecto = True` mejora el gráfico.

b) La posición de las leyendas está indicada como `'NO'` por esquina *NorOeste*.

i) Comentar el renglón correspondiente y ver cuál es la posición de las leyendas por defecto.

(1) Ver también el ejercicio 15.6.

ii) Cambiar el módulo de modo que las leyendas aparezcan en la esquina sureste.

c) Modificar *grexplog* para hacer un gráfico similar para las funciones x^2 y $\sqrt{x}$ (en vez de exp y log), tomando `xmin = 0`, `xmax = 10` y `ymin = xmin`:

i) Sin explicitar el valor de `graficar.ymax` (por ejemplo, comentando el renglón correspondiente).

ii) Tomando `graficar.ymax = 10`. ¶

Ejercicio 15.4. Cuando queremos hacer el gráfico de una función que no tiene una expresión sencilla, como $f(x) = x^2 + 3x - 1$, lo más simple es definirla primero.

a) En un módulo o la terminal poner:

```
def f(x):
    """Función a graficar."""
    return x**2 + 3*x - 1
graficar.funcion(f, -3, 1)
graficar.titulo = 'Gráfico de x**2 + 3*x - 1'
graficar.mostrar()
```

b) Las rectas pueden darse por dos de sus puntos o bien por punto y pendiente. Por ejemplo, si queremos agregar rectas representando los ejes coordenados, podríamos poner antes de `graficar.mostrar()`:

```
graficar.recta((0, 0), 0)          # eje x
graficar.recta((0, 0), (0, 1))     # eje y
```

c) El problema es que queda demasiado colorinche: la alternancia de colores es la misma para curvas, poligonales o rectas. Pero podemos ponerles colores distintos individualmente.

Cambiar las instrucciones anteriores a:

```
graficar.recta((0, 0), 0,
               estilo = {'fill': 'gray'})
```

```
graficar.recta((0, 0), (0, 1),
               estilo = {'fill': 'gray'})
```

de modo que ahora los ejes aparecen en color gris.

d) Las marcas en el eje vertical no parecen razonables. Agregar antes de `graficar.mostrar`:

```
graficar.yticks = list(range(-3, 4, 2))
```

e) Para graficar la tangente en $x = -0.5$, podemos poner (siempre antes de `graficar.mostrar`):

```
x = -0.5
y = f(x)
graficar.tangente(f, x)
graficar.puntos([(x, y)])  # lista de un punto
```

mostrando también el punto $(x, f(x))$. ¶

Ejercicio 15.5. Recordando el ejercicio 7.7:

a) Sin hacer el gráfico, ¿qué tipo de función es la que expresa c en términos de f (lineal, cuadrática,...)?

b) Hacer el gráfico de las funciones `acelsius` y la identidad en el intervalo $[-50, 50]$, y comprobar que las curvas se cortan en el punto obtenido en el ejercicio 7.7.*e*) usando el cursor.

c) Usando `leyendax` y `leyenday`,[2] cambiar los nombres de los ejes de modo que en el eje horizontal aparezca la leyenda `f` (en vez de `x`) y en el vertical la leyenda `c` (en vez de `y`), y poner un título adecuado al gráfico. ¶

Ejercicio 15.6 (gráfico de funciones con saltos). *graficar* hace el gráfico evaluando la función en algunos puntos[3] y luego uniéndolos, por lo que las funciones discontinuas no se grafican correctamente.

a) Supongamos que queremos graficar la función $y = 1/x$ —que no está definida para $x = 0$— en el intervalo $[-1, 1]$.

[2] En todo caso poner `help(graficar)`.

[3] Por defecto, 201 puntos equiespaciados.

i) El bloque

```
import graficar
def f(x):
    return 1/x
graficar.funcion(f, -1, 1)
graficar.mostrar()
```

posiblemente dará error de división por cero.

ii) Podemos cambiar la opción `npuntos` de gráficos de funciones, de modo de no evaluar $1/x$ para $x = 0$. Para un intervalo simétrico alrededor de 0, podemos poner un número par de puntos: cambiar el renglón que empieza con `graficar.funcion` en el bloque anterior por

```
graficar.funcion(f, -1, 1, npuntos=100)
```

y ver el resultado.

iii) Una solución más satisfactoria es dividir el intervalo en dos o más para evitar los saltos, como hacemos en el módulo *gr1sobrex* en el que se puede ajustar `eps`.

✍ Ponemos explícitamente el estilo (en este caso, color azul) para evitar que las ramas de la hipérbola se dibujen con distintos colores.

b) La función tangente, tan, no está definida en $\pi/2 + k\pi$, para $k \in \mathbb{Z}$. Hacer un gráfico de esta función en el intervalo $(-\pi/2, \pi)$.

✍ La evaluación de Python de tan $\pi/2$ no da error, como vimos en el ejercicio 3.13. ¶

Ejercicio 15.7. Recordando el ejercicio 8.5:

a) Hacer un gráfico aproximado (sin tener en cuenta saltos) del impuesto y la ganancia neta, cuando la ganancia bruta está entre \$ 0 y \$ 150 000.

b) En el gráfico se podrá observar que la ganancia neta para un ingreso de \$ 100 000 es mayor que la correspondiente a ingresos un poco mayores.

Usando el ratón, determinar un valor aproximado de x tal que la ganancia neta para un ingreso de \$ 100 000 + x sea igual a la ganancia neta para un ingreso de \$ 100 000. Luego determinar matemáticamente x. ¶

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.[1] 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 3.10, Python puede trabajar con cualquier número entero (sujeto a la memoria de la compu-

[1] 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 0.123×10^{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, no se decidió hacer algo similar con los decimales, que también mediante fraccionamiento y rearmado podrían tener tanta precisión como se quisiera. Esto se puede hacer con 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 1.0×10^{-3} está en forma

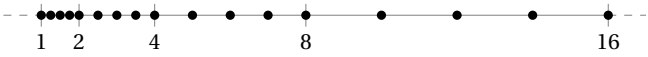


Figura 16.1: Esquema de la densidad variable en la codificación de números decimales.

normal, pero no 100.0×10^{-5} ni 0.001.

No es que sólo números grandes o irracionales no se puedan representar, como vemos en el siguiente ejercicio.

Ejercicio 16.1. Decimos que x, $0 < x < 1$, puede representarse como suma finita de potencias de 2 si podemos encontrar $a \in \mathbb{N}$ y $n \in \mathbb{N}$ tales que

$$x = a \times 2^{-n} = \frac{a}{2^n}.$$

Por ejemplo, 0.5 se puede representar de esta forma pues $0.5 = 1 \times 2^{-1}$.

a) 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.

b) Como las representaciones no son exactas, hay errores en las asignaciones `a = 0.1`, `b = 0.2` y `c = 0.7`.

Efectivamente, ver que `a + b + c` y `b + c + a` dan valores distintos, si bien muy parecidos. ¶

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.

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

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*.

Ejercicio 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 `a == 1 + a`.

Ponemos:

```
a = 1.0              # y no a = 1 !!!
while 1 + a > a:
    a = 2 * a
a
```

Si en vez de `a = 1.0` *ponemos* `a = 1` *inicialmente, tendremos un lazo infinito (¿por qué?).*

a) Encontrar n tal que el valor de `a` es 2^n de dos formas: usando logaritmos y poniendo un contador en el lazo `while`.

b) Ver que el valor de `a` es efectivamente `a + 1` pero distinto de `a - 1`: `a - 1` es el mayor decimal en Python tal que sumándole 1 da un número distinto.

c) Calcular `a + i` para `i` entre 0 y 6 (inclusive). ¿Son razonables los resultados? ¶

Ejercicio 16.3 ($\varepsilon_{\text{máq}}$). Esencialmente dividiendo por $a > 0$ la desigualdad $a + 1 > a$ en el ejercicio anterior 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 $\varepsilon_{\text{máq}}$, al que podemos interpretar 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.

a) La función `epsilon` (en el módulo *decimales*) es una variante de la expresión anterior, toma como argumento el valor `inic`, que suponemos positivo, y retorna el menor `x` positivo tal que `inic + x > inic`.[(2)] Ponemos `epsmaq = epsilon(1.0)` (el `b` anterior), o en notación matemática, $\varepsilon_{\text{máq}}$.

La función `epsilon` nos ayudará a entender la densidad variable:

i) Comparar los valores de `epsilon` con argumentos $1, 1 + 1/2, 1 + 3/4, \ldots, 2 - 2^{-10}$, y luego con argumentos $2, 3, 3 + 1/2, \ldots, 4 - 2^{-9}$.

Sugerencia: hacer listas por comprensión (observar también que $2 = 2 \times 1$, $3 = 2 \times (1 + 1/2)$,..., $4 - 2^{-j} = 2 \times (2 - 2^{-j-1})$,...).

(2) Suponemos también que `inic + 1 > inic`.

ii) Suponiendo que el valor de `epsilon(x)` se mantiene constante para x en el intervalo $I_k = [2^{k-1}, 2^k)$ para $k \in \mathbb{N}$, ver que la cantidad de números que se pueden representar en I_k es independiente de k y luego calcular esa cantidad.

iii) Evaluar `2**k / epsilon(2**k)` para $k = 1, \ldots, 10$, y ver que es una potencia de 2 (¿cuál?). ¿Cómo se relacionan estas cantidades con lo hecho en el apartado anterior?

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 $\varepsilon_{máq}$?

Sugerencia: multiplicar `a` y `epsmaq`. ¶

Ejercicio 16.4 ($\varepsilon_{mín}$). Otro indicador importante es $\varepsilon_{mín}$, el *épsilon mínimo*, que es el menor número decimal positivo que se puede representar.

✍ Podemos pensar que *para la computadora* $\varepsilon_{mín}$ es el decimal que sigue a 0.

Decir por qué falla el esquema de ejercicios anteriores:

```
c = 1.0
while c > 0:
    c = c / 2
c = 2 * c   # nos pasamos: volver al anterior
```

En el módulo *decimales* construimos `epsmin` eliminando el problema al conservar el último valor no nulo: comprobar que su comportamiento es correcto. ¶

Ejercicio 16.5 (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 3.10, 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 3.16, 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 $\varepsilon_{\text{mín}}$, en el módulo *decimales* construimos el número `maxpot2`, la máxima potencia de 2 que se puede representar.

Encontrar `n` tal que `maxpot2` es aproximadamente `2**n`.

d) En analogía con el ejercicio 16.3.*b*), averiguar la relación entre `epsmin` y `maxpot2`.

e) Usando el valor de `n` obtenido en *c*), ver que poniendo

```
y = 2**(n + 1) - 1
```

no hay error porque es un número entero.

✍ Por la forma en que trabaja Python, `float(y)` puede dar error aún cuando se puede representar como suma de potencias de 2 hasta n ($y = \sum_{k=0}^{n} 2^k = 2^{n+1} - 1$).

Poniendo `x = maxpot2`, y considerando desde las matemáticas los valores de `n` y `y`, tendríamos (usando que $n > 1$),

$$\begin{aligned} x = 2^n < y = 2^{n+1} - 1 < 2^{n+1} = 2x \\ < 2^{n+1} + (2^{n+1} - 2) = 2^{n+2} - 2 = 2y, \end{aligned}$$

pero Python piensa distinto:

```
2 * x           # -> inf
x < y           # -> verdadero
y < 2 * x       # -> verdadero
2 * x < 2 * y   # -> falso
2 * x > 2 * y   # -> verdadero
```

f) `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 * x               # -> 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`. ¶

Ejercicio 16.6. Recordando lo expresado al principio y como repaso de lo visto, para cada caso encontrar decimales `x`, `y` y `z` tales que los siguientes den verdadero:

a) `x + y == x` con `y` positivo.

b) `(x + y) + z != x + (y + z)`.

c) `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 ejercicios vemos cómo estos pequeños errores pueden llevar a conclusiones desastrosas.

Ejercicio 16.7 (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.

a) Estudiar las funciones `mcd1` y `mcd2`, y ver que se obtienen resultados esperados con los argumentos `315` y `216`.

b) 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) 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). 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 $\sqrt{\varepsilon_{\text{máq}}}$.

Ejercicio 16.8 (problemas con la ecuación cuadrática). Como sabemos, la ecuación cuadrática

$$ax^2 + bx + c = 0 \tag{16.1}$$

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

$$x_1 = \frac{-b + \sqrt{d}}{2a} \qquad \text{y} \qquad x_2 = \frac{-b - \sqrt{d}}{2a}. \tag{16.2}$$

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. ¶

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 cos, sen, log, etc., 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.

Ejercicio 16.9 (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

$$\underbrace{\sqrt{\sqrt{\cdots\sqrt{\sqrt{x}}}}}_{n \text{ raíces}} \quad (= x^{1/2^n}),$$

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 $f(x) = \cos x$.

Ejercicio 16.10. Haciendo un gráfico combinado de $f(x) = \sqrt{x}$ y de $y = x$ para $0 \le x \le 4$ con el módulo *graficar*, encontrar otro punto fijo de f (distinto de 1). ¶

Ejercicio 16.11. 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 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 $x \ge 0$).

✍ 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,

- $\operatorname{sen} 1/x$ para $x > 0$, pues cerca de $x = 0$ oscila demasiado,
- $$f(x) = \begin{cases} x \operatorname{sen} 1/x & \text{si } x \neq 0, \\ 0 & \text{si } x = 0 \end{cases}$$
  pues cerca de $x = 0$ oscila demasiado,

y hay funciones que no son continuas como

- $\operatorname{signo}(x)$ 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

$$x_1 = f(x_0),\ x_2 = f(x_1), \ldots,\ x_n = f(x_{n-1}), \ldots$$

y supongamos que tenemos la suerte que x_n se aproxima o *converge* al número ℓ 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, ℓ 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.

Probando la técnica mencionada, dado $x_0 \in \mathbb{R}$ definimos

$$x_{n+1} = \cos x_n \quad \text{para } n = 0, 1, 2, \ldots,$$

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

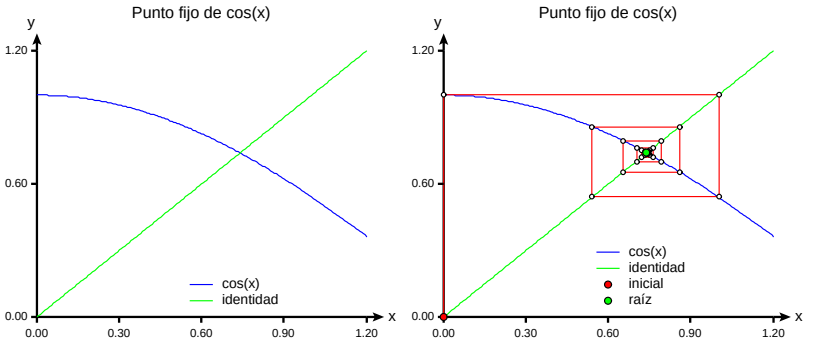


Figura 16.2: Gráfico de $\cos x$ y x (izquierda) y convergencia a punto fijo desde $x_0 = 0$ (derecha).

---

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.

Ejercicio 16.12 (punto fijo de $f(x) = \cos x$). Con las notaciones anteriores para $f(x) = \cos x$ y x_i:

a) 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.

b) Modificar la función para que también imprima $\cos x_n$ y comprobar que para n más o menos grande se tiene $x_n \approx \cos x_n$.

c) 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.

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 $|f(x_k) - x_k| < \varepsilon$. ¶

Ejercicio 16.13. La función `puntofijo` (en el módulo *numerico*) sintetiza lo hecho en el ejercicio 16.12: retorna un cero de la función con una tolerancia permitida en un número máximo de iteraciones.

a) Modificar la función de modo de que el número de iteraciones máximas y la tolerancia sean argumentos.

b) Modificar la función de modo de siempre imprimir 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 tradicional 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 ε_x, ε_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.

Ejercicio 16.14. La figura 16.2 fue hecha con el módulo *grpuntofijo*, que grafica la función y los puntos que se obtienen a partir del método. Ejecutarlo y comprobar que se obtienen resultados similares a la figura mencionada.

✍ *grpuntofijo* es una «plantilla» para explorar el comportamiento de distintas funciones cambiando algunos parámetros. Usa los módulos *graficar* y *grnumerico* que usaremos como «cajas negras».

Usar *grpuntofijo* para ilustrar el método de punto fijo:

a) Para la función $\sqrt{x}$ en el intervalo $[0, 3]$ con los puntos iniciales 0, 0.5, 1 y 2.

b) Para la función x^2, con el mismo intervalo y puntos iniciales. ¶

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, π es una raíz de la ecuación $\tan x = 0$, y para obtener un valor aproximado de π podemos tomar $g(x) = \tan x$, $f(x) = x - \tan x$, y usar la técnica anterior.

Ejercicio 16.15. Resolver los siguientes apartados con la ayuda del módulo *grpuntofijo* para ver qué está sucediendo en cada caso.

a) 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 π. Ver que π es uno de los infinitos puntos fijos.

b) Usando la función `puntofijo`, verificar que con 3 o 4 iteraciones se obtiene una muy buena aproximación de π comenzando desde $x_0 = 3$.

c) 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 $1.5, 1.51, \ldots, 1.6$ (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 del apartado *a*) no varían. Hacer los apartados *b*) y *c*) con esta variante y verificar si se obtienen resultados similares. ¶

Ejercicio 16.16. 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 $x_0 > 1$.

Usando el módulo *grpuntofijo*, hacer un gráfico de tres o cuatro iteraciones en los casos mencionados para verificar el comportamiento. ¶

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

La técnica de punto fijo para encontrar raíces de ecuaciones no surgió con las computadoras. Por ejemplo, el *método babilónico* 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

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

es decir,

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{si } |h| \text{ es suficientemente chico.} \tag{16.3}$$

Si la derivada existe, o sea, si los cocientes incrementales 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 (16.3), llegamos a

$$f(x+h) \approx f(x) + f'(x)\, h,$$

y como $h = x^* - x$, queda

$$0 = f(x^*) = f(x) + f'(x)\,(x^* - x).$$

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*, considerando la sucesión

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \text{para } n = 0, 1, \ldots, \tag{16.4}$$

siempre que f' no se anule en los puntos x_n.

Podemos interpretar la ecuación (16.4) como diciendo que buscamos un punto fijo de la función

$$g(x) = x - \frac{f(x)}{f'(x)}. \tag{16.5}$$

Ejercicio 16.17 (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 $b^2 = a$, tratamos de encontrar un cero de la función $f(x) = x^2 - a$.

a) Encontrar la función g dada en la ecuación (16.5) 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

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right) \qquad \text{para } n = 1, 2, \ldots, \tag{16.6}$$

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 (16.6) 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 (16.4), 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

$$f'(x) \approx \frac{f(x + \Delta x/2) - f(x - \Delta x/2)}{\Delta x}, \tag{16.7}$$

donde Δx es una constante dada, ya que es mucha mejor aproximación que la dada en (16.3).

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 (16.3).

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:

$$f(x + h) = f(x) + f'(x)\, h + f''(x)h^2/2 + O(h^3),$$

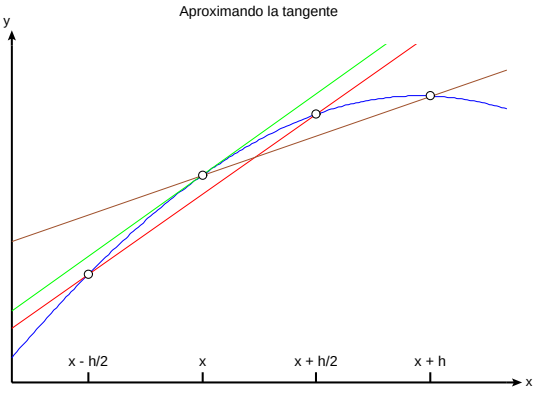


Figura 16.3: Aproximando la pendiente de la recta tangente.

de modo que el error para calcular $f'(x)$ en (16.3) es

$$\frac{f(x+h) - f(x)}{h} - f'(x) = O(h),$$

mientras que el error en (16.7)

$$\frac{f(x+h) - f(x-h)}{2h} - f'(x) = O(h^2).$$

✍ 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 (16.7) 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 196.

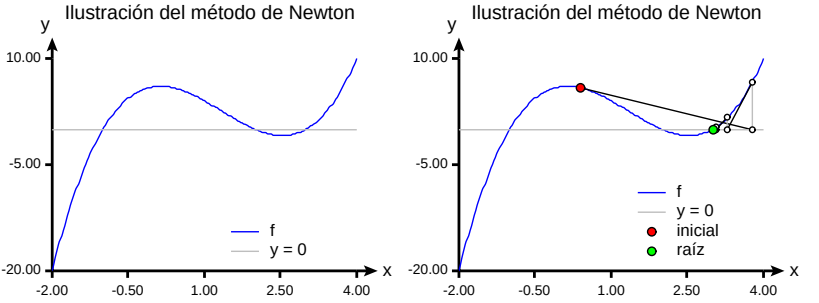


Figura 16.4: Gráfico de f en la ecuación (16.8) (izquierda) y convergencia del método de Newton desde $x_0 = 0.4$ (derecha).

✍ En la función `newton` no consideramos la posibilidad $f'(x) = 0$.

Ejercicio 16.18 (Newton con derivadas aproximadas). Tomemos

$$f(x) = (x+1)(x-2)(x-3), \tag{16.8}$$

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. Esta figura fue hecha con el módulo *grnewton*, que es una «plantilla» para ilustrar el método de Newton, análoga al módulo *grpuntofijo*.

a) Ejecutar el módulo *grnewton*, viendo que se obtienen resultados similares a los de la figura.

b) Considerar otros puntos iniciales para obtener las otras dos raíces (-1 y 2). ¶

Ejercicio 16.19. Resolver los siguientes apartados con la ayuda del módulo *grnewton* y la función `newton`:

a) Encontrar soluciones aproximadas de la ecuación $\cos x = 0$ tomando puntos iniciales 1.0 y 0.0.

b) Encontrar una solución aproximada de $\cos x = x$ y comparar con los resultados del ejercicio 16.12.

c) Encontrar aproximadamente todas las soluciones de las ecuaciones:

i) $x^2 - 5x + 2 = 0$ *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) $2 - \ln x = x$, *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 $i = 0, 1, 2, \ldots$ 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.

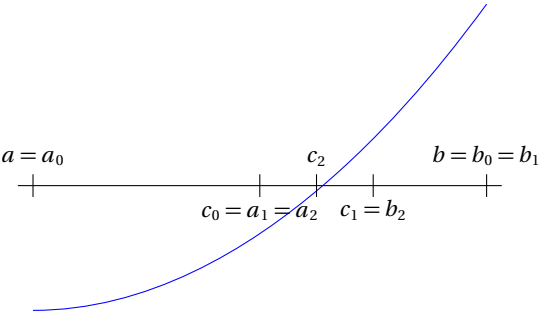


Figura 16.5: Función continua con distintos signos en los extremos.

✍ 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 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 pág. 201), 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, el valor ε_y 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.

Ejercicio 16.20 (método de la bisección). Consideremos la función

$$f(x) = x\,(x+1)\,(x+2)\,(x-4/3),$$

que tiene ceros en $x = -2, -1, 0, 4/3$, como se ilustra en la figura 16.6 (izquierda). Usando el método de la bisección para esta función tomando como intervalo inicial $[-1.2, 1.5]$, obtenemos una sucesión de

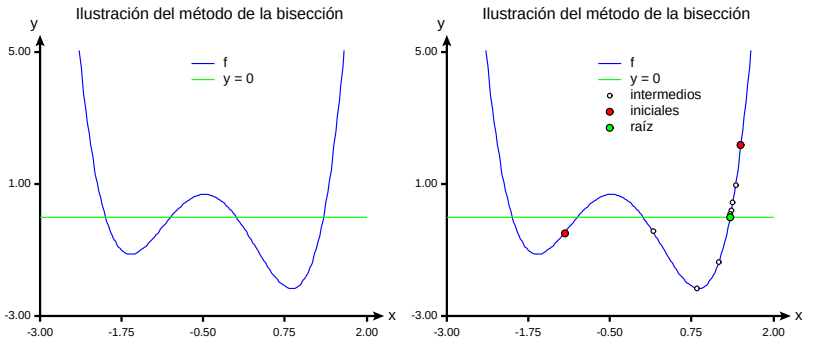


Figura 16.6: Método de bisección (ejercicio 16.20) función (izquierda) y puntos obtenidos para el intervalo inicial $[-1.2, 1.5]$ (derecha).

intervalos dados por los puntos que se muestran en la misma figura a la derecha.

a) En caso de que haya más de una raíz en el intervalo inicial, la solución elegida depende de los datos iniciales. Verificar este comportamiento ejecutando `biseccion` sucesivamente con los valores .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: recordar que 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 ε_x y ε_y (en vez de ε), de modo de terminar las iteraciones si $|mucho - poco| < \varepsilon_x$ o si $|f(medio)| < \varepsilon_y$.

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 ε_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 14.3) son esencialmente la misma cosa. Si tenemos una lista de números no decreciente $a_0 \le a_1 \le \cdots \le a_n$, uniendo los puntos $(k-1, a_{k-1})$ con (k, a_k) para $k = 1, \ldots, n$, 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

$$medio = \left\lfloor \frac{poco + mucho}{2} \right\rfloor,$$

terminando en cuanto la diferencia en x es 1, que se traduce como

$$mucho - poco \le 1.$$

Ejercicio 16.21. La figura 16.6 se hizo con el módulo *grbiseccion*, que —como los módulos *grpuntofijo* y *grnewton*— es una «plantilla», en este caso para ilustrar el método de la bisección. Ejecutar el módulo, viendo que se obtienen resultados similares.

Con la ayuda de este módulo y la función `biseccion`, usar el método de la bisección para resolver los apartados del ejercicio 16.19, y comparar con las soluciones obtenidas allí. ¶

Ejercicio 16.22 (interés sobre saldo). Maggie quiere comprar una «tablet» y en un folleto de propaganda vio que podía comprar una por

$ 1099 de contado o en 24 cuotas mensuales de $ 79.90. Si dispone de los $ 1099, ¿le conviene comprarla en efectivo o en cuotas?

Para analizar el problema, primero veamos cuál es la tasa de interés (nominal anual) que cobra el negocio. Hay muchas formas de calcularla y nos vamos a concentrar en el llamado *interés sobre saldo* que vimos en la sección 11.2, y particularmente el ejercicio 11.5.

Si pedimos prestada una cantidad c (en $) con un tasa de interés anual (nominal) r (en %), que pagaremos en cuotas mensuales *fijas* de p (en $) comenzando al final del primer mes, y que el préstamo tiene las característica de que el interés se considera sobre el saldo, esto es, si b_m es el saldo adeudado a fin del mes m, *justo antes* de pagar la cuota m-ésima, y c_m es el saldo *inmediatamente después* de pagar esa cuota, poniendo $t = 1 + r/(100 \times 12)$, tendremos:

$$b_1 = tc, \quad c_1 = b_1 - p, \quad b_2 = tc_1, \quad c_2 = b_2 - p, \ldots$$

y en general

$$c_m = b_m - p = t\, c_{m-1} - p, \tag{16.9}$$

donde inclusive podríamos poner $c_0 = c$, constituyendo una variante de las ecuaciones (11.7).

a) Considerando que c y r están fijos, ¿existe un valor de p de modo que el saldo sea siempre el mismo, es decir, $c_{m+1} = c_m$ para $m = 0, 1, 2, \ldots$?, ¿cuál?

Respuesta: cuando el interés mensual es la cuota: $cr/1200$.

b) Del mismo modo, encontrar una tasa crítica, $r_{\text{crít}}(c, p)$ (dependiendo sólo de c y p), tal que la deuda se mantenga constante, $c_{m+1} = c_m$ para $m = 0, 1, 2, \ldots$

Respuesta: $1200p/c$.

c) Definir una función `saldo(c, r, p, m)` que dados el monto inicial c, la tasa r, y el pago mensual p, calcule el saldo (la deuda) inmediatamente después de pagar la m-ésima cuota, es decir, c_m = `saldo(c, r, p, m)` para $m = 0, 1, 2, \ldots$

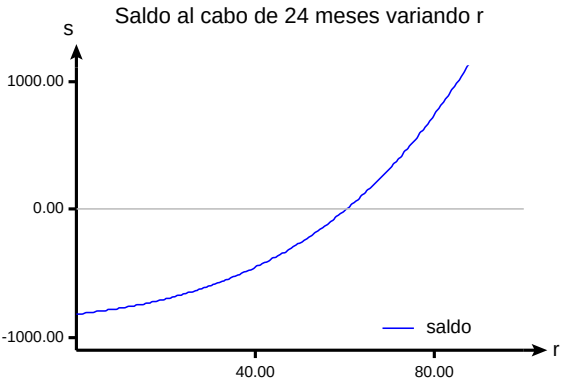


Figura 16.7: Gráfico de `saldo(c, r, p, m)` cuando r varía (c, p y m constantes).

Aclaración 1: no se pide encontrar una fórmula, sólo traducir a Python la ecuación (16.9).

Aclaración 2: suponemos $c > 0$, $r \geq 0$, $p > 0$ y $m \geq 0$.

✍ En el ejercicio 16.24 encontramos una fórmula explícita.

d) Usar la función `saldo` con $c = 1099$ (el precio de la oferta), $p = 79.90$, $r = r_{\text{crít}}$ calculado en *b*) para distintos valores de m.

e) Para c, r y p fijos, la función `saldo(c, r, p, m)` es *decreciente* con m si p es mayor que el monto calculado en *a*) porque cada vez se adeuda menos. ¿Cómo es el comportamiento de `saldo` cuando sólo varía c?, ¿y si sólo varía r?

Respuesta: crecientes en ambos casos.

f) Usando el módulo *graficar*, hacer un gráfico de la función `saldo` cuando $c = 1099$ (lo que pagaría Maggie hoy), $p = 79.90$ (el pago mensual), $m = 24$ (la cantidad de cuotas), variando r entre 0 y 100, confirmando lo visto en el apartado anterior. El gráfico debería ser similar al de la figura 16.7.

g) El gráfico anterior muestra que la curva corta al eje r, y pode-

mos usar el método de la bisección.

¿Qué condiciones sobre c, p y m podríamos pedir para usar `poco = 0` y `mucho = rcrit`, donde `rcrit` es la tasa crítica encontrada en el apartado *b*)?

Respuesta: podríamos poner `poco = 0` si $p \times m \geq c$, y `mucho = rcrit` siempre (suponiendo $c > 0$ y $p > 0$).

h) Definir una función para resolver el apartado anterior en general (para cualesquiera valores de c, p y m), usando el método de la bisección y teniendo cuidado de elegir valores apropiados de `poco` y `mucho` dentro de la función.

i) Calcular la tasa (anual nominal) que pagaría Maggie si decide comprar en cuotas.

Respuesta: 60.39 %.

j) Como dispone ahora de $ 1099, Maggie podría poner ese dinero en certificados a plazo fijo cada mes, extrayendo $ 79.90 mensualmente para pagar la cuota.

Si el banco ofrece una tasa de 12 % (nominal anual), ¿cuántos meses podría pagar la cuota (sacando del certificado del banco)?

Respuesta: 15 meses.

k) En el apartado anterior, ¿qué tasa debería ofrecer el banco para que pagar en cuotas fuera equivalente a hacer certificados en el banco?

Respuesta: la misma del apartado *i*).

l) Suponiendo que la tasa anual de inflación está entre 20 % y 30 %, y que los bancos ofrecen una tasa no mayor al 12 % en certificados a plazo fijo, ¿debería Maggie hacer la compra en efectivo o en cuotas haciendo certificados a plazo fijo en el banco?

✍ En calculadoras financieras y planillas de cálculo están implementadas funciones similares. Por ejemplo, en MS-Excel están las funciones (donde p debe ser negativo si es un pago, r es la tasa y m es la cantidad de cuotas):

AMORT(r, m, c): Calcula p dados m, r y c.

VALACT(r, m, p): Calcula c dados m, r y p.

TASA(m, p, c): Calcula r dados m, c y p.

NPER(r, p, c): Calcula m dados r, a y p.

Por ejemplo, en el problema original pondríamos `TASA(24, -79.90, 1099) * 1200` obteniendo 60.39. ¶

Ejercicio 16.23. En principio no podríamos usar el método de Newton para resolver el ejercicio 16.22, ya que no tenemos una fórmula explícita para la derivada correspondiente. Sin embargo, en el cálculo efectivo usamos una aproximación a la derivada dada por la relación (16.7), y esa objeción desaparece.

✍ En el ejercicio 16.24 vemos una fórmula explícita. De cualquier forma, el cálculo de la derivada no es sencillo.

Resolver el ejercicio 16.22 usando el método de Newton, tomando como punto inicial un valor adecuado (por ejemplo, el promedio de *poco* y *mucho* en 16.22.*g*)) y comparar con los resultados obtenidos mediante bisección.

Sugerencia no muy elegante: para determinar r, usar c, p y m como variables globales (fijas) al definir la función.

Sugerencia más elegante: definir una función que englobe al método de Newton, algo como

```
def sol(c, p, m):
    ...
    def saldo(r):
        ...
    ...
    inic = ...
    return numerico.newton(saldo, inic)
```
¶

Terminamos relacionando algunos conceptos que vimos.

Ejercicio 16.24 (interés sobre saldo II). En el ejercicio 16.22 usamos la expresión (16.9) para expresar la deuda al principio del mes m, c_m.

a) Interpretando (16.9) como la aplicación de la regla de Horner para un polinomio evaluado en t, ¿cuáles serían los coeficientes del polinomio (de grado m)?

✍ En otras palabras, la función `saldo` no hace más que evaluar un polinomio en t.

b) Recordando que $t = 1 + r/1200$, ver que si $r > 0$ podemos poner

$$c_m = c\,t^m - p\,\frac{t^m - 1}{t - 1}.$$

Sugerencia: recordar la suma de la progresión geométrica,

$$\sum_{k=0}^{n} x^k = (x^{n+1} - 1)/(x - 1) \quad \text{si } x \neq 1.$$

✍ Si los pagos o los períodos no fueran uniformes, tendríamos que volver a la versión original. ¶

16.6. Ejercicios adicionales

Los polinomios sirven para aproximar tanto como se desee cualquier función continua, lo que constituye un tema central de las matemáticas. Suponemos conocidos los polinomios de Taylor, y en esta sección estudiaremos los *polinomios interpoladores de Lagrange*.

Así, en la figura 16.8 mostramos cómo aproximar a $\operatorname{sen} x$ mediante desarrollos de Taylor de grados 3 y 5 alrededor de 0, y con el polinomio de Lagrange tomando los valores del seno en $x = 0, \pi/4, \pi/2, \pi$. Como se puede apreciar, cerca de $x = 0$ se obtiene una muy buena aproximación en todos los casos.

❦ *Los polinomios de Taylor usan la información de la función y sus derivadas en un punto, y los de Lagrange usan la información de la función en varios puntos. Una alternativa muy usada, sobre todo en gráficos, son los* splines*, polinomios que utilizan información sobre la función y sus derivadas en varios puntos.*

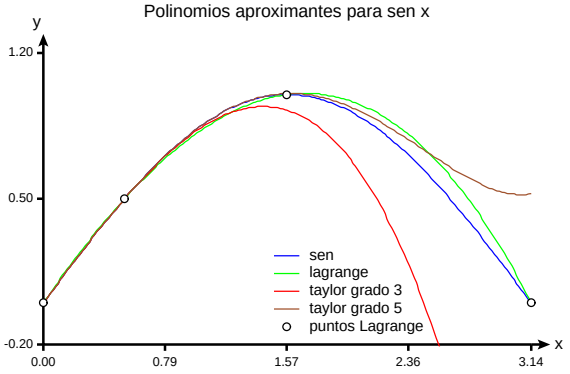


Figura 16.8: Distintas aproximaciones a sen x mediante polinomios.

Ejercicio 16.25 (polinomios interpoladores de Lagrange). Un polinomio $P(x)$ como en la ecuación (11.10), de grado a lo sumo n, está determinado por los $n+1$ coeficientes. Supongamos que no conocemos los coeficientes, pero podemos conocer los valores de $P(x)$ en algunos puntos, ¿cuántos puntos necesitaremos para determinar los coeficientes?

Como hay $n+1$ coeficientes, es natural pensar que quedarán determinados por $n+1$ ecuaciones, es decir, bastarán $n+1$ puntos.

Una forma de resolver el problema es con el *polinomio interpolador de Lagrange*: dados (x_i, y_i), $i = 1, \ldots, n+1$, definimos:

$$P(x) = \sum_{i=1}^{n+1} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}. \tag{16.10}$$

✍ El polinomio en general resulta de grado $\leq n$, y no necesariamente n. Pensar, por ejemplo, en 3 puntos sobre una recta: determinan un polinomio de grado 1 y no 2.

a) Ver que efectivamente, $P(x)$ definido por la ecuación (16.10) satisface $P(x_i) = y_i$ para $i = 1, \ldots, n+1$.

b) Construir una función para evaluar el polinomio $P(x)$ definido

en la ecuación (16.10), donde los datos son $(x_i, y_i), 1 \le i \le n+1$, y x.

Aclaración: sólo se pide una traducción literal de la ecuación.

c) Usarla para calcular el valor del polinomio de grado a lo sumo 3 que pasa por los puntos $(-1, 0), (0, 1), (1, 0), (2, 2)$, en el punto $x = -2$.

d) Usarla para calcular una aproximación de $\operatorname{sen} \pi/4$ $(= \sqrt{2}/2)$ usando los valores del seno para $0, \pi/6, \pi/2$ y π.

✍ Ver la figura 16.8.

e) Calcular aproximaciones a $\operatorname{sen} \pi/4$ usando los polinomios de Taylor de grados 3 y 5 alrededor de 0,

$$T_3(x) = x - \frac{1}{3!}x^3, \qquad T_5(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5,$$

y comparar los resultados con el obtenido en el apartado anterior.

❦ *Joseph-Louis Lagrange (1736–1813) nació en Turín (Italia) bajo el nombre de Giuseppe Lodovico Lagrangia. Fue uno de los fundadores del cálculo de variaciones y las probabilidades, e hizo numerosas contribuciones en otras áreas como astronomía y ecuaciones diferenciales.*

Quienes hayan estudiado cálculo reconocen su nombre por los «multiplicadores» para encontrar extremos de funciones de varias variables, y lo volveremos a encontrar en el ejercicio 17.3. ❡

16.7. Comentarios

- Apenas hemos arañado el caparazón del cálculo numérico.

  Por un lado, hay métodos mucho más avanzados para resolver los problemas presentados acá.

  Por otro lado, hay temas como la resolución de ecuaciones diferenciales o la de sistemas lineales que no hemos tocado en absoluto (estos temas merecen al menos uno o dos cursos destinados exclusivamente a ellos).

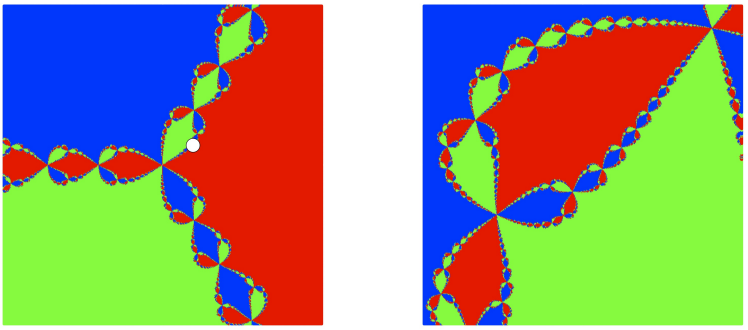


Figura 16.9: Dos imágenes del fractal asociado a la convergencia del método de Newton para $z^3 - 1$.

- 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 16.9.

  Para hacer estos gráficos usamos el método de Newton tratando de encontrar 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. $-1/2 \pm i\sqrt{3}/2$).

  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 $0.3765 + 0.2445\,i$ (punto resaltado a la izquierda en blanco) y lado 0.004.

  Observar que el gráfico a la derecha es un «zoom» por un factor de 1000 del gráfico a la izquierda, lo que también indica que no se trata de una cuestión de errores numéricos sino que

es una propiedad intrínseca.

Este es un ejemplo extremo de lo expresado en el «cartel» sobre cómo depende la solución obtenida del punto inicial (pág. 203).

- Excepto la figura 16.5, todas las figuras en este capítulo, incluso la 16.9, fueron hechas con el módulo *graficar*.

Capítulo 17

Números enteros

En este capítulo nos concentramos en la resolución de problemas de matemática discreta relacionados con números enteros.

17.1. Ecuaciones diofánticas y la técnica de barrido

Comenzamos resolviendo ecuaciones donde las incógnitas son números enteros, y las técnicas del capítulo 16 no pueden aplicarse.

El máximo común divisor y el mínimo común múltiplo de enteros positivos a y b pueden pensarse como soluciones a ecuaciones en enteros. Por ejemplo —como vimos en el problema de Pablito y su papá (ejercicio 8.13)— para el mcm queremos encontrar x e 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.

Este tipo de ecuaciones algebraicas (polinómicas) con coeficientes enteros donde sólo interesan las soluciones enteras se llaman *diofánticas*, 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*.[1]

Ejercicio 17.1. Geri y Guille compraron botellas de vino para una reunión. Ambos querían quedar bien y Guille gastó $ 30 por botella, mientras que Geri, que es más sibarita, gastó $ 50 por botella. Si entre los dos gastaron $ 410, ¿cuántas botellas compró cada uno?, ¿cuánto gastó cada uno?

a) Encontrar una solución (con lápiz y papel).

Respuesta: hay tres soluciones posibles, en las que Geri y Guille compraron (respectivamente) 1 y 12, 4 y 7, 7 y 2 botellas.

b) Definir una función para resolver el problema.

Ayuda: indicando por x la cantidad que compró Geri, y por y la cantidad que compró Guille, se trata de resolver la ecuación $50x + 30y = 410$, con $x, y \in \mathbb{Z}$, $x, y \geq 0$. Por lo tanto, $0 \leq x \leq \lfloor \frac{410}{50} \rfloor = 8$. Usando un lazo `for`, recorrer los valores de x posibles, $x = 0, 1, \ldots, 8$, buscando $y \in \mathbb{Z}$, $y \geq 0$.

✍ Debe descartarse un doble lazo `for` como en el esquema

```
for x in range(9):      # 410 // 50 -> 8
    for y in range(14): # 410 // 30 -> 13
        if 50 * x + 30 * y == 410:
            ...
```

que es sumamente ineficiente pues estamos haciendo $9 \times 14 = 126$ pasos.

✍ En cambio debe usarse (en este caso) un único lazo, ya sea

```
for x in range(1 + 410 // 50):
    y = (410 - 50 * x) // 30
    if 50 * x + 30 * y == 410:
        ...
```
(17.1)

que realiza 9 pasos, o bien

```
for y in range(1 + 410 // 30):
```

[1] Bah, ¡a lo bestia!

```
    x = (410 - 30 * y) // 50
    ...
```

que realiza 14 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 + 410 // 50):
    y = int((410 - 50 * x) / 30)
    ...
```

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 + 410 // 50):
    y = (410 - 50 * x) / 30
    if y == int(y):  # si y es entero
        ...
```

ahorrando la verificación `50 * x + 30 * y == 410`.

Esta versión puede no dar resultados correctos si y no es entero pero `y == 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 17.1 es recorrer todas las posibilidades, una por una, y por eso se llama de *barrido*. La ecuación que aparece en ese ejercicio, $50x + 30y = 410$, es lineal, y como en el caso del máximo común divisor y el mínimo común múltiplo, hay técnicas mucho más eficientes para resolver este tipo de ecuaciones. Estas técnicas son variantes del 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.

Ejercicio 17.2. 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 17.1 (y en particular el esquema (17.1)), acá habrá que hacer dos lazos `for` *pero no tres*.

✍ Se pretende 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 12.1. ¶

También la técnica de «barrido» puede usarse para resolver ecuaciones diofánticas no lineales.

Ejercicio 17.3. 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 17.1 alrededor del esquema (17.1), *sin usar raíces cuadradas* (o algún sustituto) acá 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:
        ...
```

❦ *Lagrange demostró en 1770 que todo entero positivo es suma de cuatro cuadrados (algunos eventualmente nulos), pero en 1640 Fermat había demostrado que un entero positivo es suma de dos cuadrados si y sólo los factores primos impares de n que tienen resto* 3 *al dividirlos por* 4 *aparecen a una potencia par. Por ejemplo,* $1350 = 2\,3^3\,5^2$ *no puede escribirse como suma de cuadrados y* $1377 = 3^4\,17$ *sí (e. g.,* $9^2 + 36^2$*).* ¶

17.2. Cribas

Una *criba* (o *cedazo* o *tamiz*) es una selección de los elementos de en una lista que satisfacen cierto criterio que depende de los elementos precedentes, y en cierta forma es una variante de la técnica de barrido.

Quizás la más conocida de las cribas sea la atribuida a Eratóstenes 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 (pág. 81), para nosotros un número entero p es primo si $1 < p$ y sus únicos divisores positivos son 1 y p.

❦ *Eratóstenes (276 a. C.–194 a. C.) fue el primero en medir con una buena aproximación la circunferencia de la Tierra. ¡Y pensar que Colón usaba un huevo 1700 años después!*

Ejercicio 17.4 (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

$$2 \quad 3 \quad 4 \quad \ldots \quad n.$$

Recuadramos 2, y tachamos los restantes múltiplos de 2 de la lista, que no pueden ser primos, quedando

$$\boxed{2} \quad 3 \quad \cancel{4} \quad 5 \quad \cancel{6} \quad \ldots$$

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 recuadramos, y tachamos de la lista todos los múltiplos de 3 que quedan sin marcar. La lista ahora es

$$\boxed{2} \quad \boxed{3} \quad \cancel{4} \quad 5 \quad \cancel{6} \quad 7 \quad \cancel{8} \quad \cancel{9} \quad \cancel{10} \quad 11 \quad \cancel{12} \quad \ldots$$

y seguimos de esta forma, recuadrando y tachando múltiplos hasta agotar la lista.

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.

✍ Como ya observamos, si $a \times b = m$ y $\sqrt{m} < b < m$, debe ser $1 < a < \sqrt{m}$.

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 $\lfloor a \rfloor$.

También podemos cambiar el lazo en `j` por:

```
for j in range(i * i, n+1, i):
```

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 $n \log(\log n)$ pasos, 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 (ver el ejercicio 17.13). Es decir, el trabajo que realiza la criba es prácticamente lineal con respecto a la cantidad de elementos que encuentra. ¶

Ejercicio 17.5. 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 $i = 1, 2, \ldots, n$, girar media vuelta la llave de las celdas $i, 2i, 3i, \ldots$*

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 $1, 4, 9, \ldots$ que no superan n.* ¶

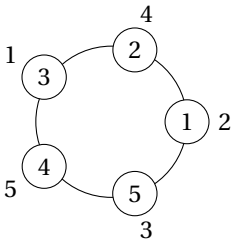


Figura 17.1: Esquema del problema de Flavio Josefo con $n = 5$ y $m = 3$.

Ejercicio 17.6 (el problema de Flavio Josefo I). 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.

Para analizar el problema —y en vez de ser tan crueles y matar personas— supondremos que tenemos inicialmente un círculo de n jugadores, numerados de 1 a n, 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.

Por ejemplo si $n = 5$ y $m = 3$, saldrán sucesivamente los jugadores numerados 3, 1, 5, 2, quedando al final sólo el número 4, como ilustramos en la figura 17.1 (con el sentido antihorario), donde los números fuera del círculo indican en qué momento salieron: el n-ésimo jugador que sale es el «sobreviviente».

Podemos estudiar el problema implementando una criba, considerando una lista `salio`, de longitud `n + 1` (el índice 0 no se usará), de modo que al terminar, `salio[j]` indicará la «vuelta» en la que salió el

jugador `j`.

Inicialmente podemos poner todas las entradas de `salio` en -1 para indicar que nadie salió, poniendo en 0 la entrada en la posición 0. Quedaría un esquema como el siguiente:

```
salio = [-1 for k in range(n+1)]    # nadie salió
salio[0] = 0                # no usar esta posición
s = n                # señala posición del que sale
for vuelta in range(1, n + 1):      # n vueltas
    cuenta = 0              # contamos m jugadores
    while cuenta < m:
        if s < n:       # incrementar 1
            s = s + 1
        else:           # empezar con 1
            s = 1
        if salio[s] < 0:            # si no salió
            cuenta = cuenta + 1     # contarlo
    salio[s] = vuelta       # la vuelta en que salió
```
(17.2)

a) Construir una función `josefo1` que toma como argumentos a los enteros positivos `n` y `m`, y retorna la lista `salio`, indicada anteriormente.

En el ejemplo con $n = 5$ y $m = 3$, se debe retornar `[0, 2, 4, 1, 5, 3]`.

b) La *permutación de Flavio Josefo* es el orden en que van saliendo, incluyendo el que queda al final. Para $n = 5$ y $m = 3$, el orden es 3, 1, 5, 2, 4.

Modificar la función del apartado anterior de modo que retorne las tupla `(salio, flavio)`, con `flavio[0] = 0`.

Es decir, para $n = 5$ y $m = 3$ se debe retornar `([0, 2, 4, 1, 5, 3], [0, 3, 1, 5, 2, 4])`.

✍ `flavio` es la permutación inversa de `salio`: `flavio[i]` es `j` $\Leftrightarrow$ `salio[j]` es `i`.

c) ¿Qué posición ocupaba en el círculo inicial Flavio Josefo (en la versión original del problema)?

d) ¿Cuál es el sobreviviente si $n = 1234$ y $m = 3$?

e) 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`.

f) Modificar la función de modo de imprimir una tabla con la posición inicial y el orden de eliminación, algo como

```
Posición inicial   Vuelta
              1         2
              2         4
              3         1
              4         5
              5         3
```

cuando $n = 5$ y $m = 3$. ¶

Ejercicio 17.7 (el problema de Flavio Josefo II). La criba según el esquema (17.2) es bastante ineficiente:

- Cuando m es grande comparado con la cantidad de jugadores que van quedando, estamos contando muchas veces el mismo jugador (por ejemplo si $n = 3$ y $m = 100$).

  Podríamos mejorar esto considerando sólo restos, es decir, usando aritmética modular con la función `%`.

  Como los restos al dividir por $k \in \mathbb{N}$ están entre 0 y $k - 1$, es conveniente considerar los índices de las listas a partir de 0.
- Una vez que han salido varios jugadores, tenemos que pasar muchas veces por el mismo jugador que ya no está para contar. Lo mejor sería sacarlo efectivamente de la lista.

Así, podríamos considerar una lista `quedan` de los que... ¡quedan!, inicialmente con los valores $0, \ldots, n - 1$ (en vez de entre 1 y n). A medida que sacamos jugadores de `quedan` formamos las listas `salio` y `flavio` de antes.

Tendríamos algo como:

```
quedan = list(range(n)) # entre 0 y n-1
nq = n          # la cantidad en quedan
salio = [-1 for x in range(n)]
flavio = []
s = -1
for vuelta in range(n): # n vueltas
    s = (s + m) % nq
    sale = quedan.pop(s)
    nq = nq - 1     # queda uno menos
    flavio.append(sale)
    salio[sale] = vuelta
    s = s - 1   # contar a partir de acá
```

Definir una función con este esquema que tome como argumentos `m` y `n` y que retorne la tupla `(salio, flavio)`. En el caso $n = 5$ y $m = 3$, se debe retornar `([1, 3, 0, 4, 2], [2, 0, 4, 1, 3])` pues numeramos entre 0 y $n - 1$.

✍ En mi máquina, el nuevo esquema es unas 5 veces más rápido que el (17.2) para $n = 1234$ y $m = 3$. ¶

17.3. Divisibilidad y números primos

Los números primos son considerados como ladrillos para la construcción de los enteros, pero en más de un sentido se sabe poco de ellos.

Quizás sea justamente por la cantidad de preguntas sencillas que no se han podido responder que estos números han fascinado a los matemáticos desde época remotas.

Por otra parte, con el auge de internet el estudio de los números primos dejó de ser un juego de matemáticos para convertirse en un tema de aplicación candente: la criptografía ha aprovechado la enorme dificultad computacional que es determinar los factores primos de un

entero para, por ejemplo, enviar datos bancarios por internet.

En esta sección veremos algoritmos elementales (¡e ineficientes!) para el estudio y resolución de algunas de estas preguntas.

Comenzamos con divisibilidad y la factorización de enteros.

Ejercicio 17.8 (divisores). En este ejercicio queremos definir una función `divisores(n)` que dado el número natural n encuentra la lista de los divisores positivos de n, es decir los enteros m, $1 \leq m \leq n$ tales que $m \mid n$. Por ejemplo:

```
>>> divisores(6)
[1, 2, 3, 6]
```

a) Una posibilidad es considerar un lazo `for` recorriendo todos los enteros entre 1 y n, viendo en cada caso si es divisor o no.

b) La anterior es bastante ineficiente, pues podríamos analizar sólo números que no superan $\sqrt{n}$: si $a, b \in \mathbb{N}$ y $ab = n$ entonces $a \leq \sqrt{n}$ o $b \leq \sqrt{n}$. Cambiar el lazo anterior usando esta propiedad. ¶

Ejercicio 17.9 (números perfectos). Un número es *perfecto* si es suma de sus divisores propios (menores que él). Por ejemplo, 6 es perfecto pues $6 = 1 + 2 + 3$.

Definir una función `esperfecto(n)` que retorna verdadero o falso según si n es perfecto o no, y luego encontrar los números perfectos que no superan 1000. ¶

Ejercicio 17.10. No es difícil ver que el entero $a > 1$ *no* es primo si y sólo si existen enteros b y c tales que $a = b \times c$ y $1 < b \leq \sqrt{a}$.

En base este resultado, definir una función `esprimo(n)` que decide si el número entero positivo n es primo o no.

Nota: la función no debería usar más de $\sqrt{n}$ pasos.

Ayuda: ver el ejercicio 17.8. ¶

Ejercicio 17.11. En este ejercicio queremos definir una función que dado $n \in \mathbb{N}$ lo descomponga en sus factores primos, contando la multiplicidad, retornando una lista de listas.

Por ejemplo:

```
>>> factores(20)
[[2, 2], [5, 1]]
>>> factores(21)
[[3, 1], [7, 1]]
```

a) Definir una función `verificar` para verificar los resultados obtenidos. Por ejemplo:

```
>>> verificar([[3, 2], [3607, 1], [3803, 1]])
123456789
```

b) Definir la función `factores` de dos formas distintas:

i) Usando la criba de Eratóstenes, encontrando los primos que no superan $\sqrt{n}$ y que dividen a n.

ii) Usando un esquema del tipo:

```
...
factores = []
d = 2       # divisor posible
while True:
    s = int(math.sqrt(n))
    while ((n % d) != 0) and (d <= s):
        d = d + 1
    if (d > s):  # acá n es 1 o es primo
        if n > 1:
            factores.append([n, 1])
        break
    # acá b divide a n y d <= s
    m = 1   # la multiplicidad
    n = n // d  # n se modifica
    while n % d == 0:
        n = n // d
        m = m + 1
    factores.append([d, m])
...
```

Probar ambas versiones con distintas entradas, como 1, 2, 4, 123456789, 1234567891, 1234567891234567891.

c) ¿Cuál de los dos métodos te parece más eficiente (realiza menos pasos)? ¶

Para muchos matemáticos, el tesoro al final del arcoíris es una fórmula para obtener todos los números primos.

En los próximos ejercicios, no completamente resueltos desde la teoría, muestran algunas de las aproximaciones humanas a ese tesoro.

Uno de los primeros intento de fórmula fue dada por Euclides:

Ejercicio 17.12 (primos de Euclides). Para demostrar que hay infinitos primos (en el libro IX de *Los Elementos*), Euclides supone que hay un número finito de ellos, digamos $(p_1, p_2, \ldots, p_n)$, y considera el número

$$x = p_1 \times p_2 \times \cdots \times p_n + 1, \tag{17.3}$$

de donde deduce que ninguno de los p_i, $i = 1, \ldots, n$ divide a x, y por lo tanto debe ser primo. Pero $x > p_i$ para todo i, por lo tanto llegamos a un absurdo pues x no está en la lista de los «finitos» primos.

Decimos que x es un *primo de Euclides* si es de la forma dada en la ecuación (17.3), donde $p_1, \ldots, p_n$ son los primeros n primos. Por ejemplo, los primeros primos de Euclides son $3 = 2 + 1$, $7 = 2 \times 3 + 1$ y $31 = 2 \times 3 \times 5 + 1$.

Sin embargo, ni todos los primos son de esta forma (como 5), ni todos los números de esta forma son primos. Por ejemplo

$$9\,699\,691 = 2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 \times 19 + 1 = 347 \times 27953.$$

Encontrar todos los números menores que 9 699 691 que son de la forma (17.3) y que *no* son primos de Euclides.

❦ *No se sabe si hay infinitos primos de Euclides.* ¶

A falta de fórmula, se ha intentado estudiar los primos como si «fueran al azar», mirando «estadísticamente» cómo se reparten los primos en la recta.

Ejercicio 17.13 (distribución de los números primos).

a) El *teorema de los números primos* establece que, a medida que n crece, el número de primos menores o iguales que n, indicado por $\pi(n)$, se aproxima a $n/\log n$.

Comprobarlo usando la función `criba` con $n = 10^k$ con $k = 4, 5, 6$.

Sugerencia: no es necesario usar `criba` para cada uno de estos valores, bastará tomar el mayor n y luego filtrar adecuadamente.

✍ La aproximación es muy lenta: para $n = 10^{20}$ el cociente $\pi(n)/(n \log n)$ es 1.021 aproximadamente.

b) Usando (o modificando) la criba de Eratóstenes, determinar cuántos de los primos que no superan 100 000 terminan respectivamente en 1, 3, 7 y 9.

✍ Observar que, al menos en el rango considerado, la distribución es muy pareja. Asintóticamente son iguales, según los resultados de Mertens que mencionamos al final del este capítulo. ¶

A contrapelo del estudio «estadístico» que sugiere que los primos están a distancias regulares, el próximo problema muestra que podría no ser así.

Ejercicio 17.14 (primos gemelos). Los primos p y q se dicen *gemelos* si difieren en 2. Es claro que excepto 2 y 3, la diferencia entre dos primos consecutivos debe ser 2 o más. Los primeros primos gemelos son los pares $(3, 5)$, $(5, 7)$ y $(11, 13)$, y el par más grande conocido[(2)] está formado por

$$3\,756\,801\,695\,685^{2\,666\,669} \pm 1,$$

(cada uno con 200 700 cifras decimales), pero no se sabe si hay infinitos pares de primos gemelos.

[(2)] Julio de 2013, `http://primes.utm.edu/top20/page.php?id=1#records`

Hacer una función para encontrar todos los pares de primos gemelos menores que n, y ver que hay 35 pares de primos gemelos menores que 1000.

✍ Si usamos listas con filtros, una primera posibilidad es usar un filtro del tipo

```
[... p in criba(n - 2)
 if p + 2 in criba(n - 2)]
```
(†)

Como no queremos calcular más de una vez `criba(n - 2)`, mejor es hacer

```
primos = criba(n - 2)
[... p in primos if p + 2 in primos]
```
(‡)

Esta última posibilidad tiene el inconveniente que recorremos toda la lista para cada `p`, haciendo el algoritmo ineficiente. Más razonable es mirar directamente a los índices:

```
p = criba(n - 2)
[... i in range(len(p) - 1)
    if p[i + 1] == p[i] + 2]
```
(∗)

Para $n = 1000$, en mi máquina el último esquema es el doble de rápido que el esquema (‡) y unas 160 veces más rápido que el esquema (†). Para $n = 10\,000$ en cambio, (∗) es 8 veces más rápido que (‡) y 1200 veces más rápido que (†). ¶

Los números primos surgen del intento de expresar un número como producto de otros, pero podemos pensar en otras descomposiciones. Por ejemplo, tomando sumas en vez de productos, como en el siguiente ejercicio.

Ejercicio 17.15 (conjetura de Goldbach). La conjetura de Goldbach, originada en la correspondencia entre Christian Goldbach y Euler en 1742, dice que todo número par mayor que 4 puede escribirse como la suma de dos números primos impares (no necesariamente distintos).

a) Definir una función que dado el número n par, $n \geq 6$, lo descomponga en suma de dos primos impares, exhibiéndolos (verificando la conjetura) o en su defecto diga que no se puede descomponer así.

b) Una variante de la conjetura, llamda *conjetura débil de Goldbach*, es que todo número impar mayor a 7 puede escribirse como suma de tres primos impares. Definir una función (que podría usar la anterior) para verificar que todo número n impar, $n \geq 9$, puede escribirse como suma de tres primos impares.

✍ Como se puede apreciar en el apartado *b*), la conjetura original implica esta versión.

c) La *cuenta de Goldbach* es la cantidad de formas en las que un número par puede escribirse como suma de dos primos impares. Por ejemplo, $20 = 13 + 7 = 17 + 3$, por lo que la cuenta de Goldbach de 20 es 2.

Hacer una función para determinar la cuenta de Goldbach de un entero par mayor que 4, y luego encontrar el máximo y mínimo de la cuenta de Goldbach de todos los pares entre 6 y 10 000. ¶

17.4. Ejercicios adicionales

Ejercicio 17.16 (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 `ifwhile.cifras`), la parte decimal se va generando de izquierda a derecha (también por divisiones sucesivas).

De la escuela recordamos que todo número racional p/q ($0 < p < q$) tiene una «expresión decimal periódica». Por ejemplo,

- $1/7 = 0.1428571428\ldots = 0.\overline{142857}$ tiene período 142857,
- $617/4950 = 0.124646\ldots = 0.12\overline{46}$ tiene período 46 y anteperíodo 12,
- $1/4 = 0.25 = 0.2500\ldots = 0.25\overline{0}$ 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. Asociar cada resto r con el cociente c que produce al hacer la división $r \times 10 = c \times q + r'$, donde r' es el nuevo resto.

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? ¶

Ejercicio 17.17 (La función ϕ de Euler). Para $n \in \mathbb{N}$, la *función ϕ de Euler* se define como la cantidad de coprimos positivos menores que n, es decir,

$$\phi(n) = |\{m \in \mathbb{N} : 1 \leq m \leq n \text{ y } \operatorname{mcd}(m, n) = 1\}|.$$

a) Probar que $n \in \mathbb{N}$ es primo $\Leftrightarrow \phi(n) = n - 1$.

b) Definir una función que dado $n \in \mathbb{N}$ calcule $\phi(n)$.
Sugerencia: Usar un filtro con el máximo común divisor.

❦ *Hay otros métodos más eficientes para calcular ϕ basados en propiedades que no veremos en el curso. Sin embargo, el cálculo de ϕ es tan difícil computacionalmente como la factorización de enteros.*

c) Calcular $\phi(n)$ para varios valores de n, y después conjeturar y demostrar para cuáles valores de n $\phi(n)$ es par.
Sugerencia para la demostración: $\mathrm{mcd}(n, k) = 1 \Leftrightarrow \mathrm{mcd}(n - k, n) = 1$.

d) Un primo p tal que $\phi(p)$ es una potencia de 2 se llama *primo de Fermat*, y debe ser de la forma $p = 2^{2^k} + 1$.
Los únicos primos de Fermat que se conocen son 3 ($k = 0$), 5, 17, 257 y 65537 ($k = 4$). Ver que para $k = 5$, $n = 2^{2^k} + 1$ no es primo.

❦ *Gauss demostró que se puede construir con regla y compás un polígono regular de n lados si $\phi(n)$ es una potencia de 2, esto es, si n es el producto de una potencia de 2 y primos de Fermat. Pierre Wantzel demostró en 1837 que la condición también es necesaria.* ❡

17.5. Comentarios

- Entre las ecuaciones diofánticas no lineales más interesantes están las de la forma $x^n + y^n = z^n$. Cuando $n = 2$, las soluciones forman una *terna pitagórica* y están completamente caracterizadas. 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!

- Como ya mencionamos, 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.

    En el ejercicio 17.11 vimos métodos para resolver estos problemas, que son *extremadamente ineficientes* para números grandes (200 o más cifras decimales).

  – La función ϕ de Euler que calculamos en el ejercicio 17.17 es un herramienta fundamental en el algoritmo de criptografía de R. Rivest, A. Shamir y L. Adleman (1978), uno de los más usados.

    Por supuesto, lo que hacemos acá es muy elemental, lejos de lo que se hace en la realidad. El ejercicio 11.9 da una idea de cosas que se pueden hacer para mejorar la eficiencia.

  – 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 los ejercicios 17.4 y 17.13, fue conjeturado por Gauss hacia 1792 o 1793, cuando tenía unos 15 años, y fue demostrado en 1896 independientemente por Jacques Hadamard (1865–1963) y Charles Jean de la Vallée-Poussin (1866–1962).

  – Johann Peter Gustav Lejeune Dirichlet (1805–1859) demostró en 1837 que toda progresión aritmética $a + bk$, $k = 1, 2, \ldots,$

contiene infinitos primos si $\mathrm{mcd}(a, b) = 1$. Este resultado implica que hay infinitos primos que terminan en 1 en su expresión decimal, infinitos que terminan en 3, etc., pero no dice cómo es la distribución.

El problema fue estudiado en forma más general por Franz Carl Joseph Mertens (1840–1927), quien refinó los resultados de Dirichlet. En particular, sus resultados implican que «hay tantos» primos que terminan en 1 como en 3, 7 o 9, como sugiere el ejercicio 17.13.*b*)

Un poco como contrapartida al teorema de Dirichlet, Ben Green y Terence Tao demostraron en 2004 que los números primos contienen progresiones aritméticas arbitrariamente largas.

– Hay muchas preguntas aún sin responder: no se sabe si hay infinitos primos de Euclides (ejercicio 17.12), o si hay infinitos primos gemelos (ejercicio 17.14), o si la conjetura de Goldbach es cierta (ejercicio 17.15).

Estos son temas de intenso estudio actual, y constantemente aparecen resultados.

Por ejemplo, si bien la conjetura de Goldbach no se ha demostrado, en mayo de 2013 H. Helfgott, demostró la versión débil (ejercicio 17.15.*b*)).

- Los ejercicios 17.5 y 17.6 están tomado de Engel (1993).

Capítulo 18

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).

18.1. 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 $1, 2, \ldots, n$, es decir, suponemos $V = \{1, 2, \ldots, n\}$. Más aún, casi siempre usaremos el nombre n (o algo que empiece con n) para el cardinal de V.

En los grafos que veremos, las aristas están formadas por un par de vértices, y como el orden no importará, indicaremos la arista que une el vértice a con el vértice b por $\{a, b\}$. Claro que decir $\{a, b\} \in E$ es lo mismo que decir $\{b, a\} \in E$.

✍ A veces se consideran grafos *dirigidos* o *digrafos*, en los que las aristas están orientadas, y por lo tanto se indican como (a, b) (y se llaman *arcos* en vez de aristas), distinguiendo entre (a, b) y (b, a). Nosotros no estudiaremos este tipo de grafos.

Así como n es el «nombre oficial» de $|V|$ (el cardinal de V), el «nombre oficial» para $|E|$ es m.

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*.

Sólo consideraremos grafos *simples* en los que no hay aristas uniendo un vértice con sí mismo, ni aristas «paralelas» o repetidas que unen los mismos pares de vértices. En este caso, 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 $m \le \binom{n}{2}$.

En la figura 18.1 mostramos un ejemplo de 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.

Siguiendo con la analogía de rutas, es común hablar de *camino*, una sucesión de vértices —el orden es importante— de la forma $(v_0, v_1, \ldots, v_k)$, donde $\{v_{i-1}, v_i\} \in E$ para $i = 1, \ldots, k$, y sin aristas repetidas (pero pueden haber vértices repetidos). Si $u = v_0$ y $v = v_k$, decimos que el camino $(v_0, v_1, \ldots, v_k)$ es un *camino de u a v*, o que

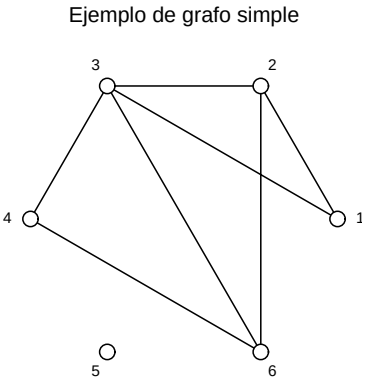


Figura 18.1: Un grafo con 6 vértices y 7 aristas. El vértice 5 es aislado.

une u y v, o sencillamente es un *camino u–v*. La *longitud* del camino $(v_0, v_1, \ldots, v_k)$ 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).

Por ejemplo, en la figura 18.1:

- $(3, 2, 6, 3, 4)$ 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 vértice. La relación « ~ » 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,[(1)] y las clases de equivalencia se llaman *componentes conexas* o simplemente *componen-*

[(1)] Ver el ejercicio 18.2.

tes del grafo. Por ejemplo, el grafo de la figura 18.1 no es conexo, pues tiene un vértice aislado, y sus componentes son $\{1, 2, 3, 4, 6\}$ y $\{5\}$.

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 $n^2/2$.[(2)]

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 creeremos):

18.1. Teorema (Caracterizaciones de árboles). *Dado un grafo simple $G = (V, E)$ con $|V| = n$, las siguientes condiciones son equivalentes:*

a) G es un árbol, es decir, es conexo y no tiene ciclos.

b) Para cualesquiera $a, b \in V$ existe *un* único *camino que los une.*

c) G es conexo y $|E| = n - 1$.

d) G no tiene ciclos y $|E| = n - 1$.

e) G es conexo, y si se agrega una arista entre dos vértices cualesquiera, se crea un único ciclo.

f) G es conexo, y si se quita cualquier arista queda no *conexo.*

A veces en un árbol consideramos un vértice particular como *raíz*, 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*

(2) Más precisamente, entre $n-1$ y $n(n-1)/2$.

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.

$G' = (V', E')$ es un *subgrafo* de $G = (V, E)$ si $V' \subset V$ y $E' \subset E$. Decimos que el subgrafo G' es *generador* de G si $V = V'$, y en particular, nos van a interesar *árboles generadores*. Si $G = (V, E)$ y $V' \subset V$, llamamos *subgrafo inducido* por V' al grafo $G' = (V', E')$ donde $E' = \{\{u, v\} \in E : u, v \in V'\}$.

Ejercicio 18.2. Definir una función `simple(uv, vw)` que dados los caminos u–v y v–w como listas de vértices, retorna un *camino simple* u–w. Por ejemplo, si uv es $[1, 2, 3, 4]$ y vw es $[4, 2, 5]$, un camino simple u–w es $[1, 2, 5]$.

Aclaración 1: suponemos u, v y w distintos entre sí.

Aclaración 2: los caminos u–v y v–w pueden no ser simples.

Aclaración 3: se pide *un* camino simple u–w aunque puede haber más de uno, por ejemplo si $uv = [1, 2, 3, 4, 5]$ y $vw = [5, 2, 4, 3]$, entonces $[1, 2, 3]$, $[1, 2, 4, 3]$ y $[1, 2, 5, 4, 3]$ son caminos simples 1–3. ¶

18.2. 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 $1, 2, \ldots, n$, y nos falta guardar la información de las aristas. Hay muchas formas de hacerlo, pero en este curso nos concentraremos en dos: dar la lista de aristas (con sus extremos) y dar 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.

Ejemplo 18.3. Para el grafo de la figura 18.1, podríamos poner

```
ngrafo = 6  # cantidad de vértices
aristas = [[1, 2], [1, 3], [2, 3], [2, 6],
           [3, 4], [3, 6], [4, 6]]
```

Observar que los elementos de `aristas` son a su vez listas en las que el orden es importante para Python (pues $[1,2] \neq [2,1]$). Sin embargo, al tratarse de aristas de un grafo para nosotros serán iguales.

✍ Recordando lo hecho en la el capítulo 14 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 18.6.

La lista de vecinos tendrá índices desde 1, por lo que pondremos `None` en la posición 0 a fin de evitar errores (pero la lista de aristas tiene índices desde 0).

```
vecinos = [None,
           [2, 3], [1, 3, 6], [1, 2, 4, 6],
           [3, 6], [], [2, 3, 4]]
```

✍ Observar que hay información redundante en la 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). ¶

Ejercicio 18.4. Definir una función para que dada la lista de vecinos imprima, para cada vértice, los vértices que son adyacentes.

Por ejemplo, si la entrada es el grafo del ejemplo 18.1, 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
```

¶

Ejercicio 18.5. 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 18.3, 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 no hay vértices u con $ngrafo < u$.
- 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 9.6.*c*)). ¶

Ejercicio 18.6. 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 $b < v$). ¶

Ejercicio 18.7. 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.

Por ejemplo, para el grafo del ejemplo 18.3, el archivo tendría (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? ¶

Ejercicio 18.8. 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». Por ejemplo, en la figura 18.2 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 $\{1, \ldots, n\}$, 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. ¶

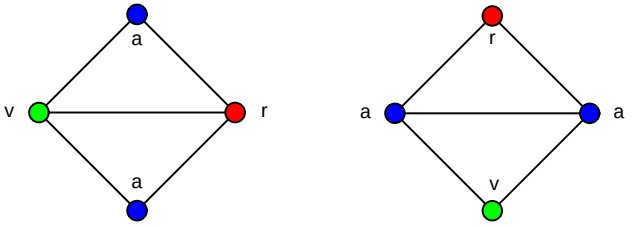


Figura 18.2: Asignando colores a los vértices de un grafo.

Ejercicio 18.9 (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 18.3, 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 18.1). ¶

Ejercicio 18.10. Es claro que si $(u = v_0, v_1, \ldots, v_k = v)$ es un camino u–v, entonces $(v_k, v_{k-1}, \ldots, v_0)$ es un camino v–u, y lo mismo para un ciclo.

Definir una función que ingresando un grafo y una sucesión de vértices $(v_0, v_1, \ldots, v_k)$, $k \geq 1$:

a) decida si $(v_0, v_1, \ldots, v_k)$ es un camino, es decir, si $\{v_{i-1}, v_i\} \in E$ para $i = 1, \ldots, k$, y no hay aristas repetidas,

✍ ¡Atención con $\{u, v\} = \{v, u\}$!

y en caso afirmativo:

b) imprima el camino inverso $v_k, \ldots, v_0$, y
c) verifique si $(v_0, v_1, \ldots, v_k)$ es un ciclo.

✍ Comparar con el ejercicio 18.2. ¶

Ejercicio 18.11. El módulo *grgrafo* (gráficos de grafos) permite hacer ilustraciones de grafos como el de la figura 18.1. El comportamiento es similar al del módulo *graficar* (en el capítulo 15), aunque una diferencia importante es que en *grgrafo* se pueden mover los vértices y etiquetas con el ratón.

Como otros módulos gráficos que vemos, *grgrsimple* es una plantilla para realizar ilustraciones, en este caso con *grgrafo*. Habrá que variar los parámetros del grafo (cantidad de vértices y aristas) y de la ilustración.

Ejecutar el módulo *grgrsimple* viendo su comportamiento moviendo los vértices y etiquetas, o ingresando otros grafos. ¶

18.3. 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 el primero y 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 a fin de no visitarlos nuevamente. Con este fin, normalmente usaremos una lista `padre` de modo que `padre[v]` nos dice desde qué vértice hemos venido a visitarlo. Para indicar el principio del recorrido, ponemos `padre[raiz] = raiz`,[3] y cualquier otro vértice tendrá `padre[v]` ≠ `v`.

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).

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 18.3 mostramos un esquema informal, donde la cola se llama Q y la raíz r.

Hay distintos tipos de cola:

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 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.

Las colas tienen ciertas operaciones comunes: inicializar, agregar un elemento y quitar un elemento:

- Inicializamos la cola con:

[3] Como siempre, no ponemos tildes en los identificadores para evitar problemas.

Algoritmo *recorrido*

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 18.3: Esquema del algoritmo `recorrido`.

```
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:

```
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.

Observar el uso de `padre` en la función `recorrido`. 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.

También podemos ver que la cola se ha implementado como pila pues sale el último en ingresar.

Ejercicio 18.12 (recorrido de un grafo).

a) Estudiar la función `recorrido` y comprobar el comportamiento para el grafo del ejemplo 18.3 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.
- ✍ En la página del libro hay una «película» (archivo pdf) de cómo se va construyendo el árbol cuando la raíz es 1. La figura 18.4 muestra la última página de ese archivo, donde podemos observar el árbol en azul, las flechas indican el sentido *hijo-padre*, y en las aristas está recuadrado el orden 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 $\{1, 2\}$).

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:
```

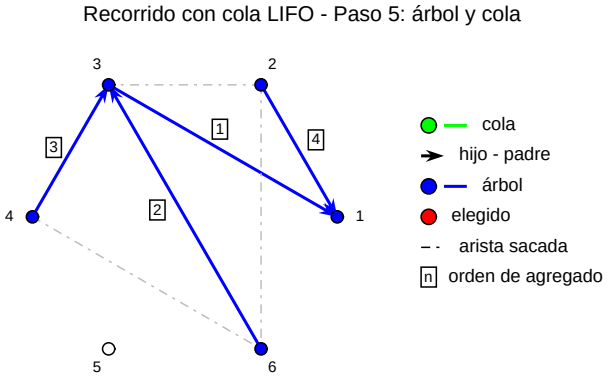


Figura 18.4: Recorrido *lifo* del grafo del ejemplo 18.3 tomando raíz 1.

```
padre[v] = u
```
? ¶

Ejercicio 18.13. 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 18.3 cuando la raíz es 1 se imprimiría

```
Orden de incorporación a la cola:
  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. ¶

Ejercicio 18.14 (componentes). En el ejercicio 18.12 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.

a) Agregar al grafo del ejemplo 18.3 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.

b) 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 18.3 el resultado debe ser algo como `[[1, 2, 3, 4, 6], [5]]`. ¶

En la figura 18.4 podemos ver que las aristas elegidas por la función `recorrido` forman un árbol. Usando la raíz 1 en el grafo del ejemplo 18.3, las aristas del árbol son $\{1,2\}$, $\{1,3\}$, $\{3,4\}$ y $\{3,6\}$, como ya mencionamos. En cambio, el árbol se reduce a la raíz cuando ésta es 5, y no hay aristas.

Ejercicio 18.15. Agregar instrucciones a la función `recorrido`, de modo que en vez de retornar los vértices del árbol obtenido, se retorne la lista de aristas que forman el árbol. ¶

Ejercicio 18.16. 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 `recorrido` con 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 18.1 y el ejercicio 18.14.*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: 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. ¶

Ejercicio 18.17 (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 18.9, 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 18.20).

✍ 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).

En el ejercicio 19.22 estudiamos *caminos de Hamilton*, que son similares y en algún sentido equivalentes a los ciclos.

❦ *Euler (1707–1783) fue uno de los más grandes y prolíficos matemáticos de todos los tiempos, haciendo contribuciones en todas las áreas de las matemáticas. Fue tan grande su influencia que se unificaron notaciones que él ideó, como la de* π *(=* $3.14159\ldots$*) (del griego* perypheria *o circunferencia),* i *(=* $\sqrt{-1}$*) (por imaginario), y* e *(=* $2.71828\ldots$*) (del alemán* einhalt *o unidad).*

Entre otras tantas, Euler inició el estudio de teoría de grafos y

la topología al resolver en 1736 el famoso problema de los puentes de Königsberg, demostrando el teorema que mencionamos en el ejercicio. ¶

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. No veremos esta variante ya que la programación es bastante más complicada e ineficiente.
- ✍ 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 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*.

Ejercicio 18.18 (recorrido a lo ancho).

a) Modificar la función `recorrido` de modo que la cola sea ahora *fifo*.

Sugerencia: cambiar `pop()` a `pop(0)` en el lugar adecuado.

b) Repetir el ejercicio 18.13, comparando las diferencias entre el recorrido en profundidad y el recorrido a lo ancho.

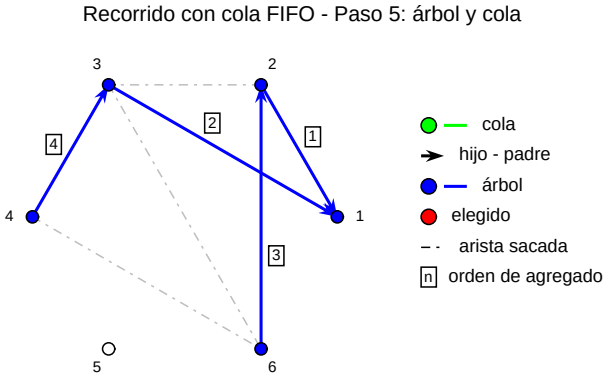


Figura 18.5: Recorrido *fifo* del grafo del ejemplo 18.3 tomando raíz 1.

✍ En la página del libro se muestra una «película» (archivo pdf) de cómo se va construyendo el árbol cuando la raíz es 1. La figura 18.5 es la última página de ese archivo, destacando el orden en que las aristas se incorporan. Comparar con la figura 18.4. ¶

Una forma dramática de mostrar las diferencias de recorridos lifo o fifo es construyendo laberintos como los de la figura 18.6.

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 el algoritmo `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 18.6 a la izquierda mostramos el árbol obtenido para cola lifo y a la derecha para cola fifo, para $m = n = 20$, usando una

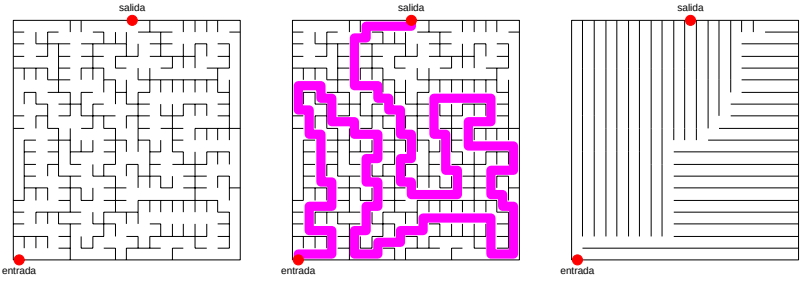


Figura 18.6: Laberinto (izquierda) y solución (centro) usando pila (lifo), y laberinto usando cola fifo (derecha).

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.

18.4. Ejercicios Adicionales

Ejercicio 18.19. Construir laberintos de $m \times n$ como los de la figura 18.6 (ver descripción correspondiente). ¶

Ejercicio 18.20 (ciclo de Euler II). En el ejercicio 18.17 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, debe haber un vértice en el ciclo construido que tenga una arista (aún no eliminada) incidente 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 18.17), 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.*b*), y para «pegar» ciclos podría usarse una construcción del tipo `ciclo1[:i-1] + ciclo2 + ciclo1[i+1:]`. ¶

18.5. Comentarios

- Las figuras de los laberintos (figura 18.6) se hicieron con el módulo *graficar*. Las restantes figuras en este capítulo se hicieron con el módulo *grgrafo*, así como las «películas» en la página del libro para ilustrar los distintos algoritmos.
- La presentación de la función `recorrido` y del algoritmo 18.3 están basadas en la de Papadimitriou y Steiglitz (1998).

Capítulo 19

Recursión

recursivo, va. 1. adj. ver recursivo.

19.1. Introducción

Una forma de sumar los números $a_0, a_1, a_2, \ldots$, es ir construyendo las sumas parciales que llamamos *acumuladas* en el ejercicio 10.11:

$$s_0 = a_0,\ s_1 = s_0 + a_1,\ s_2 = s_1 + a_2,\ \ldots\ ,\ s_n = s_{n-1} + a_n,\ \ldots,$$

ecuaciones que pueden expresarse más sencillamente como

$$s_0 = a_0 \qquad \text{y} \qquad s_n = s_{n-1} + a_n \quad \text{para } n \geq 1. \tag{19.1}$$

Cuando la sucesión de números es $1, 2, 3, \ldots$ y cambiamos suma por producto, obtenemos el factorial (ejercicio 10.9):

$$1! = 1,\ 2! = 1! \times 2,\ 3! = 2! \times 3,\ \ldots\ ,\ n! = (n-1)! \times n,\ \ldots,$$

y en realidad es usual definir $n!$ mediante

$$0! = 1 \qquad \text{y} \qquad n! = n \times (n-1)! \quad \text{para } n \in \mathbb{N}. \tag{19.2}$$

Cuando se dan uno o más valores iniciales y una «fórmula» para calcular los valores subsiguientes como en (19.1) o (19.2), 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*[1] en programación, aunque los conceptos no son completamente equivalentes. Por ejemplo, la relación (19.2) 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, para calcular 4! primero calculamos 3!, para calcular 3! primero calculamos 2!, para calcular 2! primero calculamos 1!, y finalmente 1! = 1.

19.2. Funciones definidas recursivamente

Ejercicio 19.1. En Python podemos definir una función recursiva para calcular el factorial basada en la ecuación (19.2), 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.
```

[1] La palabra *recursión* no existe en castellano según la RAE.

```
    """
    if n == 1:      # paso base
        return 1
    return n * factorial(n-1)
```

a) Comparar esta función con la definida en el ejercicio 10.9 para $n = 1, 2, \ldots, 10$, usando alguna técnica del ejercicio 10.18 para compararlas.

b) 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.

c) ¿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.

Ejercicio 19.2. Definir una función recursiva para calcular las sumas de Gauss,

$$s_n = 1 + 2 + \cdots + n = \sum_{k=1}^{n} k,$$

luego definir otra función usando que

$$s_n = \frac{n \times (n+1)}{2},$$

y compararlas.

✍ Usar división entera para la segunda función.

❦ *Según se dice, Johann Carl Friederich Gauss (1777–1855) tenía 8 años cuando el maestro de la escuela le dio como tarea sumar los números de* 1 *a* 100 *para mantenerlo ocupado (y que no molestara). Sin embargo, hizo la suma muy rápidamente al observar que la suma era* 50×101.

> *Las contribuciones de Gauss van mucho más allá de esta anécdota, sus trabajos son tan profundos y en tantas ramas de las matemáticas (y la física) que le valieron el apodo de «príncipe de los matemáticos». Para algunos, fue el más grande matemático de todos los tiempos.* ¶

Las funciones recursivas pueden tener uno o varios argumentos, y no tienen que 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.

Ejercicio 19.3. 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 `cociente(a, b)` $\leftrightarrow$ `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. ¶

Ejercicio 19.4. Dar una definición recursiva de la función `saldo(c, r, p, m)` en el ejercicio 16.22.*c*) según las ecuaciones (16.9). ¶

Ejercicio 19.5. 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 $\mathrm{mcd}(a, b)$ cuando a y b son enteros positivos.

Ayuda: si $a > b$ entonces $\mathrm{mcd}(a, b) = \mathrm{mcd}(a-b, b)$, análogamente para el caso $a < b$, y $\mathrm{mcd}(a, a) = a$.

✍ En este ejercicio la recursión se hace en ambos argumentos, a diferencia del ejercicio 19.3.

✍ Observar que no se hace la «llamada a `f(n - 1)`». ¶

19.3. Números de Fibonacci

Los números de Fibonacci f_n se definen mediante:

$$f_1 = 1, \quad f_2 = 1, \quad \text{y} \quad f_n = f_{n-1} + f_{n-2} \quad \text{para } n \geq 3, \tag{19.3}$$

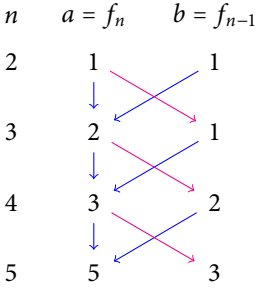


Figura 19.1: Ilustración del cálculo de los números de Fibonacci.

obteniendo la sucesión $1, 1, 2, 3, 5, 8, \ldots$:

$$1,\ 1,\ 2\,(=1+1),\ 3\,(=2+1),\ 5\,(=3+2),\ 8\,(=5+3), \ldots$$

En la figura 19.1 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).

Ejercicio 19.6.

a) Como para el cálculo de f_n con $n \geq 3$ sólo necesitamos conocer los dos valores anteriores, 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 19.1.

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.

b) 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 (19.3).

✍ Observar que en la versión recursiva se hacen dos llamadas a la misma función.

c) Comparar los resultados de ambas versiones para $n \in \mathbb{N}$, $n \leq 10$.

d) ¿Qué pasa con la versión recursiva si $n < 1$?, ¿es correcto el resultado?, ¿y en la versión con `for`?

❦ *Leonardo Bigollo (1170–1250) fue conocido como Leonardo de Pisa y también como Fibonacci (*hijo de Bonacci*). En el libro «Liber Abaci» propuso el famoso problema con los conejos donde aparecen por primera vez los números que luego llevaron su nombre.*

A pesar de la similitud de los nombres, Leonardo da Vinci (1452–1519) es muy posterior a Leonardo de Pisa. ❡

Los números de Fibonacci están íntimamente relacionados con el *número de oro*, también llamado *proporción áurea*,

$$\tau = \frac{1+\sqrt{5}}{2} = 1.61803\ldots, \tag{19.4}$$

que aparece muchas veces en matemáticas, las artes y la arquitectura.

τ es solución de la ecuación

$$x^2 = x + 1,$$

cuya otra solución es

$$\tau' = \frac{1-\sqrt{5}}{2} = -\tau^{-1}.$$

De la teoría general de relaciones de recurrencia, se ve que

$$f_n = \frac{\tau^n - (\tau')^n}{\sqrt{5}} = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}, \tag{19.5}$$

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 (19.5) es un número entero!

❦ *Jacques Binet (1786–1856) publicó la ecuación (19.5) para los números de Fibonacci en 1843. Leonhard Euler (1707–1783) la había publicado en 1765, y de ahí el nombre de la relación. Sin embargo, A. de Moivre (1667–1754) ya la había publicado en 1730, ¡y en forma más general!*

Ejercicio 19.7 (fórmula de Euler-Binet).

a) Definir una función `binet` para calcular f_n según (19.5).
Sugerencia: usar división común y no entera.

b) Comparar los valores de `binet` con los de la versión recursiva definida en el ejercicio 19.6.*b*) para $n = 1, \ldots, 10$.

c) Comparar el n-ésimo número de Fibonacci f_n con el redondeo (`round`) de

$$\frac{\tau^n}{\sqrt{5}} = \frac{(1+\sqrt{5})^n}{2^n\sqrt{5}}. \tag{19.6}$$

¿A partir de qué n son iguales?, ¿podrías decir por qué?

d) Construir una función `binet2` para calcular f_n según (19.6).

e) Calcular f_{345} de dos formas distintas: con la función definida en el ejercicio 19.6.*a*) 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 `math.sqrt` y otras operaciones con decimales. ❡

Ejercicio 19.8 (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

$$n = b_2 f_2 + b_3 f_3 + \cdots + b_m f_m, \tag{19.7}$$

donde $b_m = 1$, $b_k \in \{0, 1\}$, y $b_k \times b_{k+1} = 0$ para $k = 2, \ldots, m-1$.

Por ejemplo, $10 = 8 + 2 = f_6 + f_3$, $27 = 21 + 5 + 1 = f_8 + f_5 + f_2$.

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 (19.7)). Comprobar con algunos ejemplos la veracidad de la salida. ¶

19.4. Ventajas y desventajas de la recursión

Expliquemos un poco cómo funciona recursión.

Como sabemos, una función ocupa un lugar en la memoria al momento de ejecución, conceptualmente llamado *marco* o *espacio* o *contexto* de la función. Ese marco 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 marco propio. Cuando termina su tarea, la copia se borra y el espacio 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.

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 19.6.*a*)), 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:

$$
\begin{aligned}
f_5 &= f_4 + f_3 \\
f_4 &= f_3 + f_2 \\
f_3 &= f_2 + f_1 \\
f_2 &\to 1 \\
f_1 &\to 1 \\
f_2 &\to 1 \\
f_3 &= f_2 + f_1 \\
f_2 &\to 1 \\
f_1 &\to 1
\end{aligned}
$$

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 19.6, 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.

Ejercicio 19.9. 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. 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 19.23 vemos lo absurdo de usar recursión para calcular los números de Fibonacci. ¶

19.5. 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.

Ejercicio 19.10 (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 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 realizar esta transferencia, mostrando los pasos realizados.

Supongamos 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 al 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 *recursion2*), 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) 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?

c) ¿Cuántos años tardaría una computadora en calcular la solución para $n = 64$, suponiendo que tarda un nanosegundo (ns) por movimiento[(2)] (nano = dividir por mil millones)?

[(2)] ¡Y que no hay cortes de luz!

✍ 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 .

d) 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?

e) Modificar la función `hanoi` de modo que las «agujas» a, b y c se representen por 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

$$a = [n, n-1, \ldots, 1], \quad b = [\,], \quad c = [\,],$$

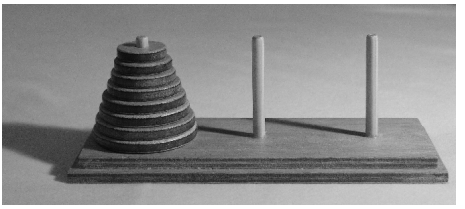
y al terminar debe ser

$$a = [\,], \quad b = [n, n-1, \ldots, 1], \quad c = [\,].$$

Por ejemplo:

```
>>> hanoi(3)
Paso 0
    a: [3, 2, 1]
    b: []
    c: []
Paso 1
    a: [3, 2]
    b: [1]
    c: []
Paso 2
    a: [3]
```

Tapa del juego original



Fotografía del juego

Figura 19.2: Las torres de Hanoi.

```
    b: [1]
    c: [2]
...
```

Sugerencia: trabajar con las listas como pilas.

✍ 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.

✍ 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 imágenes de la figura 19.2 fueron tomadas respectivamente de[3]

- `http://www.cs.wm.edu/~pkstoc/`
- `http://en.wikipedia.org/wiki/Tower_of_Hanoi`.

❦ *«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 19.2, 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

[3] Enlaces válidos a febrero de 2013.

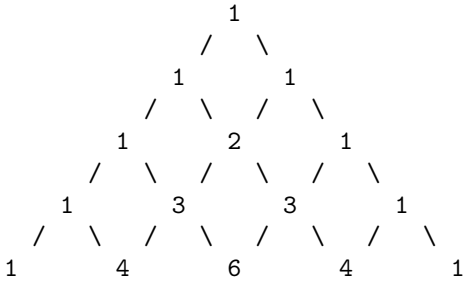


Figura 19.3: Triángulo de Pascal de nivel 4.

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. ¶

19.6. Contando objetos combinatorios

Es posible dar una definición inductiva de los coeficientes binomiales, alternativa de la que vimos en (10.10):

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \text{y} \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n. \tag{19.8}$$

Recordemos que $\binom{n}{k}$ representa la cantidad de subconjuntos con exactamente k elementos si el total tiene n. Así, si $I_n = \{1, \ldots, n\}$, (19.8) 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 (19.8) da lugar al llamado *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

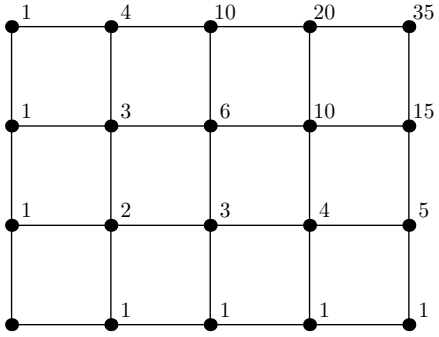


Figura 19.4: Contando la cantidad de caminos posibles.

fila anterior, empezando y terminando con 1, como se muestra en la figura 19.3.

Ejercicio 19.11. Definir una función recursiva para calcular $\binom{n}{k}$ en base a (19.8). ❡

Ejercicio 19.12. Para $m, n \in \mathbb{N}$, consideremos una cuadrícula rectangular de dimensiones $m \times n$ (4×3 en la figura 19.4), 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.

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 19.4

indican, para cada intersección, la cantidad de caminos para llegar allí 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 demuestra que $h(m,n)$ es el número combinatorio $\binom{m+n}{n}$, lo que podemos apreciar comparando el rectángulo de la figura 19.4 con el triángulo de Pascal de la figura 19.3.

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). ¶

Ejercicio 19.13. 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 $0 \le x < m$ y $0 \le y < n$). ¶

19.7. Las grandes listas: otro peligro de la recursión

Un problema muy distinto al de *contar* objetos, como hicimos con los caminos del ejercicio 19.12, 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 $I_n = \{1, \ldots, n\}$.

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 subconjuntosNO(n):
    """Lista de subconjuntos de {1,..., n}."""
    if n == 0:          # no hay elementos
        return [[]]     # sólo el conjunto vacío
    subs = subconjuntosNO(n-1)
    return subs + [s + [n] for s in subs]
```
(19.9)

✍ Aunque para conjuntos el orden no importa y no hay elementos repetidos, los representamos mediante listas como en la sección 14.2.

Veamos el porqué de la señal 💣 en esta construcción.

Ejercicio 19.14 (subconjuntos I). Probar la función en (19.9) 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 = subconjuntosNO(8)` y luego averiguar `len(s)`, o bien poner `len(subconjuntosNO(8))` directamente. ¶

El algoritmo para definir `subconjuntosNO` en (19.9) 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 querremos 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.

Ejercicio 19.15. 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 `subconjuntosNO(n)`.

a_n tiene 2^n listas, y eliminando los corchetes, en total tiene $n\,2^{n-1}$ números (¿por qué?).[4]

a) 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!

b) Si la memoria de la computadora que usamos tiene 4 gigabytes ($1\,\text{GB} = 10^9$ bytes), ¿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: $n = 23$.

✍ 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.

c) Es posible que Python no use exactamente 8 bytes por número, o que la máquina tenga más o menos memoria, por lo que el valor de n obtenido en el apartado anterior es sólo una estimación del lugar que ocupan los números.

Probar `subconjuntosNO(n)`, por ejemplo encontrando la cantidad de subconjuntos, comenzando desde $n = 18$ y luego 19, 20, etc., hasta que los tiempos sean muy largos,[5] 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

[4] *Sugerencia*: un conjunto y su complemento tienen n elementos entre los dos.

[5] 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.

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* `subconjuntosNO` *en* (19.9).

19.8. `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 19.10. 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)
```

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
```

(19.10)

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 19.6.

Estudiemos el comportamiento y propiedades de `fibgen`.

Ejercicio 19.16.

a) Definir la función `fibgen` según el esquema (19.10) en un módulo, y ejecutarlo.

b) `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))
```

c) 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(range(5))
list(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 (19.10), 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 19.16.*c*).

Ejercicio 19.17. 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 (19.10) de modo de obtener *todos* los números de Fibonacci (y no sólo n) con el iterador. ¶

19.9. Generando objetos combinatorios

Veamos cómo `yield` nos permite evitar la construcción de la «gran lista» del ejercicio 19.14.

Ejercicio 19.18 (subconjuntos II). Para construir los subconjuntos de I_n podemos considerar:

```
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
```

a) Poner la función en un módulo, ejecutarlo, y evaluar:

```
for x in subs(3):
    print(x)
```

Comparar el resultado cambiando `for s in subs(3)` por `for s in subconjuntosNO(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 19.16:

```
[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 9.13 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 19.18 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 `subconjuntosNO(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.

Ejercicio 19.19. Repetir el ejercicio 19.15.*c*) con `subs(n)` para $n = 18, 19, 20, \ldots$, 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 (19.8) 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 255):

```
...
for p in perms(n-1):
    for i in range(n):
        p.insert(i, n)
        yield p
        p.pop(i)
...
```

Ejercicio 19.20. Repetir los apartados del ejercicio 19.18, 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:* $[]$, $[1]$, $[1,2]$, $\ldots$, $[1,2,\ldots,k]$.

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.* ¶

Ejercicio 19.21.

a) Definir una función `cadenas(m, n)` para generar todas las listas de longitud n con los elementos de $\{0, \ldots, m-1\}$ ($m, n \in \mathbb{N}$). 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 (ver la sección 11.4). Definir una función para encontrar las listas del apartado anterior recorriendo los números entre 0 y $m^n - 1$ (inclusivos). ¶

Ejercicio 19.22 (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, ..., n:
    si p es camino del grafo:
        retornar p    # y salir
reportar 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 18.10.

✍ 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).

❦ *W. R. Hamilton (1805–1865) ideó un juego en el que había que recorrer los vértices de un dodecaedro con los nombres de capitales del mundo, y de allí el nombre del camino.*

Caminos y ciclos de Hamilton son usados en cosas tan diversas como criptografía o el «problema del viajante» en investigación operativa.

Hamilton es mucho más conocido por sus contribuciones fundamentales a la física (en electromagnetismo y mecánica cuántica) y matemáticas (cuaterniones en álgebra, principios variacionales y ecuaciones diferenciales). ¶

19.10. Ejercicios adicionales

Ejercicio 19.23.

a) Usando el ejercicio 19.9, 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! ¶

Ejercicio 19.24. A veces queremos ordenar 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, 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*

$$\binom{n_1 + n_2 + \cdots + n_k}{n_1, n_2, \ldots, n_k} = \frac{(n_1 + n_2 + \cdots + n_k)!}{n_1!\, n_2! \ldots n_k!}.$$

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). ¶

19.11. Comentarios

- En cursos de matemática discreta es usual estudiar las relaciones de recurrencia *lineales*, *homogéneas* y *con coeficientes constantes*,

$$a_n = A\, a_{n-1} + B\, a_{n-2} \quad \text{para } n > 1, \tag{19.11}$$

  donde A y B son constantes. Se ve que si la ecuación cuadrática *característica*, $x^2 = A\, x + B$, tiene dos raíces distintas r y r' (podrían ser complejas), entonces existen constantes c y c' de modo que los números a_n en (19.11) se pueden expresar como

$$a_n = c\, r^n + c'\, (r')^n.$$

  La fórmula de Binet (19.5) es un caso particular de esta relación.
- 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.

  En cuanto a aplicaciones teóricas, vale la pena mencionar que en el Congreso Internacional de Matemáticas de 1900, David Hilbert (1862–1943) propuso una serie de problemas que en gran parte determinaron 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 el problema es irresoluble, ¡usando números de Fibonacci!

- En este capítulo vimos que si bien recursión en general es ineficiente, se puede mejorar su desempeño con distintas técnicas.

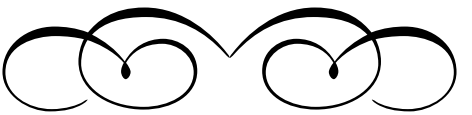
  También vimos que, de cualquier forma, siempre nos topamos con la denominada «explosión combinatoria»: la cantidad exponencial de objetos a examinar cuando usamos técnicas de barrido.

  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 (remedando los problemas planteados por Hilbert un siglo antes) se encuenctra justamente decidir si $P \neq NP$.
- Los generadores vía `yield` son un caso particular de *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 desciende también Pascal. A su vez, Modula-2 fue desarrollado a fines de los 70 por N. Wirth como sucesor «profesional» de Pascal, que es de un carácter más «pedagógico».
- 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

dados (ejercicios 13.4 y 13.5)

```
"""Simulaciones con dados usando random en Python."""

import random

def dado1():
    """Simular tirar un dado, obteniendo un entero entre 1 y 6."""
    return random.randint(1, 6)

def dado2(k):
    """Veces que se tira un dado hasta que aparece k.

    k debe ser entero, 1 <= k <= 6.

    """
    veces = 0
    while True:                         # repetir
        veces = veces + 1
        if random.randint(1, 6) == k:   # hasta obtener k
            break
```

```
    return veces
```

dearchivoaconsola (ejercicio 12.6)

```
"""Leer un archivo de datos e imprimirlo en la consola."""

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
```

decimales (capítulo 16)

```
"""Algunas propiedades de la representación decimal en Python.

- epsmaq: épsilon de máquina

- epsmin: épsilon mínimo (menor potencia de 2 que es positiva).

- maxpot2: máxima potencia de 2 que se puede representar.

"""

def epsilon(inic):
    """Menor potencia de 2 que sumada a inic da mayor que inic.

    - Suponemos 0 < inic < 1.0 + inic.
    - epsilon(1.0) es lo que llamamos epsmaq en los apuntes.

    """
    x = 1.0
    while inic + x > inic:
        x = x / 2
    return 2 * x    # volver al valor anterior
```

```
epsmaq = epsilon(1.0)

# cálculo de epsmin
x = 1.0
while x > 0:
    x, epsmin = x / 2, x

# cálculo de maxpot2
x = 1.0
while 2 * x > x:
    x, maxpot2 = 2 * x, x
```

eratostenes (ejercicio 17.4)

```
"""Versión sencilla de la criba de Eratóstenes.

Esta es una primera versión que debe mejorarse según los ejercicios
en el libro.

"""

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 16.7)

```
"""Complicaciones con Euclides usando decimales."""

# 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 7.15)

```
"""Función donde 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 7.14)

```
"""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)
```

gr1sobrex (ejercicio 15.6)

```
"""Graficar la función discontinua 1/x entre -1 y 1.

- La discontinuidad está en x = 0.
- Separamos en partes el dominio: antes y después de 0 con eps > 0.
- Si eps = 0 da error de división por 0.

"""

import graficar

# titulo de la ventana
graficar.titulo = 'Gráfico de 1/x'

# cotas
eps = 0.01 # distancia a separar, poner también 0

def f(x):
    return 1/x

# valores extremos para x
a = -1
b = -a  # intervalo simétrico

# valores extremos para y
ymax = 100
ymin = - ymax

graficar.ymax = ymax
graficar.ymin = ymin

# mismo color para ambos tramos
graficar.funcion(f, a, -eps, estilo={'fill': 'blue'})
graficar.funcion(f, eps, b, estilo={'fill': 'blue'})

# ejes coordenados
graficar.recta((0, 0), (1, 0), estilo={'fill': 'black'})
graficar.recta((0, 0), (0, 1), estilo={'fill': 'black'})
```

```
graficar.mostrar()
```

grafos (capítulo 18)

```
"""Funciones para grafos."""

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.

    Los índices para los vértices empiezan desde 1.

    """
    ngrafo = len(vecinos) + 1  # no usamos vecinos[0]
    aristas = []
    for v in range(1, ngrafo):
        for u in vecinos[v]:
            # 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.

    - La lista de vecinos debe ser de la forma
        [None, vecinos[1],..., vecinos[n]]

    - Se retornan 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]
```

grbiseccion (ejercicio 16.21)

```
"""Ilustración del método de la bisección.

Plantilla ilustrando el método de la bisección para
encontrar ceros de una función: cambiar apropiadamente.

Elegir (interactivamente) los extremos de un intervalo
que encierra al menos un cero de la función.

Se imprime genéricamente 'f' en los resultados por terminal.

"""

# módulos de Python
# import math   # si se usan funciones trascendentales

# módulos propios
import graficar, grnumerico
```

```
# función donde encontrar el cero
def f(x):
    return x * (x + 1) * (x + 2) * (x - 4/3)

# intervalo
a = -3
b = 2

# constante para el método
eps = 1.0e-7 # error permitido

# leyenda para la función en el gráfico
leyendaf = 'f'

# otras opciones gráficas
graficar.titulo = 'Ilustración del método de la bisección'
graficar.leyendaspos = 'N'

# limitar si no se ven bien positivos/negativos
graficar.ymin = -3
graficar.ymax = 5

# ejecución
grnumerico.biseccion(f, a, b, eps, leyendaf)
```

grexplog (ejercicio 15.3)

```
"""Gráficos de exp, log y la identidad."""

# módulos de Python
import math

# módulos propios
import graficar

graficar.titulo = 'Comparación de la exponencial y el logaritmo'
```

```
xmin = -2
xmax = 5
ymin = -5
ymax = xmax
graficar.xticks = list(range(xmin, xmax, 2))
graficar.yticks = list(range(ymin + 1, ymax, 2))

eps = 0.001 # log no está definida en 0

graficar.ymin = ymin
graficar.ymax = ymax

graficar.aspecto = False

graficar.funcion(math.exp, xmin, xmax, leyenda='exp(x)')
graficar.funcion(math.log, eps, xmax, leyenda='log x')

# identidad para comparar
def identidad(x):
   return x
graficar.funcion(identidad, xmin, xmax, leyenda='identidad')

# ejes coordenados en negro
graficar.recta((0, 0), (1, 0), estilo={'fill': 'black'})
graficar.recta((0, 0), (0, 1), estilo={'fill': 'black'})

graficar.leyendaspos = 'NO'

graficar.mostrar()
```

grgrsimple (ejercicio 18.11)

```
"""Ilustración del uso del módulo grgrafo para un grafo simple.

Recordar que se pueden mover los vértices y etiquetas del grafo
con el ratón.

"""
```

```
import grgrafo

#-------------------------------------------
# descripción del grafo, modificar a gusto
# ejemplo en los apuntes
ngrafo = 6
aristas = [[1, 2],
           [1, 3],
           [2, 3],
           [2, 6],
           [3, 4],
           [3, 6],
           [4, 6]]
#-------------------------------------------

grgrafo.titulo = 'Ejemplo de grafo en los apuntes'

for i in range(1, ngrafo + 1):
    grgrafo.vertice(texto=str(i))
for u, v in aristas:
    grgrafo.arista(grgrafo.vertices[u], grgrafo.vertices[v])

grgrafo.mostrar()
```

grnewton (ejercicio 16.18)

```
"""Ilustración del método de Newton.

Plantilla ilustrando el método de Newton para encontrar
ceros de una función: cambiar apropiadamente.

Elegir (interactivamente) el punto inicial.

La derivada se estima poniendo un incremento bien pequeño.

Se imprime genéricamente 'f' en los resultados por terminal.
```

```
"""

# módulos de Python
# import math # si se usan funciones trascendentales

# módulos propios
import graficar, grnumerico

# función donde encontrar el cero
def f(x):
    return (x + 1) * (x - 2) * (x - 3)

# intervalo
a = -2
b = 4

# constantes para el método
eps = 1.0e-7    # error permitido
maxit = 20      # máximo número de iteraciones

# leyenda para la función en el gráfico
leyendaf = 'f'

# otras opciones gráficas
graficar.titulo = 'Ilustración del método de Newton'

graficar.leyendaspos = 'SE'

# limitar si no se ven bien positivos/negativos
graficar.ymin = -20
graficar.ymax = 10

# ejecución
grnumerico.newton(f, a, b, eps, maxit, leyendaf)
```

grpuntofijo (ejercicio 16.14)

```
"""Ilustración de la técnica del punto fijo.
```

```
Plantilla ilustrando el método de punto fijo iterando una
función: cambiar apropiadamente.

Elegir (interactivamente) el punto inicial.

Se imprime genéricamente 'f' en los resultados por terminal.

"""

# módulos de Python
import math     # si se usan funciones trascendentes

# módulos propios
import graficar, grnumerico

# función donde encontrar el punto fijo
def f(x):
    return math.cos(x)

# intervalo
a = 0
b = 1.2

# constantes para el método
eps = 1.0e-7    # error permitido
maxit = 20      # máximo número de iteraciones

# leyenda para la función en el gráfico
leyendaf = 'cos'

# otras opciones gráficas
graficar.titulo = 'Ilustración del método de punto fijo'
graficar.leyendaspos = 'S'

# limitar si no se ven bien positivos/negativos
# graficar.ymin = -3
# graficar.ymax = 5
```

```
# ejecución
grnumerico.puntofijo(f, a, b, eps, maxit, leyendaf)
```

grseno (ejercicio 15.1)

```
"""Gráfico del seno entre 0 y pi."""

# módulos de Python
import math

# módulos propios
import graficar

graficar.funcion(math.sin, -math.pi, math.pi)

graficar.mostrar()
```

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 llamás?')
pepe = input()
print('Hola', pepe, 'encantada de conocerte')
```

ifwhile (capítulo 8 y ejercicio 9.4)

```
"""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 mcd2(a, b):
    """Máximo común divisor entre los enteros a y b.

    Versión usando divisiones enteras y restos.

    mcd2(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."""

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.

    'poco' y 'mucho' deben ser puntos encerrando un cero de 'func'.

    """

    # 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 19)

```
"""Ejemplos sencillos de funciones recursivas."""

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.

    Usamos la versión original de Euclides
    recursivamente.

    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
```

recursion2 (capítulo 19)

```
"""Funciones recursivas avanzadas."""

def hanoi(n):
    """Solución recursiva a las torres de Hanoi."""

    # 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')

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
```

santosvega.txt (ejercicio 12.6)

```
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.
```

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 12.5)

```
"""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()
```

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.

$\varnothing$ El *conjunto vacío*, $|\varnothing| = 0$.

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 $\sqrt{2}$, π, 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.

$\lfloor x \rfloor$ 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 $e = 2.718281828459\ldots$

$\log_b x$ 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, $\log x = \ln x = \log_e x$.

$\operatorname{sen} x$,
$\sin x$ 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*, $\tan x = \operatorname{sen} x / \cos x$.

arcsen x,
arccos x,
arctan x Funciones trigonométricas inversas respectivamente de sen, cos y tan.

En Python, se indican como `asin`, `acos` y `atan` (resp.).

signo(x) Las 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 signo(0) no está definido.

$\sum$ Indica suma, $\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n$.

$\prod$ 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.

✍ Notas de índole más bien técnica. Están en letras más pequeñas y en general se pueden omitir en una primera lectura.

❦ Comentarios históricos, curiosidades, etc. Están en letras cursivas más pequeñas, y se pueden omitir en la lectura.

Cosas que hay que evitar. *En general quitan puntos en los exámenes.*

Pasaje con conceptos que pueden llevar a confusión y que debe leerse cuidadosamente.

«Moraleja», explicación o conclusión especialmente interesante. El texto está en cursiva para distinguirlo.

«Filosofía», ideas a tener en cuenta. Están a un nivel superior a las «moralejas», que a veces son sólo aplicables a Python.

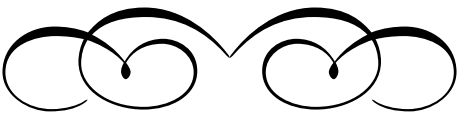
B.6. Generales

␣ Indica un espacio en blanco en entrada o salida de datos.

i. e. *es decir* o *esto es*, del latín *id est*.

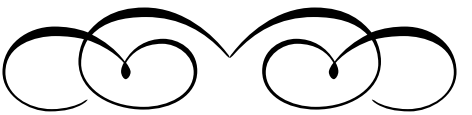
e. g. *por ejemplo*, del latín *exempli gratia*.

RAE Real Academia Española.



Parte IV

Índices



Comandos de Python que usamos

Índice de figuras y cuadros

Autores mencionados

Bibliografía

A. Engel, 1993. *Exploring Mathematics with your computer*. The Mathematical Association of America. (*págs. 7 y 243*)

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. 175*)

B. W. Kernighan y D. M. Ritchie, 1991. *El lenguaje de programación C*. Prentice-Hall Hispanoamericana, 2.ª ed. (*pág. 7*)

D. E. Knuth, 1997a. *The art of computer programming. Vol. 1, Fundamental Algorithms*. Addison-Wesley, 3.ª ed. (*págs. 7 y 8*)

D. E. Knuth, 1997b. *The art of computer programming. Vol. 2, Seminumerical algorithms*. Addison-Wesley, 3.ª ed. (*págs. 7, 8 y 163*)

D. E. Knuth, 1998. *The art of computer programming. Vol. 3, Sorting and searching*. Addison-Wesley, 2.ª ed. (*págs. 7, 8 y 179*)

M. Litvin y G. Litvin, 2010. *Mathematics for the Digital Age and Programming in Python*. Skylight Publishing. URL `http://www.skylit.com/mathandpython.html`. (*págs. 8 y 36*)

C. H. Papadimitriou y K. Steiglitz, 1998. *Combinatorial Optimization, Algorithms and Complexity*. Dover. (*pág. 263*)

R. Sedgewick y K. Wayne, 2011. *Algorithms*. Addison-Wesley, 4.ª ed. (*págs. 8 y 179*)

N. Wirth, 1987. *Algoritmos y Estructuras de Datos*. Prentice-Hall Hispanoamericana. (*págs. 7, 8, 164 y 179*)

S. Wolfram, 1988. *Mathematica - A System for Doing Mathematics by Computer*. Addison-Wesley, 1.ª ed. (*pág. 7*)

Índice alfabético