Notas para los cursos de Computación y Programación con Python

Néstor Aguilera

Año 2012

Contenidos

Capítulo 1

Preliminares

Estos apuntes, que cariñosamente llamamos libro, son una introducción a la resolución de problemas matemáticos con la computadora.

A diferencia de cursos tradicionales de programación, se cubre muy poco de las aplicaciones informáticas como bases de datos o interfases gráficas con el usuario: el énfasis es en matemáticas y algoritmos.

Usando el lenguaje Python, vemos temas elementales de análisis y cálculo numérico, teoría de números, combinatoria y grafos, sirviendo tanto de presentación de algunos temas como de repaso y fortalecimiento de otros.

Quedan aparte temas de álgebra lineal, ya que la resolución numérica de problemas lineales requiere un estudio cuidadoso de los errores, lo que está fuera del propósito introductorio de estas notas.

Hay muy poca teoría, que se habrá visto o se verá en otros cursos. Lo esencial aquí son los ejercicios.

En 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 capítulos 2 a 17 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.

Las páginas del libro están diseñadas para ser impresas en doble faz (y eventualmente anillarse), mientras que la versión electrónica (en formato «pdf» y disponible en `http://www.santafe-conicet.gov.ar/~aguilera/libros/2012`) ofrece la ventaja de vínculos (*links*) remarcados en azul, que pueden ser externos como el anterior o internos como en ejercicio 3.1.

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 a la izquierda:

```
como éste
```

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 del libro y no están incluidos en la distribución de Python. Hay dos excepciones: los módulos *grpc*

(para gráficos de puntos y curvas con ejes cartesianos) y *grgr* (para grafos) que se usan como «cajas negras». No se incluyen acá (pero sí están en la página del libro) porque son relativamente grandes y —aunque elementales— no veremos muchas de las estructuras que tienen. Todos los módulos no estándar que usamos están en la página del libro.

En el apéndice B hay una síntesis de notaciones, convenciones o abreviaturas.

Al final se incluyen índices de palabras y autores, tal vez no necesarios en la versión electrónica pero posiblemente sí en la impresa.

1.2. 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.

Actualmente no hay lenguajes destinados a la enseñanza de la programación desde un enfoque matemático y que estén ampliamente disponibles.

Entre los lenguajes con mayor difusión hoy,[1] hemos elegido Python, `http://www.python.org`, creado por G. van Rossum hacia 1990.

La elección de Python (pronunciado *páizon*) se debe a varios motivos, entre ellos:

- Es fácil empezar a programar, pudiendo empezarse con 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 lleva a confusión.
- La sintaxis no es consistente.

  ✍ Por ejemplo, `print` es una función, `remove` es un método, `return` y `del` no son ni funciones ni métodos, `sort` y `reverse` modifican una lista pero `sorted` da una lista y `reversed` da un iterador.

- Expresiones que no tienen sentido en matemáticas son válidas en Python.

  ✍ En la sección 3.6 vemos algunos ejemplos.

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] Ver `http://www.tiobe.com/index.php/content/paperinfo/tpci/`. C es muy duro como primer lenguaje.

1.2.1. Censura, censura, censura

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

> *De los 255 módulos que trae la distribución, sólo permitiremos el uso explícito de math y random.*

Aunque debemos mencionar que:

- usaremos la función `os.getcwd` para entender dónde busca Python los módulos al usar `import`,
- el módulo *builtins* se usará implícitamente.

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

- Veremos pocos tipos de datos (`int`, `float`, `str`, `bool` y las secuencias `tuple`, `list` y `range`). Por ejemplo, no veremos diccionarios (`dict`) o conjuntos (`set`).
  - ✍ Usaremos diccionarios sin mayores explicaciones (apretar botones) en los módulos *grpc* y *grgr*.
- 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`.
- No veremos generadores, a pesar de que hacen que Python sea más eficiente. En particular, no veremos `filter`, `map`, y `zip`.
  - ✍ Veremos `range` (capítulo 8) y lo usaremos ampliamente.
  - ✍ Las listas de Python son ineficientes, pero para nosotros no tendrá consecuencias graves ya que en general manejaremos tamaños pequeños en el curso (salvo algunos objetos combinatorios que examinaremos individualmente en el capítulo de recursión).

> *Si decimos que no veremos determinada construcción de Python en el curso, está prohibido usarla en la resolución de ejercicios.*

En la página 183 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.

Terminamos destacando una convención muy importante.

Un párrafo que empieza 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.*

1.3. Comentarios

- En la redacción se puede notar la influencia de 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) sigue una filosofía similar a la nuestra, pero a nivel de secundaria y está en inglés.
- A los lectores que quieran aprender más de programación, les sugerimos el ya mencionado de Wirth (1987) y, para un estudio más profundo, los de Knuth (1997a; 1997b; 1998), dependiendo del tema de interés.
- Al final de algunos capítulos hay referencias bibliográficas sobre temas y problemas particulares, pero muchos de ellos pertenecen al «folklore» y es difícil determinar el origen.
- La mayoría de las referencias históricas están tomadas de `http://www-history.mcs.st-and.ac.uk/`, pero hay varias tomadas de `http://mathworld.wolfram.com/` y también de WikipediA.
- Una posibilidad muy interesante para quienes les haya «picado el bichito» es tomar problemas o directamente participar en las competencias estudiantiles de la ACM (Association for Computing Machinery) en las que los estudiantes de nuestro país participan exitosamente desde hace varios años.

Capítulo 2

El primer contacto

En este capítulo hacemos 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 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*, es decir, uno a la vez, aunque 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,
- este intercambio es secuencial, y

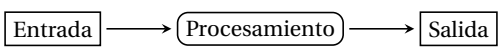


Figura 2.1: Esquema de entrada, procesamiento y salida.

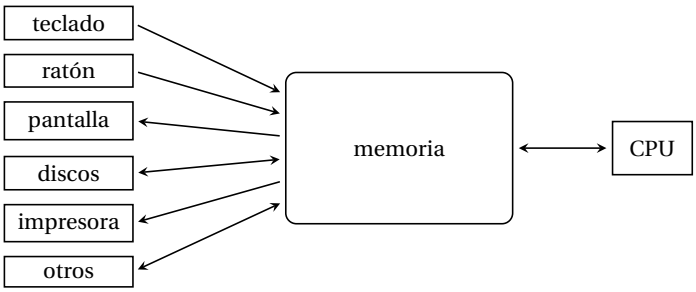


Figura 2.2: Esquema de transferencia de datos en la computadora.

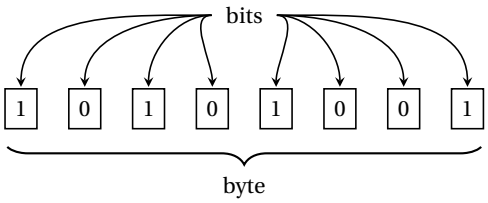


Figura 2.3: Un byte de 8 bits.

- 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 pequeñas 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.

❦ *La palabra* dígito *viene del latín* digitus *= dedo, y originalmente se refería a los números entre 1 y 10 (como la cantidad de dedos en las manos). El significado fue cambiando con el tiempo y actualmente según la RAE:*

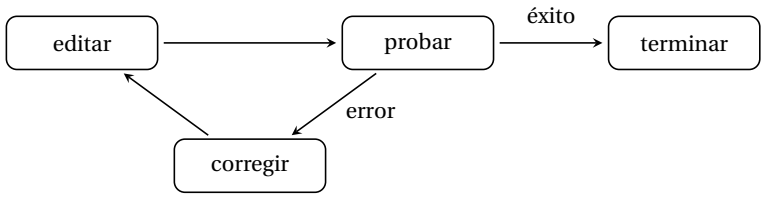


Figura 2.4: Esquema del desarrollo de un programa.

un número dígito es aquél que puede expresarse con un solo guarismo.

Así, en la numeración decimal los dígitos son los enteros entre cero y nueve (ambos incluidos), mientras que en base 2 *los dígitos son* 0 *y* 1.

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

2.3. Programas y lenguajes de programación

El conjunto de instrucciones que damos a la computadora para realizar determinada tarea es lo que llamamos *programa*. 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 0 y 1.

Cuando trabajamos con Python, el programa llamado (casualmente) *python* hace esta 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 a Python que las ejecute.

Cuando trabajamos en Python y hablamos de programas, nos referimos tanto a las instrucciones que escribimos en la terminal o que guardamos en *módulos*, como veremos en el capítulo 5.

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*.

Varios editores de texto, como *emacs* o *vim* en Unix, o *notepad* en MS-Windows, son de uso general y se pueden usar para otros lenguajes de programación u otras tareas. Por ejemplo, estas notas están escritas con un editor de texto general y luego procesadas con LaTeX (que se pronuncia *látej*), en vez de Python.

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, Python puede instalarse con 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 (3.2.2 al escribir estas notas), que puede obtenerse del sitio oficial de Python, donde hay instalaciones disponibles para los principales sistemas operativos.

Algunas observaciones:

- Si hay problemas para instalar la versión oficial de Python, puede probarse con la instalación ActivePython de ActiveState, siendo gratis las versiones «Community Edition».
- La versión que usaremos no es completamente 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 de los sistemas Unix vienen con Python preinstalado, pero posiblemente se trate de una versión anterior. Para verificar si la versión 3.2 está instalada, puede ponerse en la terminal de Unix:

  ```
  which python3.2
  ```

  y si no aparece la ubicación, debe instalarse.

  En estos sistemas generalmente hay una opción para instalar nuevas aplicaciones, y puede ser que no esté disponible directamente la última versión sino alguna anterior. Para lo que haremos en el curso no habrá mayores diferencias si se usa alguna de éstas (siempre que sea mayor que 3 y no 2.7 o menor).

En algunos casos, IDLE se ofrece como instalación separada (que debe instalarse). Además, IDLE usa el programa *Tcl*, también 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 3.2.2 (v3.2.2:137e45f15c0b, Sep  3 2011, 17:28:59)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

si se instaló la versión oficial, o a

```
ActivePython 3.2.2.3 (ActiveState Software Inc.) based on
Python 3.2.2 (default, Sep  8 2011, 12:20:28)
[GCC 4.0.2 20051125 (Red Hat 4.0.2-8)] on linux2
Type "copyright", "credits" or "license()" for more information.
>>>
```

con la versión de ActiveState.

Lo importante aquí es verificar que aparezca anunciado `Python 3.2.2` en alguna parte de los carteles: versiones menores nos pueden dar 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 «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 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. Para salir de IDLE podemos elegir el menú correspondiente.

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

Capítulo 3

Tipos de datos básicos

En este capítulo hacemos un estudio más sistemático de Python como calculadora, empezando con varias operaciones con números. Vemos que hay distintas clases de números, que podemos hacer preguntas esperando respuestas de «verdadero» o «falso», y que podemos poner carteles en las respuestas.

3.1. Enteros y decimales

Ejercicio 3.1 (operaciones con números). 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 `+` (suma) 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) `*` (producto, en matemáticas « $\times$ »),

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!).

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 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`
d) `4 - 3 - 2` *e*) `4 - (3 - 2)` *f*) `(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 $2+3\times4$, sabemos que tenemos que calcular primero 3×4 y a eso agregarle 2, o sea, « $\times$ » tiene mayor precedencia que « $+$ ». Si en cambio tenemos $2-3+4-5$, no hay precedencias entre « $+$ » y « $-$ », y evaluamos de izquierda a derecha. La cosa se complica si consideramos $3/4\times5$: en matemáticas no tiene sentido.

Python sigue reglas similares, aunque evalúa cosas como `3 / 4 * 5` de izquierda a derecha, como en el caso de sumas y restas. Como veremos en la sección 3.6, en el curso evitaremos este tipo de construcciones y otras que parecen retorcidas desde las matemáticas.

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.

✍ 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).

a) Ver qué hacen estas funciones poniendo `help(abs)` y luego `help(round)`.

b) Conjeturar y evaluar el resultado:

i) `abs(12)` *ii*) `abs(12.3)` *iii*) `abs(-12.3)`
iv) `round(12.3)` *v*) `round(12.7)` *vi*) `round(12.5)`
vii) `round(-12.3)` *viii*) `round(-12.7)` *ix*) `round(-12.5)`

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(12.3)` *b*) `round(12.3)`
c) `type(round(12.3))` *d*) `type(int(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. Por otro lado,* `float` *viene de* punto flotante, *el nombre de la codificación para números decimales, tema que vemos con detalle en el capítulo 15.*

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) `type(12.3)` *b*) `int(12.3)`
c) `type(int(12.3))` *d*) `type(-45)`
e) `float(-45)` *f*) `type(float(-45))`
g) `int(float(89))` *h*) `float(int(7.65))` ¶

Ejercicio 3.9. Explorar la diferencia entre `int` y `round` cuando aplicados a números decimales. Por ejemplo, calcular estas funciones para 1.2, 1.7, −1.2, −1.7.
También poner `help(round)` (que ya vimos) y `help(int)`. ¶

3.2. ¿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.

Matemáticas:	$=$	$\neq$	$>$	$\geq$	$<$	$\leq$	$\wedge$ (y)	$\vee$ (o)	$\neg$ (no)
Python:	`==`	`!=`	`>`	`>=`	`<`	`<=`	`and`	`or`	`not`

Cuadro 3.1: Traducción de algunas expresiones entre matemática y Python.

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, 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 e inconvenientes, como estudiamos en la sección 3.6.

  ❦ `bool` *es una abreviación de Boolean, que podemos traducir como* booleanos *y pronunciar* buleanos. *Los valores lógicos también reciben ese nombre en honor a G. Boole (1815–1864), quien hizo importantes progresos al «algebrizar» la lógica.*

- `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*).*

3.3. Tipo lógico

Repasemos un poco, 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 3.10. 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
```

En el cuadro 3.1 vemos cómo convertir algunas expresiones entre matemática y Python, recordando que «and», «or» y «not» son los términos en inglés para *y*, *o* y *no*, respectivamente.

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

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

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

- *la igualdad «$=$» en matemáticas se representa con «`==`» en Python,*

- *la instrucción «`=`» en Python es la asignación que veremos en el capítulo 4.*

Ejercicio 3.11. Usar Python para determinar la validez de las expresiones del ejercicio 3.10, observando que en Python (como en matemáticas) la expresión `a < b < c` es equivalente a `(a < b) and (b < c)`. ¶

Ejercicio 3.12. Conjeturar y verificar en Python:

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

Ejercicio 3.13. Usando `type`, ver que las expresiones `4 < 5`, `4 != 5`, `4 == 5` son de tipo `bool`. ¶

Ejercicio 3.14. ¿Cuál es el resultado de `1 > 2 + 5`?, ¿cuál es la precedencia entre `>` y `+`? ¿Y entre `>` y otras operaciones aritméticas (como `*`, `/`, `**`)? ¶

Ejercicio 3.15. 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 3.16 (errores numéricos). Usando `==`, `<=` y `<`, comparar

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

✍ `1.0e-10` en notación 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?

✍ En el capítulo 15 estudiaremos problemas de este tipo. ¶

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». Python también usa esta convención. 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, hay que tener en cuenta la precedencia de `or` y `and` que estudiamos en el ejercicio 3.32.

Ejercicio 3.17 (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)` *e*) `(1 < 1/0) or True` ¶

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.

Otra diferencia con las matemáticas es que

a los efectos de evaluaciones lógicas, Python considera que todo objeto tiene un valor verdadero o falso,

lo cual es muy confuso y no sucede en otros lenguajes de programación. Vemos un poco de esto en la sección 3.6, pero mientras tanto:

> *Aunque válidas en Python, 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`,
>
> *que son ridículas en matemáticas.*

3.4. 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 3.18. Ingresar en la terminal `'Ana Luisa'` y averiguar su tipo poniendo `type('Ana Luisa')`. ¶

Ejercicio 3.19 (comparación de cadenas).

a) Ver que para Python `'mi mama'` y `"mi mama"` son lo mismo usando `==`.

b) Evaluar `'mi mama' < 'mi mama me mima'`.
Repetir cambiando `<` por `==` y después por `<=`.

c) Repetir el apartado anterior (comparando mediante `==`, `<` y `<=`) para:

i) `'mi mama'` y `'me mima'`.

ii) `'mi mama'` y `'123'`.

iii) `123` y `'123'`. ¶

Ejercicio 3.20. Como en casos anteriores, muchas veces podemos pasar de un tipo a otro.

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')`

✍ Como ya mencionamos para casos similares, es ridículo decidir si `'pepe'` es verdadero o falso. Ver también la sección 3.6.

c) No siempre podemos volver al objeto original:
i) `int('1')` *ii*) `int('1.0')` *iii*) `int('False')`
iv) `float('1')` *v*) `float('1.0')` *vi*) `float('False')`
vii) `bool('1')` *viii*) `bool('1.0')` *ix*) `bool('False')` ¶

Ejercicio 3.21 (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 3.22 (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):

i) `'pa' + 'ta'` *ii*) `'ta' + 'pa'`

c) Evaluar `'mi mama' + 'me mima'`.
¿Cómo podría modificarse la segunda cadena para que el resultado de la «suma» sea `'mi mama me mima'`?

d) ¿Hay diferencias entre `len('mi mama') + len('me mima')` y `len('mi mama me mima')`?, ¿es razonable? ¶

Ejercicio 3.23 (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 3.24.

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'` ¶

Ejercicio 3.25 (`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(1.2, int)`
iv) `isinstance(1, bool)`
v) `isinstance('mama', float)`
vi) `isinstance(True, int)`
vii) `isinstance(True, bool)`
viii) `isinstance(True, float)` ¶

Con `help(str)` podemos ver las muchas operaciones que tiene Python para cadenas. Nosotros veremos sólo unas pocas en el curso, algunas de ellas en el capítulo 8, dentro del contexto general de sucesiones.

3.5. `print`

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

Ejercicio 3.26.

a) ¿Cuál es la diferencia entre poner en la terminal `123` y `print(123)`?
Repetir cuando en vez de `123` se pone

i) `-7.89` *ii*) `True` *iii*) `'mi mama me mima'`

b) `print` puede no tener argumentos explícitos. Evaluar:

i) `print()` *ii*) `print('')` *iii*) `print('␣')` *iv*) `print`

c) Y 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 3.27. Aunque matemáticamente no tiene mucho sentido, la multiplicación de un entero por una cadena que vimos en el ejercicio 3.24 es útil para imprimir. Evaluar:

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

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

Ejercicio 3.28. 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 3.29. 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. ¶

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 3.30.

a) Ver el resultado de `print('/\\')`.

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

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

¶

3.6. En el filo de la navaja

En esta sección vemos cosas que no tienen sentido en matemáticas, pero sí en Python, y algunos remedios posibles. Recordando que en este curso usamos Python como herramienta, *pero no es un curso de Python*, huiremos de estas construcciones.

Ejercicio 3.31. En este ejercicio seguimos analizando la precedencia de operadores, encontrando expresiones no muy elegantes.

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

b) Poner `2 + - 2` y explicar el resultado.

c) Repetir el apartado anterior para

i) `2 + + 2` *ii*) `2 + + + 2` *iii*) `2 + - + 2`

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

e) Conjeturar y verificar el resultado de

i) `2 + 3 * - 4` *ii*) `2 * - 3 ** - 4 + 1`.

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

Ejercicio 3.32. Como en el caso de la expresión $3/4 \times 5$, en matemáticas tratamos de evitar expresiones como $a \wedge b \vee c$ (que se lee «a y b o c»), ya que no es claro si se debe evaluar primero $\wedge$ o $\vee$ (no está definida la precedencia). Pero en Python...

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

b) Decidir si al mezclar `and` y `or` hay precedencia de uno sobre otro, o si se realizan las evaluaciones de izquierda a derecha (como en $1-2+3$, recordar el ejercicio 3.31). ¶

Ejercicio 3.33. Como ya mencionamos, *a los efectos de evaluaciones lógicas* Python considera que *todo objeto tiene un valor verdadero o falso.*

Afortunadamente, son pocas las cosas que evalúan a `False`, y bastará acordarse de éstas ya que todas las otras darán `True`. De las que hemos visto, sólo `0`, `0.0`, la cadena vacía `''`, y (claro) `False` se evalúan (en lógica) como falsos.

a) Evaluar con Python:

i) `bool(1)`	*ii*) `bool(0)`	*iii*) `bool(1.2)`
iv) `int(False)`	*v*) `int(True)`	*vi*) `float(False)`
vii) `0 < True`	*viii*) `3 + False`	*ix*) `False + True`
x) `bool('')`	*xi*) `bool('␣')`	*xii*) `bool('0')`

b) Recordando el ejercicio 3.32, conjeturar y luego comprobar el valor de

i) `1 and 2 or 3` *ii*) `3 or 1 and 2`

c) En Python, las expresiones

`-1 + 3 + 2`, `(-1 + 3) + 2`, y `-1 + (3 + 2)`

dan todas el mismo resultado.

Ver que, en cambio, `-1 < 3 < 2` da un resultado distinto de `(-1 < 3) < 2` y de `-1 < (3 < 2)`. ¶

3.7. Comentarios

- La presentación como calculadora es típica en sistemas interactivos como *Matlab* o *Mathematica*. En particular, aparece en el *tutorial de Python*.
- Los ejercicios 3.30 y 3.31 son variantes de similares en Litvin y Litvin (2010).

Capítulo 4

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. Este mecanismo se llama *asignación*.

4.1. 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 4.1.*a*).

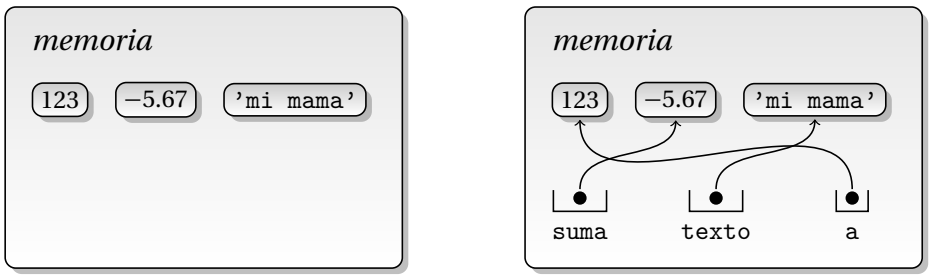
En Python podemos hacer referencia posterior a estos objetos, poniendo un nombre, llamado *identificador*, y luego una asignación, indicada por `=`, que relaciona el identificador con el objeto. Así, el conjunto de instrucciones

```
a = 123
suma = -5.67
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 4.1.*b*).

✍ Recordemos que los datos, incluyendo instrucciones, se guardan en la memoria de la computadora como ristras de ceros y unos. En la figura 4.1 ponemos los valores «humanos» para entender de qué estamos hablando.

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.



a) Objetos en la memoria.

b) Asignaciones.

Figura 4.1: Objetos en la memoria.

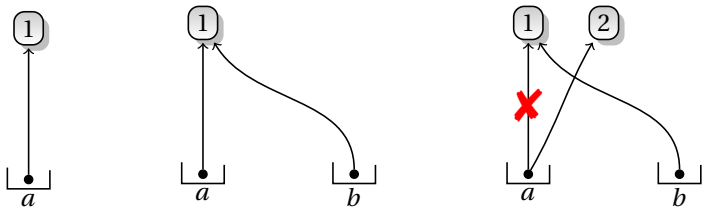


Figura 4.2: Ilustración de asignaciones.

Ejercicio 4.1.

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 valor 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 4.2, izquierda)
`b = a`	→	`b` también es una referencia a 1 (figura 4.2, centro)
`a = 2`	→	ahora `a` es una referencia a 2 (figura 4.2, derecha)
`a`	→	el valor de `a` es 2
`b`	→	`b` sigue siendo una referencia a 1

Ejercicio 4.2. En cada caso, predecir el resultado del grupo de instrucciones y luego verificarlo, recordando que primero se evalúa el miembro derecho y luego se hace la asignación:

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

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

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

d)
```
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 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 4.3.

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?

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

Ejercicio 4.4 (palabras reservadas). Python tiene *palabras reservadas* que no pueden usarse como identificadores.

a) Para encontrarlas, ponemos en la terminal `help()` y después `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)
```

☞ *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 no pueden usarse como identificadores. ¶

Veamos algunos ejemplos donde aparece la ventaja de usar variables (en contraposición a usar sólo fórmulas o valores).

Ejercicio 4.5. Python puede trabajar con números enteros de cualquier tamaño.

a) Poner

```
a = 87
b = 12
c = a ** b
c
```

viendo que el valor de `c` es muy grande y difícilmente podamos escribirlo sin cometer errores: sintetizarlo como `c` es una buena idea.

b) Poner `float(c)` para obtener una idea del tamaño de `c` (tiene 24 cifras en base 10).

c) Poner `type(c)` para ver que efectivamente `c` es entero.

d) Calcular `d = float(a) ** float(b)` y comparar el valor de `d` y `c` (por ejemplo, tomando la diferencia). ¶

Ejercicio 4.6. Si bien Python puede trabajar con enteros de cualquier tamaño, los tamaños de los decimales con los que puede trabajar es limitado.

Repitiendo lo hecho en el ejercicio anterior, poner:

a)
```
a = 876
b = 123
c = a ** b
c
```
viendo que `c` es muy grande, pero no hay error.

b) Ver que poniendo `float(c)` da error pues no puede representar ese valor como decimal.

c) ¿Qué pasará si ponemos `float(a) ** float(b)`?

✍ 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} (en computadoras modernas). $876^{123}\approx 8.4724\times10^{361}$ es demasiado grande.

Estudiaremos estos temas en el capítulo 15. ¶

Ejercicio 4.7. Cuando trabajamos con 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
```
. (4.1)

✍ En el capítulo 9 veremos la función `divmod` que hace una tarea similar con una única instrucción.

Para ver el comportamiento de Python, probamos los siguientes tomando 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 (4.1), 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 `b` decimal y positivo.

Ayuda: hay varias posibilidades, una es pensar en radianes y $b=2\pi$, otra es pensar en horas, y en general en cosas cíclicas.

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

4.2. Comentarios

- Es posible ver cuáles son las variables definidas usando `globals`, obteniendo un *diccionario*, con entradas de la forma `'identificador': valor`.

  Como tantas otras cosas de Python, en este curso no veremos ni `globals` ni la estructura de diccionario.

Capítulo 5

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 uso posterior, como ya mencionamos en la sección 2.3. 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 muy pocos de éstos, sólo *math*, *random*, y un breve pasaje por *os*.

  ✍ El módulo *builtins* se instala automáticamente al iniciar Python en la terminal.

- 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 6).

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

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

  ```
  import pepe
  ```

  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 5.4. En la gran mayoría de los casos, usaremos el primer método con los módulos estándares y el segundo con los módulos que construimos.

Veamos algunos ejemplos.

5.1. Módulos estándares: *math*

Muchas funciones matemáticas no están entre las predeterminadas (no están en el módulo *builtins*) de Python, y a nosotros nos van a interesar particularmente las funciones trigonométricas seno, coseno y tangente, la exponencial y su inversa el logaritmo, así como las constantes relacionadas $\pi = 3.14159\ldots$ y $e = 2.71828\ldots$

Para agregar estas funciones y constantes apelamos al módulo estándar *math*, usando las traducciones de matemática a Python del cuadro 5.1, y recordando que en las

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

Cuadro 5.1: Traducciones entre matemáticas y el módulo `math`.

instrucciones de Python hay que anteponer «`math.`», por ejemplo, π en matemáticas se escribe como `math.pi` en Python.

Ejercicio 5.1. 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) ¿Es $\sqrt{4}$ entero? ¿De qué tipo es `math.sqrt(4)`? ¶

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

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

Ejercicio 5.2. 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 5.3 (funciones trigonométricas). En Python, como en matemáticas en general, las funciones trigonométricas toman los argumentos en radianes, y en todo caso podemos pasar de grados a radianes multiplicando por $\pi/180$. Así, $45° = 45° \times \pi/180° = \pi/4$ (radianes).

Poniendo `import math` en la terminal, 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) Averiguar qué hacen las funciones `math.radians` y `math.degrees` usando `help`, y luego calcular nuevamente $\tan 60°$ usando `math.tan` y alguna de estas funciones.
f) Calcular seno, coseno y tangente de 30° usando Python (las respuestas, por supuesto, deberían ser aproximadamente $1/2$, $\sqrt{3}/2$ y $\sqrt{3}/3$).
g) ¿Cuánto es $\tan 90°$?, ¿y según Python?

✍ Observar que Python no puede calcularlo exactamente: tanto `math.pi` como `math.tan` son sólo aproximaciones a π y tan. ¶

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 5.4 (exponenciales y logaritmos). Poniendo `import math` en la terminal, realizar los siguientes apartados:

a) Calcular `math.e`.

b) Usando `help(math.log)`, 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, podemos usar `math.log10`: ver qué hace esta función (usando `help`).

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

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 5.5. ¿Cuál es mayor, e^π o π^e? ¶

Ejercicio 5.6. 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 como `math.floor` (*piso*) y `math.ceil` (*techo*) respectivamente. Ver el funcionamiento de estas funciones calculando el piso y el techo de ± 1, ± 2.3 y ± 5.6. ¶

Ejercicio 5.7 (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) Recordando el ejercicio 4.6, usar estas ideas para encontrar la cantidad de cifras (en base 10) de 876^{123}.

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

5.2. Módulos propios

En esta sección construiremos nuestros 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 5.8 (Hola Mundo).

a) Abrir una ventana nueva en IDLE distinta de la terminal (menú *File* → *New Window*), escribir en ella (en un único renglón) `print('Hola Mundo')`, y guardar en un archivo con nombre *holamundo.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). Además, por prolijidad es mejor que el directorio no sea el directorio principal del usuario.
Por ejemplo, podrían ponerse todos 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 el módulo tiene 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.

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`.

f) Poner `import os` y luego `os.getcwd()`: aparecerá el nombre de un directorio (o carpeta), llamado *directorio actual de trabajo* (*current working directory*).

Verificar que coincide con el directorio donde hemos guardado el archivo *holamundo.py*. ¶

5.3. Ingreso interactivo de datos

Cuando trabajamos sólo con la terminal de IDLE, podemos asignar o cambiar valores sin mucho problema, pero la situación es diferente cuando se ejecuta o importa un módulo y queremos ingresar los datos a medida que se requieren. Una forma de ingresarlos es interactivamente, donde el módulo nos los va pidiendo a medida que son necesarios.

Ejercicio 5.9 (holapepe). *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 en IDLE, 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) `pa'que'` *ii*) `123␣'123"` *iii*) `agu"ero`

d) Los renglones que se escriben en la ventana *holapepe* no deben tener sangrías, aunque puede 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 5.10 (comentarios en el código). Cuando un renglón tiene el símbolo `#`, Python ignora este símbolo y todo lo que le sigue en ese renglón. Esta acción se llama *comentar* el texto.

Veamos el efecto en el módulo *holapepe*:

a) Agregar `#` al principio del primer renglón que empieza con `print` y en el renglón siguiente cambiar `input()` por `input('¿Cómo te llamás?')`. ¿Cuál es el efecto?

b) El nombre ingresado queda demasiado junto a la pregunta en `input`. ¿Cómo se podría agregar un espacio a la pregunta para que aparezcan separados?

c) *Descomentar* el renglón con `print` —o sea, sacar el `#`— y cambiar la instrucción por `print('Hola, soy la compu')` viendo el efecto.

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 3.22). ¶

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 interfase 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 interfase estéticamente agradable. En cambio:

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

Ejercicio 5.11. *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? ¶

5.4. Usando `import`

Al poner `import módulo`, Python no busca el módulo en toda la computadora (lo que llevaría tiempo) sino en ciertos directorios, dentro los cuales *siempre* están los módulos estándares como *os* o *math*.

Cuando se ejecuta un módulo en IDLE (como hicimos con *holamundo*), el directorio donde está el archivo correspondiente pasa a estar entre esos directorios. De allí la recomendación de poner todos los módulos construidos por nosotros en un mismo directorio.

✍ La lista inicial de directorios puede verse en el menú *File* → *Path Browser* de Python.
Es posible (usando instrucciones que no veremos) encontrar la lista completa de directorios y también cambiar la lista, por ejemplo, agregando otros directorios.

Ejercicio 5.12. Abrir una nueva sesión de IDLE para realizar este ejercicio.

a) Repitiendo lo hecho en el ejercicio 5.8.*f*), poner en la terminal `import os` y luego `os.getcwd()`. Si se siguieron las instrucciones del ejercicio 5.8.*a*), el directorio no será el mismo donde reside el archivo *holamundo.py*.

b) Poner `a = 5` y verificar el valor poniendo `a`.

c) Reiniciar la terminal de IDLE (con el menú *Shell* → *Restart Shell*), y poner nuevamente `os.getcwd()` (sin `import os`) o preguntar el valor de `a` (sin asignar valor a `a`): en ambos casos dará error.

☞ *Al reiniciar IDLE, se pierden los valores que teníamos:* `os` *y* `a` *no están definidos.*

d) Empezar una nueva sesión de IDLE[(1)] y volver a poner `import os` y luego `os.getcwd()`, verificando que aparece un directorio distinto al que aloja a *holamundo*.

e) Poner `import holamundo` y ver que da error.

f) En una nueva ventana en IDLE (distinta de la terminal), escribir `pass` en un único renglón, guardarlo en el archivo de texto *nada.py en el mismo directorio donde está holamundo.py*, y ejecutarlo. El resultado visible será un renglón en blanco en la terminal de IDLE.

✍ La instrucción `pass` (literalmente *paso* o *pasar*) hace... ¡nada!

✍ *nada* es una versión a la cual se agregó documentación.

g) Verificar que el directorio actual es el mismo en donde está *holamundo.py* usando `os.getcwd()` (importando `os` primero, claro), y luego poner `import holamundo` (como en el apartado *e*)), viendo que ahora no da error y en la terminal se imprime `Hola Mundo`.

h) Al poner `help(holamundo)`, la documentación de *holamundo* aparece al principio de la respuesta.

✍ Comparar con el ejercicio 5.8.*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.

Ejercicio 5.13 (espacios de nombres).

a) Repitiendo lo hecho en el ejercicio 5.3, 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.*

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

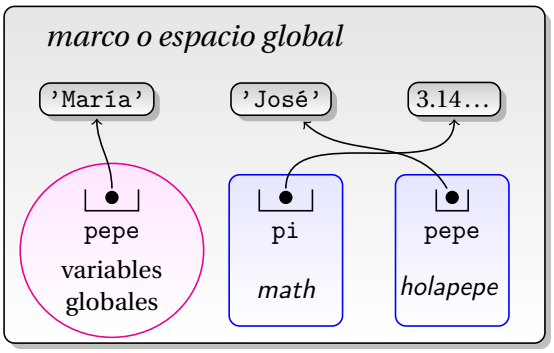


Figura 5.1: Espacio o marcos global y de módulos en el ejercicio 5.13.

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'`.

En la figura 5.1 vemos un esquema de la situación planteada, con los espacios de los módulos en azul. ¶

Capítulo 6

Funciones

Como hemos visto, por ejemplo en el ejercicio 5.9.*c*), se hace tedioso ejecutar el módulo cada vez que queremos probar con distintas entradas, y 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
```

6.1. Ejemplos simples

Ejercicio 6.1. Siguiendo las ideas del ejercicio 5.9, vamos a definir una función `hola` que dado un argumento, imprime «Hola» seguido del argumento. Por ejemplo, queremos que `hola('Mateo')` imprima «Hola Mateo».

a) Poner en una nueva ventana de IDLE (no la terminal):

```
def hola(nombre):
    print('Hola', nombre)
```

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*. IDLE pone la sangría automáticamente cuando el renglón anterior termina en `:`.

b) Guardar el módulo como *holas*, ejecutarlo (con *Run* → *Run Module*) y probar la función con las siguientes entradas:

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

✍ Como en matemáticas, primero se evalúa el argumento y luego la función (en este caso `hola`).

c) ¿Qué pasa si ponemos `hola` (sin argumentos) en la terminal? Ver que la respuesta es similar a poner, por ejemplo, `abs` (sin argumentos).

d) ¿Qué pasa si ponemos `nombre` en la terminal?

☞ *La variable* `nombre` *es* local *a la función* `hola` *y no se conoce afuera, siguiendo un mecanismo de contextos (ejercicio 5.13). El tema se explica con más detalle en la sección 6.2.*

e) Conjeturar el valor de `a` después de la asignación

```
a = hola('toto')
```

verificar la conjetura poniendo

```
a
```

y finalmente

```
print(a)
```

☞ `None` *es un valor de Python para indicar que no hay resultado visible (pero no que hay error).*

f) Siempre con `a = hola('toto')`, poner `a == None` para ver que, efectivamente, el valor de `a` es `None`.

g) Poner `None` en la terminal, y ver que no se imprime resultado alguno (a diferencia de lo que ocurre cuando se pone, e. g., `1`).

h) Como vamos a agregar otras funciones al módulo, no ponemos por ahora la documentación del módulo mismo, pero sí vamos a agregar documentación a la función `hola`, poniendo

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

Ejecutar nuevamente el módulo y poner `help(hola)`.

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

i) Imitando lo hecho con *holapepe*, en la ventana *holas* poner:

```
def hola2():
    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!

j) Guardar los cambios, ejecutar el módulo, y verificar la función y su documentación poniendo `hola2()` y `help(hola2)`.

k) Repetir los apartados *e*) y *f*) cambiando `hola` por `hola2`.

l) `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.

✍ *holas* es una versión final del módulo, que incluye documentación. ¶

El ejercicio anterior nos muestra varias cosas. Por un lado, vemos que en un mismo módulo se pueden definir varias funciones. Por otro lado, las funciones del módulo *holas* tienen el resultado visible de imprimir, pero no podemos hacer asignaciones como las que estamos acostumbrados en matemáticas del tipo $y = f(x)$, obteniendo en cambio `None`.

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`.

Ejercicio 6.2. Basados en el ejercicio 5.11, ahora definimos una función que toma *dos* argumentos y que retorna un valor que podemos usar.

a) En una ventana nueva de IDLE, poner

```
def suma(a, b):
    return a + b
```

Guardar en un archivo adecuado, y ejecutar el módulo.

b) Hacer la asignación `a = suma(8, 9)` y verificar que el valor de `a` no es `None` sino el esperado.

c) Conjeturar y verificar el resultado de los siguientes

i) `suma(2, 3)` *ii*) `suma(4.5, 6)`
iii) `suma('pi', 'pa')` *iv*) `suma(pi, 'pa')`
v) `suma(1)` *vi*) `suma(1, 2, 3)`
vii) `suma(1, None)` ¶

Ejercicio 6.3. En cada una de las funciones `hola` y `hola2` del ejercicio 6.1 incluir la instrucción `return None`, y volver a hacer las asignaciones `a = hola('toto')` y `a = hola2()`, viendo que el resultado es efectivamente `None`.

Si en vez de poner `return algo` se pone `return` y nada más, el efecto es el mismo que poner `return None` o directamente no poner `return`: verificarlo cambiando el `return None` anterior por sólo `return`. ¶

Ejercicio 6.4. En este ejercicio tratamos de deshacer los entuertos de Python con las variables lógicas que vimos en el ejercicio 3.33, apoyándonos en la función `isinstance` (ejercicio 3.25).

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:

con 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:

con 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:

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

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

6.2. Variables globales y locales

Las funciones también son objetos, y cuando definimos una función se fabrica un objeto de tipo *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.

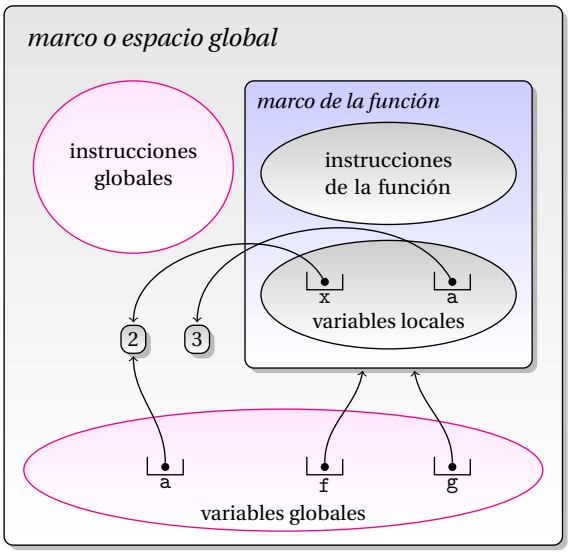


Figura 6.1: Variables globales y locales.

Así, 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`.

Ejercicio 6.5. En una nueva sesión de IDLE, 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*.

a) Poner sucesivamente:

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

y estudiar los resultados.

b) Poner

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

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

c) Poner

```
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 6.5 están dispuestos como se muestra en la figura 6.1:

- 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.

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!*

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

Ejercicio 6.6. El módulo *globyloc* tiene las funciones definidas en este ejercicio. A diferencia de otros, no es un módulo para ejecutar o importar, pues algunas definiciones son incorrectas y darán error. Lo incluimos para tener un lugar sencillo desde donde copiar y luego pegar en algún lugar adecuado.

a) En la terminal de IDLE poner

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

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

```
def f(x):
    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, no tiene sentido llamar a* `f`*.*

c) Reiniciar la terminal, poner

```
def f(x):
    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):
    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(x):
    b = a   # b es local y a podría ser global
    a = 1   # a es local porque se asigna
            # y entonces estamos en problemas
            # porque la usamos antes de asignar
    return x + a + b
```

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

f) Reiniciar la terminal, poner

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

g) Poner

```
def f(x):
    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.

h) En otra tónica, podemos usar una variable local con el mismo identificador que la función como en

```
def f(x):
    f = x + 1       # x y f son locales
    return f
```

y ejecutar el bloque

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

✍ Desde ya que este uso no es muy recomendado. ¶

Ejercicio 6.7. 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 6.8. 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))`

¶

6.3. Comentarios

- Ocasionalmente vamos a necesitar variables que no son ni locales ni globales, pero posponemos su introducción hasta la sección 17.3, cuando tengamos que usarlas.

Capítulo 7

Funciones numéricas y sus gráficos

En cálculo o análisis matemático trabajamos con funciones de la forma $f : A \to \mathbb{R}$, donde A es un intervalo o eventualmente todo $\mathbb{R}$. En este capítulo vemos cómo Python nos puede ayudar en el estudio haciendo gráficos aproximados de estas funciones.

7.1. Funciones numéricas

Ejercicio 7.1. 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 7.2. En este ejercicio consideramos funciones de varias variables.

a) Construir una función `hmsas(h, m, s)` que dado un tiempo expresado en horas (`h`), minutos (`m`) y segundos (`s`), lo pase a segundos.
Por ejemplo, 12 hs 34 m 56.78 s son 45296.78 segundos, i. e., `hmsas(12, 34, 56.78)` debe dar `45296.78`.

b) De modo similar, construir una función `gmsar(g, m, s)` que ingresando la medida de un ángulo en grados, minutos y segundos, retorne la medida en radianes.
Por ejemplo, 12° 34′ 56.78″ son 0.21960… radianes. ¶

7.2. El módulo *grpc*

Aprovechando las facilidades gráficas del módulo *tkinter* de Python, usaremos el módulo *grpc* (por gráficos de puntos y curvas) para hacer gráficos en dos dimensiones.

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

En el curso usaremos *grpc* como «caja negra», como hacemos con *math*, sólo usaremos algunos comandos y no estudiaremos las instrucciones en él. En esta sección nos concentramos en los gráficos de funciones, pero *grpc* también nos permite agregar otros elementos, como puntos, poligonales, texto y otros más (poner `help(grpc)`).

✍ *grpc* es muy largo para incluirlo aquí y no tiene mucho sentido hacerlo ya que no estudiaremos su contenido. Se puede bajar de la página del libro, como los otros módulos mencionados no incluidos en la distribución de Python.

Ejercicio 7.3.

a) Copiar el módulo *grpc* en el directorio donde se guardan nuestros módulos, y poner `help(grpc)` y `help(grpc.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 `grpc.funcion` es la función, que debe estar definida con anterioridad, y los otros dos son los extremos del intervalo (cerrado) donde hacer el gráfico.
- *grpc* espera que primero se definan los elementos del gráfico, y luego lo construye con `grpc.mostrar()`.
- El gráfico no conserva la escala 1 : 1 entre los ejes, sino que pone el gráfico en el tamaño de la ventana disponible.[(1)]
- 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 (dentro de *grseno*, antes de `grpc.mostrar()`):

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

iii) Recíprocamente, poniendo `grpc.ymin = 0.2` y `grpc.ymax = 0.8` hace que se «rebane» el gráfico.

✍ 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`. ¶

Ejercicio 7.4. Graficar:

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

(1) Ver también el ejercicio 7.5.

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

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

Ejercicio 7.5. *grpc* permite hacer el gráfico de varias funciones simultáneamente, alternando los colores respectivos. En estos casos es conveniente poner leyendas para distinguirlas, como se hace en el módulo *grexplog*, donde graficamos las funciones e^x entre -2 y 5, $\log x$ entre (casi) 0 y 5 (usando la técnica del ejercicio 7.7), 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) 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.

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

iii) El gráfico no refleja bien la simetría respecto de la diagonal $y = x$, debido a que las escalas de los ejes es distinta.

La ventana gráfica tiene 485 pixeles de ancho y 330 pixeles de alto (valores defecto). ¿Qué valor de `ymax` habrá que poner en *grexplog*, con `xmin = -2`, `xmax = 5` y `ymin = xmin` para que los ejes se muestren en la misma escala (o sea, que la diagonal aparezca a 45°)?

Sugerencia: hacer los cálculos en el mismo módulo.

Respuesta: 2.76288…

b) 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 `grpc.ymax` (por ejemplo, comentando el renglón correspondiente).

ii) Tomando `grpc.ymax = 10`.

iii) Repitiendo el apartado *iii)* anterior. ¶

Ejercicio 7.6. Recordando el ejercicio 7.1:

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 7.1.*e)* usando el cursor.

c) Cambiar los nombres de los ejes usando `leyendax` y `leyenday`[2] 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 7.7 (gráfico de funciones con saltos). *grpc* hace el gráfico evaluando la función en algunos puntos[3] y luego uniéndolos, por lo que algunas funciones no se grafican correctamente.

a) La función $y = 1/x$ no está definida para $x = 0$. Si queremos graficarla en el intervalo $[-1, 1]$, nos dará error si uno de los puntos evaluados es $x = 0$, pero en otro caso nos dará un gráfico incorrecto.

i) El bloque

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

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

[3] Por defecto, 201 puntos equiespaciados.

posiblemente dará error de división por cero.

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

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

y ver el resultado.

iii) Una solución más satisfactoria es dividir el intervalo en dos o más, evitando los saltos, como en el módulo *gr1sobrex*, donde se puede ajustar `eps` (que debe ser pequeño).

✍ Como la función es la misma a ambos lados, ponemos explícitamente el estilo (en este caso, dibujar en azul), en otro caso se dibujarían con distintos colores (ver ejercicio 7.5).

✍ También se introduce `grpc.poligonal`, que dibuja una poligonal uniendo puntos, que deben estar dados en la forma `[(x1, y1), (x2, y2),...]`, las coordenadas de cada punto entre paréntesis, y todas encerradas por corchetes.

De esta forma podemos dibujar los ejes coordenados en el lugar usual, cortándose en $(0,0)$.

b) La función tangente no está definida en $\pi/2+k\pi$, para $k \in \mathbb{Z}$. Repetir el apartado anterior para graficar esta función en el intervalo $(-\pi/2, \pi)$.

✍ La evaluación de Python de $\tan \pi/2$ no da error, como vimos en el ejercicio 5.3. ¶

Capítulo 8

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 en Python es con *sucesiones*, de las cuales nos van a interesar especialmente las *listas*.

Las sucesiones de Python tienen varias similitudes con los conjuntos (finitos) de matemáticas. Por ejemplo, podemos ver si cierto elemento está o no, podemos agregarlos (como en la unión de conjuntos), podemos encontrar la cantidad de elementos que tiene, y existe la noción de sucesión *vacía*, que no tiene elementos. No obstante 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 como `'nata'` —que hemos visto en el capítulo 3— son sucesiones. Acá veremos tres nuevos tipos de sucesiones: *tuplas*, *listas* y *rangos*.

Las sucesiones comparten una serie de operaciones en común, como el cardinal o *longitud* está dado por `len`, que ya hemos visto para cadenas. Veamos dos de estas características comunes: *índices* y *secciones*.

8.1. Índices y secciones

Ejercicio 8.1 (índices de sucesiones). Si la sucesión `a` tiene n elementos, éstos pueden encontrarse individualmente con `a[i]`, donde i es un *índice*, $i = 0, \dots, n-1$. Índices negativos cuentan desde el final hacia adelante ($i = -1, \dots, -n$), y poniendo i fuera del rango $[-n, n-1]$ da error.

Resolver los siguientes apartados ingresando `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) Poner `a[0]` y `a[n-1]`, comprobando que dan la primera y última letras de `a`.

c) Poner `a[-1]` y `a[-n]`, comprobando que dan la última y primera letras de `a`.

d) Ver que `a[n]` y `a[-n-1]` dan error. ¶

Ejercicio 8.2 (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 3.22).

c) Es un error usar un único índice fuera de rango como vimos en el ejercicio 8.1. 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]` ¶

8.2. Tuplas (`tuple`)

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 8.3 (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 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 8.4 (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.

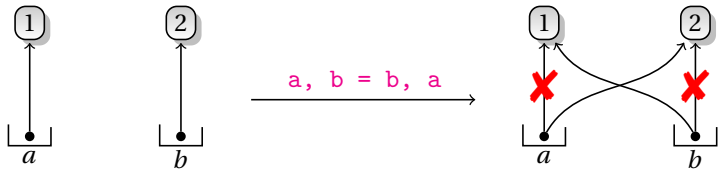


Figura 8.1: Efecto del intercambio.

a) Poner

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

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

b) Poner ahora

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

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

✍ El efecto del intercambio se ilustra en la figura 8.1.

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

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

¶

8.3. Listas (`list`)

Nosotros usaremos tuplas muy poco: en general las usaremos para indicar coordenadas como en $(1,-1)$, para el intercambio mencionado en el ejercicio 8.4, a veces para hacer asignaciones múltiples como en el ejercicio 8.3.*d*), y ocasionalmente como argumentos o resultados de funciones como `isinstance` (en el ejercicio 8.22) o `divmod` (que vemos en el capítulo 9).

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 8.5 (listas).

a) Poniendo en la terminal

```
a = [123, 'mi mamá', 456]
```

i) Repetir el apartado *a*) del ejercicio 8.3.

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

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,]`

c) Aunque hay que escribir un poco más, también podemos hacer asignaciones múltiples e intercambios con listas como hicimos con tuplas (en los ejercicios 8.3 y 8.4).

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 8.6 (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.[(1)]

El concepto de mutabilidad e inmutabilidad es propio de Python, y no existe en lenguajes como C o Pascal.

Ejercicio 8.7. Los números y cadenas también son inmutables. Comprobar que

```
a = 'mi mamá'
a[0] = 'p'
```

da error. ¶

Las sucesiones mutables, como las listas, admiten que se cambien individualmente sus elementos, pero también que se agreguen o borren.

Ejercicio 8.8 (operaciones con listas). Pongamos

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

a) Sin usar Python: ¿cuál es la longitud de `a`?, ¿qué valor tiene `a[3]`?

b) En cada uno de los siguientes, verificar el valor resultante de `a` y su longitud luego de la operación.

i) `a.append('e')`
ii) `a.extend(['x', 'y', 'z'])`
iii) `a.pop()`
iv) `a.pop(0)`
v) `a.insert(3, 's')`
vi) `a.remove('x')`
vii) `a.remove('a')`
viii) `a.remove('p')`
ix) `a.sort()`
x) `a.reverse()`

✍ Con `help(list)` obtenemos información sobre éstas y otras operaciones de listas. ¶

Las operaciones como `append`, `extend`, `pop`, `insert`, `remove`, `sort` y `reverse` 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()`). Todos ellos modifican la lista sobre la cual actúan, mientras que el valor de la operación es `None` en general, como puede verse al hacer la asignación `b = a.sort()` (por ejemplo) y comprobar que `b == None`. En cambio, `pop` retorna el objeto extraído.

Ejercicio 8.9. Si `a` es una lista (por ejemplo `a = [1, 2, 3]`), ¿cuál es la diferencia entre `a.reverse()` y `a[::-1]`? ¶

(1) Pero hay excepciones, como en el ejercicio 8.13.

Ejercicio 8.10 (`del`). `del` (por delete o *borrar*) permite eliminar elementos de listas y también se puede usar para eliminar completamente cualquier variable (como si nunca hubiera sido asignada)

Ver los resultados de:

a)
```
a = ['a', 'b', 'c']
del a[1]
a
```

b)
```
a = ['a', 'b', 'c', 'd']
del a[1:3]
a
```

c)
```
a = ['a', 'b', 'c']
del a[:]
a
```

d)
```
a = ['a', 'b', 'c']
del a
a
```

e)
```
a = 123
del a
a
```

f)
```
a = 'mi mama'
del a
a
```

g) `insert` (en cualquier lugar) y `append` (al final) agregan elementos a una lista, y podemos pensar que `pop` y `del` (para índices) y `remove` (para valores) son operaciones inversas. El apartado *b)* sugiere una operación inversa a `extend`, que agrega toda una lista y no sólo elementos. Si `b` es una lista de longitud n, podemos deshacer `a.extend(b)` tomando `del a[-n:]`.

Evaluar:

```
a = [1, 2, 3, 4]
b = [5, 6, 7]
n = len(b)
a.extend(b)
a
del a[-n:]
a
```

¶

Ejercicio 8.11. La mutabilidad hace que debamos ser cuidadosos cuando las listas son argumentos de funciones.

Por ejemplo, definamos

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

Poniendo

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

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

Ejercicio 8.12. Si queremos trabajar con una copia de la lista `a`, podemos poner `b = a[:]`. Evaluar

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

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

Ejercicio 8.13. En realidad la cosa no es tan sencilla como sugiere el ejercicio 8.12. 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 8.6 y 8.11.

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     # ¡estamos modificando una tupla!
b
```

c) En fin, ya que estamos modificando tuplas (inmutables ellas):

```
a = [1, 2]
b = (3, a)  # b es tupla
a[-1] = 4   # a es elemento de b que modificamos
b           # se modifica a -> se modifica b
del a[-1]
b           # se modifica a -> se modifica b
del a
b           # pero no siempre
```

El comportamiento en el apartado *a)* es particularmente fastidioso cuando miramos a matrices como listas (ver ejercicio 10.21), 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*). Eventualmente, una solución recursiva y casera puede hacerse después de ver el capítulo 17. ¶

Ejercicio 8.14 (problemas con asignaciones múltiples). Vimos que las asignaciones múltiples e intercambio son convenientes (ejercicios 8.3 y 8.4), 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. ¶

8.4. Rangos (`range`)

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 lista *virtual*, sus elementos no existen hasta ser requeridos. Como con las cadenas y a diferencia de tuplas y listas, los elementos de `range` son siempre del mismo tipo: enteros.

`range` tiene tres argumentos, que se usan en forma similar a las secciones en el ejercicio 8.2. Con un único argumento se interpreta que el primero es 0 y no se ha explicitado, y si hay tres argumentos el último se interpreta como el *paso* o *incremento* de la sucesión (y si no está se toma como 1). 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 (el tercer argumento no puede ser 0).

Veamos algunos ejemplos.

Ejercicio 8.15 (`range`).

a) Usando `help`, encontrar qué hace `range`.

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

↠ `range` *es una sucesión* virtual *de enteros.*

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 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])` ¶

8.5. Operaciones comunes

Vimos que índices y secciones son operaciones que se pueden realizar en todas las sucesiones. Veamos algunas más.

Ejercicio 8.16 (`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)` ¶

Ejercicio 8.17 (subcadenas). A diferencia del comportamiento en otras sucesiones, `in` en cadenas nos permite decidir si una cadena es parte de otra, es decir, si es una *subcadena.*

a) Ver los resultados de:

i) `'ma' in 'mi mama'` *ii*) `'mm' in 'mi mama'`
iii) `'pa' in 'mi mama'` *iv*) `'mi mama' in 'a'`

b) Estudiar las diferencias entre:

i) `'1' in '123'` *ii*) `1 in '123'` *iii*) `1 in 123` ¶

Ejercicio 8.18 (`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.

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

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

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

c) Como se puede ver, la letra «p» no está en `a`.

i) Comprobarlo preguntando `'p' in a`.

ii) Ver el resultado de `a.index('p')`.

↠ *Preguntando la posición con* `index` *da error si el elemento no está en la sucesión.*

iii) Ver el resultado de `a.count('p')`.

☞ *Si el elemento no está en la sucesión, la cantidad de veces que está es cero (!!!).*

d) Como ya vimos en el ejercicio 8.17, en cadenas (pero no en otras sucesiones) podemos preguntar si una es subcadena de otra.

Ver los resultados de

i) `'ma' in a` *ii*) `a.index('ma')` *iii*) `a.count('ma')`
iv) `'pa' in a` *v*) `a.index('pa')` *vi*) `a.count('pa')` ¶

Ejercicio 8.19. La concatenación que vimos en el ejercicio 3.22 con `+`, se extiende a tuplas y listas.

a) Ver el resultado de:

i) `[1, 2] + [3, 4]` *ii*) `(1, 2) + (3, 4)`
iii) `[1, 2] + (3, 4)`

b) ¿Qué diferencias hay entre poner `[1, 2].extend([3, 4])` y poner `[1, 2] + [3, 4]`? ¶

Ejercicio 8.20 (máximos y mínimos). Otra operación común es la de encontrar máximos y mínimos (mientras tenga sentido). Evaluar

a) `max([1, 2, 3])` *b*) `min((1, 2, 3))`
c) `min(1, 2, 3)` *d*) `max('mi mama')`
e) `min('mi mama')` *f*) `max([1, 'a'])` ¶

Ejercicio 8.21 (cambiando el tipo de sucesión). Es posible cambiar el tipo de una sucesión a otra, dentro de ciertas restricciones.

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

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

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

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

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

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

e) Supongamos `a = 'Ana Luisa'`.

i) Encontrar `b = list(a)`.

ii) Conjeturar el valor de `str(b)` y verificar la respuesta.

iii) Para volver a la cadena original `a` a partir de `b`, podemos usar el método `join` de cadenas: evaluar `''.join(b)` (donde `''` es la cadena vacía).

☞ *No hay problemas en convertir una cadena a, por ejemplo, lista. Para volver para atrás y recuperar la cadena se puede usar el método* `join`. ¶

Ejercicio 8.22 (`isinstance` II). Con `isinstance` (ejercicio 3.25) 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`

b) La última respuesta del apartado anterior no es satisfactoria si queremos que los valores lógicos no sean considerados como enteros (ver el ejercicio 6.4). Agregar una condición con `and` para descartar esa posibilidad.

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

Ejercicio 8.23 (cifras II). En el ejercicio 5.7 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)
c = len(b)
c
```

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

Proponer un método para encontrar la cantidad de cifras 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 876^{123} y comparar con el resultado del ejercicio 5.7. ¶

8.6. Comentarios

- El concepto de lista es distinto del de *arreglo* que se usa en otros lenguajes y que no está implementado en Python.

  La estructura de arreglo (listas con objetos del mismo tipo consecutivos en memoria) es de suma importancia para cálculos científicos de gran tamaño. 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 9

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.

> *La palabra* algoritmo *deriva del nombre de Abu Ja'far Muhammad ibn Musa al-Khwarizmi', quien presumiblemente nació alrededor de 780 en Bagdad (Irak), cuando Harun al-Rashid —el califa de* Las mil y una noches— *comenzaba su califato, y murió en 850.*
>
> *Al-Khwarizmi' escribió y tradujo varios textos científicos (de matemática, geografía, etc.) del griego al árabe. El más importante de ellos es* Hisab al-jabr w'al-muqabala, *de donde surge la palabra* álgebra *con la que ahora designamos a esa rama de la matemática.*
>
> *También escribió un tratado sobre números indo-arábigos que se ha perdido, pero se conservó una traducción al latín llamada* Algoritmi de numero Indorum, *para indicar su autor, dando lugar a la palabra* algoritmo.

Los lenguajes de programación tienen distintas estructuras de control, pero casi siempre tienen estructuras similares a las de `if` y `while` que estudiamos en este capítulo, y con las cuales se pueden simular las otras estructuras.

9.1. `if`

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** bajar 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** tomar el colectivo **en otro caso** tomar 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  # bajar 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          # tomar el colectivo
else:                   # en otro caso
    hacer otra cosa     # tomar 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, si hubiera 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 sencillos.

Ejercicio 9.1. Los siguientes ejemplos usan la estructura `if... else...`, donde se espera que el argumento sea un número (entero o decimal).

En todo los casos estudiar la construcción y probar la función con distintos argumentos, numéricos y no numéricos.

a) La función `espositivo` (en el módulo *ifwhile*) determina si el argumento es positivo o no:

✍ Si queremos que la función retorne valores lógicos en vez de imprimir, podríamos poner simplemente `return x > 0`, sin necesidad de usar `if`.

b) La función `esentero` (en el módulo *ifwhile*) es una variante de la del ejercicio 6.4 para determinar si el argumento es entero o decimal. Acá imprimimos también un cartel acorde. A diferencia del apartado anterior, ahora el resultado es verdadero o falso.

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 es equivalente a:

```
if (not isinstance(x, bool)) and (int(x) == x):
    print(x, 'es entero')
    return True
print(x, 'no es entero')
return False
```

¶

Ejercicio 9.2 (piso y techo). Recordando las definiciones de *piso* y *techo* (ver el ejercicio 5.6), la función `pisotecho` del módulo *ifwhile* imprime los valores correspondientes al piso y el techo de un número real usando una estructura `if... elif... else`.

a) Estudiar las instrucciones de la función.

b) Comparar los resultados de `pisotecho` con los de las funciones `math.floor` y `math.ceil` para ± 1, ± 2.3 y ± 5.6.

c) Cambiando `print` por `return` (en lugares apropiados), modificar `pisotecho` de modo de retornar una tupla `(a, b)`, donde `a` es el piso y `b` es el techo del argumento. ¶

Ejercicio 9.3 (signo de un número real). La función signo 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 `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 9.4 (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 en 1582, pero no todos los países lo adoptaron inmediatamente. En el ejercicio adoptamos este criterio para cualquier año, anterior o posterior a 1582. ¶

Ejercicio 9.5. 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) Hacer 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: usar `round` (hay varias posibilidades).

d) Usando el módulo *grpc*, 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.

e) En el gráfico anterior se puede observar que la ganancia neta cuando se gana \$100 000 es mayor que se gana un poco más de \$100 000.

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

9.2. `while`

La estructura `while` permite realizar una misma tarea varias veces. Junto con `for` —que veremos más adelante— reciben el nombre común de *lazos* o *bucles*.

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

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 9.1: Prueba de escritorio para el ejercicio 9.6.

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 en el ejercicio 15.4, 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 9.6. 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) 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 9.1, 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 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 —bastante más primitiva— de entender el comportamiento y eventualmente encontrar errores, es probarlo con distintas entradas, como hemos hecho hasta ahora.

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 correr la función!).

f) Modificar la función para comparar el valor obtenido con el resultado de la operación `a % b`.

g) 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.

h) 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 obtenido en el apartado *g*)?, ¿podrías explicar por qué?

i) Averiguar qué hace la función `divmod` y comparar el resultado de `divmod(a, b)` con los valores de `c` y `r`. ¶

Ejercicio 9.7 (cifras III). En los ejercicios 5.7 y 8.23 vimos distintas formas para encontrar las cifras de un número entero positivo.

Otra 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 9.6.

Informalmente pondríamos:

$c = 0$
repetir:
  $c = c + 1$
  $n = n//10$
hasta que n sea 0

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, pero debe tenerse cuidado:

> `break` *sólo sale del lazo que lo encierra, y hay que tomar otros recaudos cuando hay lazos anidados.*

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) En los ejercicios 5.7 y 8.23 encontramos la cantidad de cifras en base 10 de 876^{123}. Ver que esos resultados coinciden con el obtenido al usar `cifras`.

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).*

✍ `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`). ¶

Ejercicio 9.8 (números romanos). Recordemos que las letras usadas para números romanos son I, V, X, L, C, D, M, correspondientes respectivamente a 1, 5, 10, 50, 100, 500, 1000.

a) Desarrollar una función que, tomando como entrada un número natural entre 1 y 3000, lo escriba en romano. Por ejemplo, si el argumento es `2999` se debe obtener `'MMCMXCIX'`.

Sugerencia: primero hacerlo para casos chicos, por ejemplo entre 1 y 10, y después ir agregando posibilidades.

✍ El problema es tedioso para escribir, pero es típico de muchas aplicaciones, en las que no hay atajos o donde perdemos más tiempo tratando de encontrarlo que en escribir el programa.

✍ La función se puede hacer usando sólo `if`, pero al usar también `while` se pueden ahorrar algunos renglones.

b) Hacer una función que ingresando un número expresado como cadena de caracteres en romano, lo pase a la escritura en base 10.

Sugerencia: Encontrar la lista de caracteres y comenzar desde atrás.

✍ Es común que al hacer la operación inversa se proceda revirtiendo el orden de los pasos.

Por ejemplo, si dado x se calcula $y = 2x + 3$, lo último que hicimos fue sumar 3, y será lo primero que hagamos para calcular x en términos de y. En definitiva obtenemos $x = (y - 3)/2$.

Para pasar un número de base 10 a escritura romana, vamos considerando las cifras más significativas de izquierda a derecha. Cuando queremos pasar de romano a base 10, es natural empezar de derecha a izquierda, con las cifras menos significativas. ¶

Capítulo 10

Recorriendo sucesiones

Muchas veces queremos 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 para cada elemento. Como estas acciones son tan comunes, Python tiene funciones predefinidas para esto, y para contar la cantidad de elementos pondríamos simplemente `len(b)`.

Sin embargo, habrá veces que no hay funciones predefinidas o cuesta más encontrarlas que hacerlas directamente. Para estos casos, Python cuenta con la estructura de repetición `for` que estudiamos en este capítulo.

10.1. Uso básico de `for`

En su uso más simple (y quizás más común en otros lenguajes), `for` se aplica con un esquema del tipo

```
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) o un archivo (que veremos en el capítulo 11).

✍ Recordar que las secuencias tienen un orden que se puede obtener mirando los índices, como hicimos al principio del capítulo 8.

✍ Las similitudes con el uso de `in` que vimos en el ejercicio 8.16 son intencionales.

Veamos algunos ejemplos sencillos.

Ejercicio 10.1. Consideremos `a = [1, 2, 3]`.

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) La variable que aparece inmediatamente después de `for` puede tener cualquier nombre (identificador), pero hay que tener cuidado porque es global. Comprobar que el valor de `x` cambia:

```
x = 5   # ponemos un valor que no está en a
for x in a:
    print(x)
x       # ahora x es el último elemento en a
```

c) Si queremos encontrar la cantidad de elementos, como en `len(a)`, imitando lo hecho en el ejercicio 9.6.*g*), llamamos `long` al contador y ponemos

```
long = 0
for x in a:
    long = long + 1  # la acción no depende de x
long
```

d) Repetir los apartados *a*) y *c*) cuando `a` es

i) `[1, [2, 3], 4]` *ii*) `'pepe'` *iii*) `['Ana', 'Luisa']` ¶

Podemos copiar la idea del contador `k` en el ejercicio 10.1.*c*) para sumar los elementos de `a`. Veamos esto con más detalle.

Para encontrar la suma de los elementos de

```
a = [1, 2.1, 3.5, -4.7]
```

podemos hacer las sumas sucesivas

$$1+2.1=3.1, \quad 3.1+3.5=6.6, \quad 6.6-4.7=1.9. \tag{10.2}$$

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 (10.2), y puede escribirse con un lazo `for` como:

```
s = 0
for x in a:     # s será sucesivamente
    s = s + x   #    1, 3.1, 6.6 y 1.9
s
```
(10.3)

Ejercicio 10.2 (sumas y promedios). Copiando el esquema en (10.3), 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 lista 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. ¶

Ejercicio 10.3. Volviendo a mirar las ecuaciones en (10.2), las instrucciones en (10.3) y el ejercicio 10.2, vemos que podríamos poner una única función para sumar números o cadenas si ponemos el primer elemento de la sucesión como valor inicial del acumulador.

El tema sería definir un valor conveniente para la suma de sucesiones vacías como `''` o `[]`. Arbitrariamente decidimos que en ese caso pondremos un cartel avisando de la situación y retornaremos `None`.

Tendríamos algo como:

```
if len(a) == 0:      # nada para sumar
    print('*** Atención: Sucesión vacía')
    return           # nos vamos
s = a[0]             # el primer elemento de a
for x in a[1:]:      # para c/u de los restantes...
    s = s + x
return s
```

Definir una nueva función `suma` 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)` ¶

Los últimos apartados del ejercicio 10.3 muestran que no tenemos que tener una sucesión explícita para usar `for`, pudiendo usar `range`. Veamos otros ejemplos de esta situación.

Ejercicio 10.4 (suma de impares I). Si queremos encontrar la suma de los n primeros impares positivos, no es necesario generar una lista (o tupla) explícita y podemos poner directamente

```
s = 0
for x in range(1, 2*n, 2):
    # x = 1, 3,..., 2*n - 1
    s = s + x
s
```
(10.4) o

bien

```
s = 0
for x in range(n):
    s = s + 2*x + 1
s
```
(10.5)

a) Evaluar las instrucciones en (10.4) y en (10.5) para $n = 5$, $n = 10$ y $n = 0$, comprobando que los resultados son correctos.

b) ¿Cuál será el valor de cada una de las expresiones cuando `n` es entero negativo?, ¿y si `n` es decimal?

c) Usando (10.4) o (10.5), construir una función `f` tal que `f(n)` dé como resultado la suma de los n primeros impares positivos, dando `0` (cero) si $n \notin \mathbb{N}$.
Sugerencia: las funciones del ejercicio 6.4 pueden ayudar (... o no). ¶

10.2. Listas por comprensión usando `for`

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]
```
(10.6)

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
```
(10.7)

La estructura en (10.6) tiene similitudes con la estructura en (10.7), y no ahorra mucho en escritura pero sí tal vez 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.
- `iterable` tiene que estar definido, en caso contrario obtenemos un error.

Ejercicio 10.5.

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.

☞ *En esta forma de usar la definición por comprensión (más adelante veremos otras), la longitud del resultado es la misma que la original.*

b) Repetir el apartado anterior poniendo tuplas en vez de listas (es decir, poniendo `a = (1, 2)`, etc.), viendo si da error. ¶

Ejercicio 10.6. En el ejercicio 10.1.*b*) vimos que la variable `x` usada en `for` era global. En cambio, las variables dentro de la definición por comprensión se comportan como variables locales.

Comprobarlo poniendo

```
x = 0
a = [1, 2, 3]
b = [x + 1 for x in a]
x
```
¶

Ejercicio 10.7. Resolver los siguientes apartados si `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 10.8. Poniendo `a = 'Mateo Adolfo'`, calcular `[2*x for x in a]`. ¶

Ejercicio 10.9. Construir una función `cifraslista` que dado un número entero construya la lista de cifras (en base 10) del número. Por ejemplo,

Ingresando	debe dar
`cifraslista(123)`	`[1, 2, 3]`
`cifraslista(-123)`	`[1, 2, 3]`
`cifraslista(0)`	`[0]`

Aclaración: los elementos de la lista construida deben ser enteros no negativos.
Sugerencia: Recordar `abs`, el ejercicio 8.23 y hacer una lista por comprensión. ¶

Ejercicio 10.10. Poniendo

```
a = [1, 2, 3]
b = [4, 5]
```

encontrar las diferencias de valores y longitud entre `c`, `d` y `e` si

```
c = [x*y for x in a for y in b]
d = [x*y for x in b for y in a]
e = [[x*y for x in a] for y in b]
```
¶

10.3. Filtros

Las listas por comprensión nos permiten fabricar listas a partir de sucesiones, pero podríamos tratar de poner en la nueva lista sólo los elementos que satisfacen cierta condición, digamos `p(x)` (que debe tener valores lógicos).

Así, el esquema en (10.7) puede ponerse como

```
lista = []
for x in iterable:
    if p(x):                                    (10.8)
        lista.append(f(x))
lista
```

Correspondientemente, el esquema en (10.6) se transforma en

```
[f(x) for x in iterable if p(x)]                (10.9)
```

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 10.11. Dando definiciones por comprensión con filtros, encontrar:

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

b) Si `a = 'mi mama me mima'`, la letra `m` aparece en las posiciones 0, 3, 5, 8, 11 y 13. Usar listas con filtros para encontrar la lista de `i` para los que `a[i]` es:

i) `'a'` *ii*) `'i'` *iii*) `'A'`

c) Las palabras terminadas en «o» en

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

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

✍ La estructura de listas por comprensión y la de filtro no está disponible en lenguajes como Pascal o C.

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.

- También los lenguajes *declarativos* como SQL tienen filtros.
- Python tiene las funciones `map` y `filter` con características similares a las descriptas (ver `help(map)` o `help(filter)`). `zip` (ver ejercicio 10.22) es otra estructura de programación funcional disponible en Python. No veremos estas estructuras en el curso, usando en cambio listas por comprensión con filtros.

- Python tiene módulos estándares para programación funcional, y es interesante un documento sobre el tema en la página de Python. El mismo Guido van Rossum, creador de Python, ha hecho una propuesta para eliminar aspectos de programación funcional.

10.4. Aplicaciones

Ejercicio 10.12 (suma de Gauss). Para $n \in \mathbb{N}$ consideremos la suma

$$s_n = 1+2+3+\cdots+n = \sum_{i=1}^{n} i, \tag{10.10}$$

que, según la fórmula de Gauss, es

$$s_n = \frac{n \times (n+1)}{2}. \tag{10.11}$$

a) Usando un lazo `for` y `range`, construir una función `gauss1` a partir de la ecuación (10.10), de modo que `gauss1(n)` dé la suma de los primeros n naturales.

b) De modo similar, construir una función `gauss2` a partir de la ecuación (10.11).
Aclaración: `gauss2` no debe usar `for`, y es conveniente usar división entera.

c) Verificar que las funciones `gauss1` y `gauss2` coinciden para los primeros 10 valores de n poniendo

```
[gauss1(n) == gauss2(n) for n in range(1, 11)]
```

✍ Otra posibilidad es considerar las listas separadamente:

```
uno = [gauss1(n) for n in range(1, 11)]
dos = [gauss2(n) for n in range(1, 11)]
uno == dos
```

Ver el ejercicio 10.22.

d) Usando inducción, demostrar la ecuación (10.11) a partir de la definición (10.10).

❦ *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 que le valieron el apodo de «príncipe de los matemáticos». Para algunos, fue el más grande matemático de todos los tiempos. ¶

En el ejercicio 10.12.*c*) construimos una lista con valores lógicos. Si esa lista fuera muy larga, sería conveniente obtener una síntesis (como hacemos con la suma o promedio de números) para determinar si todos los valores son verdaderos.

Una posibilidad es considerar un esquema de la forma

```
t = True            # hace las veces de s = 0
for x in lista:
    t = t and x
t
```

Podemos mejorarla un poquitín (de paso repaso):

- `t and x` primero evaluará `t` y luego `x`. Si `t` ya es verdadero, es más razonable considerar `x and t`, por el tema de los «cortocircuitos» que vimos en el ejercicio 3.17.
- Como una vez que `t` sea falso seguirá siendo falso, podemos salir inmediatamente del lazo `for` con `break` (recordar el ejercicio 9.7).

El esquema entonces toma la forma

```
t = True
for x in lista:
    if not x:
        t = False
        break   # salir del lazo
return t
```
(10.12)

Ejercicio 10.13. Si pusiéramos el esquema (10.12) dentro de una función, podríamos cambiar `break` por `return` y eliminar completamente `t`, quedando:

```
for x in lista:
    if not x:
        return False
return True
```
(10.13)

a) Usando (10.13), definir una función `todos`, que aplicada a una lista de valores lógicos da verdadero si todos los elementos de la lista son verdaderos, y falso en otro caso.

Conjeturar primero y luego verificar el resultado cuando las listas son:

i) `[True, False, True]` *ii*) `[True, True, True]`
iii) `[False, False, False]` *iv*) `[]`

b) Usar `todos` para verificar que las funciones `gauss1` y `gauss2` (en el ejercicio 10.12) coinciden para n entre 1 y 100.

c) Tomando como punto de partida la función `todos`, definir una función `alguno` tal que dada una lista de valores lógicos decida si alguno de ellos es verdadero.

Luego repetir las listas del apartado *a*) reemplazando `todos` por `alguno` (primero conjeturar y después verificar).

d) Python tiene las funciones `all` y `any`. Averiguar qué hacen estas funciones y comparar su uso con el de `todos` y `alguno`.

e) En cualquier caso, tanto `todos` y `alguno` como `all` y `any` dan resultados curiosos cuando los elementos de la lista no son valores lógicos, como en `todos([1, 2])` o `all([1, 2])` (ver la sección 3.6).

f) Para eliminar los inconvenientes del apartado anterior, construir una función `todossonbool` que cuando aplicada a una lista determine si todos los elementos de la lista son lógicos (dando verdadero) o si hay alguno que no lo es (retornando falso).

Sugerencia: usar las ideas de `todos` y de `esbool` (ejercicio 6.4). ¶

Ejercicio 10.14 (suma de impares II). En el ejercicio 10.4 construimos la función `f` para encontrar la suma de los primeros n impares $(1,3,5,\ldots)$.

a) Python permite el uso de `sum` (ejercicio 10.2) en la forma `sum(2*i + 1 for i in range(n))`, como si pusiéramos

```
sum([2*i + 1 for i in range(n)])
```

Verificar esta construcción.

b) Construyendo una lista con los 10 primeros valores de la suma con

```
[f(n) for n in range(1, 11)]
```

conjeturar una fórmula explícita para `f(n)` y luego probar la conjetura usando inducción.

Ayuda: Comparar con `[n**2 for n in range(1, 11)]`. ¶

Ejercicio 10.15 (factorial). Recordemos que para $n \in \mathbb{N}$, el factorial se define por

$$n! = 1 \times 2 \times \cdots \times n. \tag{10.14}$$

a) Construir una función `factorial` tal que `factorial(n)` dé como resultado $n!$ usando la ecuación (10.14).

Sugerencia: usar un esquema similar a

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

☞ *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.

✍ Esto es consistente con las propiedades de la exponencial (e^x) y su inversa, el logaritmo ($\log x$). Si queremos evaluar $p = \prod_i a_i$, tomando logaritmos, debemos evaluar

$$s = \log p = \sum_i \log a_i,$$

y luego podemos obtener p tomando $p = e^s$. Para calcular s ponemos inicialmente su acumulador en 0, y el acumulador para el producto debe tener el valor inicial $e^0 = 1$.

Si $a = (a_1, a_2, \ldots, a_n)$ son los elementos a multiplicar y los logaritmos correspondientes son $b = (\log a_1, \ldots, \log a_n)$, los esquemas en paralelo serían:

```
p = 1                   s = 0               # s = log p
for x in a:             for y in b:         # y = log x
    p = p * x               s = s + y       # <-> p = p * x
p                       s                   # s = log p
```

b) Usando la función `all` (ejercicio 10.13), comparar los valores de `factorial(n)` y `math.factorial(n)` para $n = 1, \ldots, 10$.

c) 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 con $n = 10, 100, 1000$, comparando los resultados con `factorial`. ¶

Ejercicio 10.16. El teorema del binomio expresa que

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}, \tag{10.15}$$

donde

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k!} \tag{10.16}$$

son los coeficientes binomiales ($0 \le k \le n$).

Hacer el cálculo de los tres factoriales en el medio de las igualdades en (10.16) es ineficiente, y es mejor hacer la división entera a la derecha de esas igualdades, que 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 (10.16).

Agregar un `if` a la función del apartado anterior para usar las operaciones a la derecha de (10.16) sólo cuando $k \le n/2$. ¶

Ejercicio 10.17 (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

$$s_k = \sum_{i=1}^{k} a_i. \tag{10.17}$$

✍ Ponemos las listas a y s como vectores para indicar que el orden es importante.

Esto se parece mucho a las ecuaciones (10.2) y podemos combinar los esquemas (10.3) y (10.7) para obtener, suponiendo que la lista original es `a`:

```
suma = 0
solucion = []
for x in a:
    suma = suma + x
    solucion.append(suma)
solucion
```
(10.18)

a) Hacer una función `acumuladas` siguiendo el esquema (10.18), y aplicarla para calcular las acumuladas de los impares entre 1 y 19: ¿cuánto debe dar?
Ayuda: recordar el ejercicio 10.14.

💣 *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) Otra posibilidad es primero copiar la lista, e ir modificando la nueva lista:

```
solucion = a[:]
for i in range(1, len(a)):
    solucion[i] = solucion[i-1] + solucion[i]
solucion
```

d) A partir de (10.17), podemos poner

$$s_1 = a_1, \qquad s_k = s_{k-1} + a_k \quad \text{para } k > 1, \tag{10.19}$$

y de aquí que

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

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 (10.17) y (10.19) empiezan con 1, pero Python pone los índices a partir de 0. ¶

Ejercicio 10.18. En 1202 Fibonacci propuso el siguiente problema en su libro *Liber Abaci*:

> *¿Cuántos pares de conejos se producen a partir de una única pareja en un año si cada mes cada par de conejos da nacimiento a un nuevo par, el que después del segundo mes se reproduce, y no hay muertes?*

Resolver el problema (con lápiz y papel).

❦ *Leonardo Bigollo es el verdadero nombre de Leonardo de Pisa (1180–1250), también conocido como Fibonacci (contracción de las palabras «hijo de Bonacci»). Además de «Liber Abaci», publicó numerosos tratados sobre teoría de números, geometría y la relación entre ellos. Sin embargo, fue prácticamente ignorado durante la Edad Media, apareciendo sus resultados nuevamente publicados unos 300 años después.* ¶

Ejercicio 10.19 (números de Fibonacci I). Los números de Fibonacci f_n están definidos por

$$f_1 = 1, \qquad f_2 = 1, \qquad \text{y} \qquad f_n = f_{n-1} + f_{n-2} \quad \text{para } n \geq 3, \tag{10.20}$$

obteniendo la sucesión $1, 1, 2, 3, 5, 8, \ldots$

✍ Hay cierta similitud entre (10.20) y (10.19), y en especial con el ejercicio 10.17.*c*).

Una forma de construir f_n con Python usando las igualdades (10.20) es construir una lista con los n primeros números de Fibonacci

a) Construir una función `listafib` con el siguiente esquema:

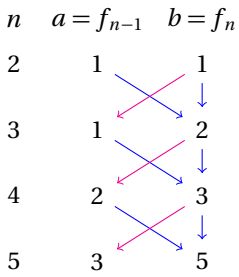


Figura 10.1: Ilustración del algoritmo para calcular los números de Fibonacci.

```
lista = [1, 1]            # f1 y f2
for k in range(2, n):     # para k = 2,...
    (a, b) = lista[-2:]   # tomamos los dos últimos
    lista.append(a + b)   # los sumamos y
                          # agregamos a la lista
return lista
```

b) ¿Es correcta la función? Probarla para distintos valores de n.

c) ¿Qué pasa si $n = 1$?, ¿qué longitud debería tener la lista?, ¿cómo se puede modificar el valor retornado?

d) ¿Cómo podría obtenerse únicamente el n-ésimo número de Fibonacci a partir de la lista (y no toda la lista)?

e) ¿Qué pasa si cambiamos `range(2, n)` por `range(n-2)`? ¶

Ejercicio 10.20 (números de Fibonacci II). En general no nos interesa obtener la lista de los números de Fibonacci, sino sólo el último. Mantener toda la lista parece un desperdicio, ya que sólo necesitamos los dos últimos valores (`a` y `b` en la definición de `listafib` en el ejercicio anterior).

Si llamamos, justamente, `a` y `b` a los dos últimos valores, los próximos dos últimos serán `b` y `a + b`, como se ilustra en la figura 10.1.

Entonces podemos poner el bloque de la función para calcular f_n como:

```
a = 1
b = 1
for k in range(n-2):
    a, b = b, a + b
return b
```

✍ Observar el uso de intercambio/asignaciones múltiples en la construcción `a, b = b, a + b`.

a) Definir una función `fibonacci(n)` para calcular f_n siguiendo esta idea.

b) Usando una lista por comprensión con `fibonacci` y `range`, encontrar los primeros 10 valores de f_n usando `fibonacci` y compararla con `listafib(10)`.

c) La *fórmula de Euler-Binet* establece que

$$f_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}. \tag{10.21}$$

✍ Créase o no, el miembro derecho es un número entero. ¿Podrías decir por qué es un número entero si no supieras que es f_n?

Construir una función `binet` para calcular f_n según (10.21), y comparar con los valores de `fibonacci` para $n = 1, \ldots, 10$.

Sugerencia: usar división común y no entera.

d) Según la fórmula (10.21), f_n crece *exponencialmente*, por lo que rápidamente toma valores muy grandes.

Comparar el número de Fibonacci f_n con el redondeo (`round`) de

$$\frac{(1+\sqrt{5})^n}{2^n\sqrt{5}}.$$

¿A partir de qué n son iguales?, ¿podrías decir por qué?

e) ¿Cuántos dígitos tiene f_{345}?
Respuesta: 72.

f) Podríamos haber empezado tomando $f_0 = 0$, $f_1 = 1$, y luego usar la ecuación a la derecha en (10.21). En otras palabras, podríamos ir hacia atrás, usando $f_{n-2} = f_n - f_{n-1}$. Construir una función apropiada para encontrar f_n cuando n es entero no positivo, y encontrar los valores de f_n para $n = 0, -1, \ldots, -10$. ¿Son razonables los valores obtenidos?

✍ El módulo *fibonacci* tiene versiones de las funciones `listafib` y `fibonacci`.

❦ *Jacques Binet (1786–1856) publicó la fórmula para los números de Fibonacci en 1843, pero ya había sido publicada por Leonhard Euler (1707–1783) en 1765, y de ahí el nombre de la relación. Sin embargo, A. de Moivre (1667–1754) ya la había publicado (y en forma más general) en 1730.*

❦ *Mucho después de Fibonacci, se observó que los números f_n aparecen en muy diversos contextos, algunos insospechados como en la forma de las flores del girasol, y son de importancia tanto en las aplicaciones prácticas como teóricas. Por ejemplo, han sido usados para resolver problemas de confiabilidad de comunicaciones.*

En cuanto a aplicaciones teóricas, vale la pena mencionar que en el Congreso Internacional de Matemáticas de 1900, David Hilbert propuso una serie de problemas que consideró de importancia para resolver durante el siglo XX. Entre ellos, el décimo pide encontrar un algoritmo para determinar si un polinomio con coeficientes enteros, arbitrariamente prescripto, tiene raíces enteras (resolver la ecuación diofántica asociada). Recién en 1970 pudo obtenerse una respuesta al problema, cuando Yuri Matijasevich (quien tenía 22 años) demostró que el problema es irresoluble, es decir, no existe algoritmo que pueda determinar si una ecuación diofántica polinomial arbitraria tiene soluciones enteras. En su demostración, Matijasevich usa la tasa de crecimiento de los números de Fibonacci. ❡

Ejercicio 10.21 (matrices como listas I). A veces es conveniente escribir una matriz como una lista de listas. Por ejemplo, la matriz $A \in \mathbb{R}^{2\times 3}$ dada por

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{10.22}$$

puede representarse en Python como

```
a = [[1, 2, 3], [4, 5, 6]]
```
(10.23)

es decir, una lista donde cada elemento corresponde a una fila de la matriz original.

a) ¿Cuántos elementos tienen A y `a`?, ¿coinciden?

b) ¿Cuál es el resultado de `len(a)`?

c) Ver que con `a[0]` obtenemos la primera fila de `a`.

d) ¿Cómo se puede obtener la entrada 5 de A usando `a` con Python?

e) Un intento para encontrar la primera columna de `a` como vector es poner `a[:][0]`. Ver el resultado de esta operación y decir por qué no funciona.
Encontrar ahora la columna usando una lista por comprensión.

f) Comprobar que las sumas por columnas pueden obtenerse como

```
sumacols = [a[0][j] + a[1][j] for j in [0, 1, 2]]
```

¿Cómo podrían obtenerse las sumas por filas?

g) ¿Podrían hacerse los apartados anteriores cambiando listas por tuplas, es decir, poniendo `a = ((1, 2, 3), (4, 5, 6))`? ❡

Ejercicio 10.22 (Matrices como listas II). Supongamos que representamos la matriz A de dimensiones $m \times n$,

$$A = \left[a_{ij}\right] = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \tag{10.24}$$

como en el ejercicio 10.21, haciendo una lista `a` de m elementos, cada uno de los cuales a su vez es una lista de longitud n.

a) Construir una función `dimensiones` que dada la matriz `a` retorna sus dimensiones en una tupla `(número de filas, número de columnas)`. Por ejemplo, para la matriz en (10.23) debe retornar `(2, 3)`.

b) La *transpuesta* de la matriz A en (10.24) es

$$A^{\mathsf{T}} = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & & \ddots & \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix},$$

que tiene dimensiones $n \times m$, y sus filas son las columnas de A (y recíprocamente). Por ejemplo, si A es la matriz en (10.22), entonces

$$A^{\mathsf{T}} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

Usando listas por comprensión y eventualmente `range`, construir una función `transpuesta` que tomando como argumentos una matriz construya su transpuesta. Así, si `a` está definida como en (10.23), `transpuesta(a)` debe ser

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

Ayuda: La columna `j` está formada por las entradas `f[j]` donde `f` recorre las filas de `a`, y `j` varía entre `0` y `len(a) - 1`.

✍ Python tiene la función `zip` que hace una tarea similar, pero no la veremos en el curso.

✍ Usualmente pensamos que las entradas de una matriz son numéricas, pero podrían ser de otro tipo. En el ejercicio 10.12.*c*) y la nota allí, surge la curiosidad de ver cómo pasar de una lista cuyos elementos son las tuplas `(gauss1(n), gauss2(n))`, al par de listas formadas por `gauss1` y `gauss2`: matemáticamente lo que estamos pidiendo es encontrar la transpuesta de una matriz de entradas no numéricas y dimensión $m \times 2$.

c) Construir una función `sumafilas` que cuando aplicada a la matriz `a` retorne la suma de sus filas (si la matriz tiene dimensión $m \times n$, `sumafilas` es una lista de longitud m).
Ayuda: usar la función `suma` (ejercicio 10.2) o directamente `sum`.

d) Análogamente, construir una función `sumacols` para encontrar la suma de las columnas de una matriz.
Ayuda: ver el ejercicio 10.21 y/o el apartado *b*). ¶

Capítulo 11

Formatos y archivos de texto

11.1. Formatos

En esta sección vemos distintas formas de imprimir secuencias, y para recorrerlas usamos `for`.

Ejercicio 11.1 (opciones de `print`). Pongamos `a = 'Mateo Adolfo'` en la terminal.

a) Ver el resultado de

```
for x in a:
    print(x)
```

b) `print` admite un argumento opcional de la forma `end=algo`. Por defecto, omitir esta opción es equivalente a poner `end='\n'`, iniciando un nuevo renglón después de cada instrucción `print`.

Poner

```
for x in a:
    print(x, end='')
```

viendo el resultado.

c) En algunos casos la impresión se junta con `>>>`, en cuyo caso se puede agregar una instrucción final `print('')`. Mejor aún es poner (en un módulo que se ejecutará):

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

¶

Ejercicio 11.2 (construcción de tablas). Muchas veces queremos la información volcada en tablas.

a) Un ejemplo sencillo sería construir los cuadrados de los primeros 10 números naturales. Podríamos intentar poniendo (en un módulo que se ejecutará)

```
for n in range(1, 11):
    print(n, n**2)
```

Probar este bloque, viendo cuáles son los inconvenientes.

b) Uno de los problemas es que nos gustaría tener alineados los valores, y tal vez más espaciados. Se puede mejorar este aspecto con *formatos* para la impresión. Python tiene distintos mecanismos para *formatear*[(1)] y nos restringiremos al que usa llaves «{» y «}».

Veamos cómo funciona con un ejemplo:

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

(1) Sí, es una palabra permitida por la RAE.

El primer juego de llaves tiene dos entradas separadas por «`:`». La primera entrada, `0`, corresponde al índice de la primera entrada después de `format`. La segunda entrada del primer juego de llaves es `2d`, que significa que vamos a escribir un entero (`d`) con 2 lugares 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 usan la especificación `s`, 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 = 1234.567891234
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 11.3. 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]:
```

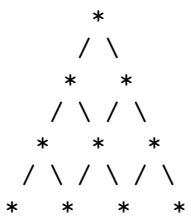


Figura 11.1: «Arbolito de Navidad» con 4 estrellas en la parte de abajo.

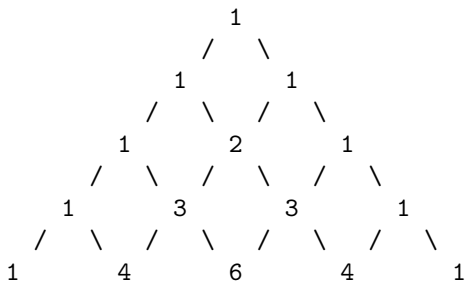


Figura 11.2: Impresión del triángulo de Pascal de nivel 4.

```
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, $p \wedge q$, $p \vee q$.

c) Una de las leyes de de Morgan establece que $\neg(p \wedge q)$ es equivalente a $(\neg p) \vee (\neg q)$, es decir, las tablas de verdad coinciden.

Comprobar esta ley construyendo la tabla con las cuatro columnas p, q, $\neg(p \wedge q)$, $(\neg p) \vee (\neg q)$. ¶

Ejercicio 11.4. Otro ejemplo donde es mejor usar paréntesis (recordar la sección 3.6).

a) Construir tablas con `a` y `b` tomando los valores `False` y `True` para responder las siguientes preguntas.

i) Para Python, ¿`not a == b` es lo mismo que `not (a == b)` o que `(not a) == b`?

Respuesta: ambos son equivalentes.

ii) Si bien `not a == b` tiene sentido para Python, ¿tiene sentido `a == not b`?, ¿cuál es el resultado de Python?

b) En base a los resultados obtenidos, ¿cuál es la precedencia entre `==` y `not` (es decir, cuál se evalúa primero)? ¶

Ejercicio 11.5. Construir una función `navidad(n)` que muestre un «arbolito de Navidad» como el que se muestra en la figura 11.1 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 «casita» en el ejercicio 3.30. ¶

Ejercicio 11.6 (triángulo de Pascal). El triángulo de Pascal se obtiene poniendo un uno en la primera fila, y luego en cada fila siguiente la suma de los elementos consecutivos de la fila anterior, empezando y terminando con 1, como se muestra en la figura 11.2.

La función `pascal` del módulo *pascal* construye los números en las filas del triángulo de Pascal.

a) Estudiar la función, comparándola con la construcción de los números de Fibonacci (ejercicios 10.19 y 10.20).

b) La función da una lista cuyos elementos son listas, la k-ésima fila ($k = 0, \ldots, n$) está formada por los coeficientes binomiales $\binom{k}{i}$ ($0 \le i \le k$). Verificar que esto es cierto comparando la última lista de `pascal(n)` con la lista de coeficientes binomiales $\binom{n}{i}$, según la construcción del ejercicio 10.16.

c) Modificar la función de modo de obtener una figura similar a la de la figura 11.2, donde cada número ocupa 3 lugares. Verificar la corrección tomando $n = 4, 7, 10$.
Ayuda: combinar la función original con el ejercicio 11.5. ¶

11.2. Archivos de texto

Cuando se genera mucha información, como en las tablas, en general no queremos imprimirla en pantalla sino guardarla en un archivo en el disco de la computadora. Para 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...).

✍ Otras 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 5.9): `pepe` es el nombre interno y el nombre externo puede ser `'Ana Luisa'`, `'Mateo Adolfo'` o cualquier otro.

Ejercicio 11.7 (guardando en archivo). Supongamos que queremos guardar una tabla del seno, la primera columna en grados y la segunda con el valor correspondiente del seno como hicimos en el ejercicio 11.2.*e*).

El módulo *tablaseno* hace este trabajo y 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 5.8.*f*).

- `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) se borrará.*

  ✍ Hay formas de verificar si el archivo ya existe y en ese caso no borrarlo, pero no las veremos en el curso.

- 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* en inglés) 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, IDLE) para verificar sus contenidos.

b) Ver el resultado de eliminar `\n` en el argumento de `archivo.write`.

c) Cambiar en todos los casos `archivo` por `salida` y verificar el comportamiento.

d) Cambiar `'tablaseno.txt'` por `'tabla.sal'` y verificar el comportamiento.

e) Cambiar el módulo de modo de preguntar interactivamente el nombre del archivo a escribir.

Ayuda: Recordar `input` (ejercicio 5.9). ¶

Ejercicio 11.8 (lectura de archivos). *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*,[2] ponemos en terminal:

```
archivo = open('santosvega.txt', 'r')
```

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.

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` lee todo el archivo y lo guarda como una 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 cada renglón separadamente, lo que podemos hacer poniendo

```
archivo = open('santosvega.txt', 'r')
for renglon in archivo:
    print(renglon)
```

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

f) El resultado del apartado anterior es que se deja otro renglón en blanco después de cada renglón. Podemos usar la técnica del ejercicio 11.1 poniendo

```
archivo = open('santosvega.txt', 'r')
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

[2] Recordar el ejercicio 5.12.

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

✍ No es una sucesión pero es un *iterable* de Python, como lo son las sucesiones. Veremos algo más sobre el tema en el ejercicio 11.10.

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.*c*).

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 11.7.

Poner

```
archivo = open('santosvega.txt', 'r')
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 11.9. 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). ¶

Ejercicio 11.10 (leyendo tablas). En el ejercicio 5.11 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')
```

donde *tablaseno.txt* es el archivo construido en el ejercicio 11.7.

b) `readline` hace que se lea un renglón del archivo: poner sucesivamente

```
archivo.readline()
archivo.readline()
```

viendo los primeros renglones del archivo.

c) El valor de `archivo.readline()` es una cadena de caracteres, terminada con `\n`, y los números están separados por espacios (por la forma en que construimos el archivo). Poner

```
a = archivo.readline()
```

y comprobar el valor de `a`.

d) Para separar los números en el renglón ponemos

```
a.split()
```

que elimina los espacios entre medio, así como el final de renglón `\n`, dando una lista con las «palabras» de interés.

e) Para verificar el comportamiento de `split`, poner

```
b = 'mi␣␣␣mamá␣me...␣␣␣␣¡mima!'
b.split()
```

donde en `b` ponemos arbitrariamente espacios entremedio.

f) ¿Qué resultado da `''.split()` (sin espacios)?, ¿y con uno o más espacios como `'␣␣'.split()`?

g) Volviendo a `a`, poner

```
g, s = a.split()
```

donde `g` son los grados y `s` el seno correspondiente. Verificar que tanto `g` como `s` son cadenas de caracteres.

h) Finalmente ponemos

```
g = int(g)
s = float(s)
```

para obtener los valores de `g` como entero y de `s` como decimal.

i) Reabriendo el archivo, construir una lista

```
[[g1, s1], [g2, s2],...]
```

de modo que los valores `g1, g2,...` sean enteros y los valores `s1, s2,...` sean decimales.

Sugerencia: empezando con una lista vacía, para cada renglón en el archivo usar `split`, y luego agregar los valores adecuados usando `append`. ¶

Ejercicio 11.11. Consideremos el archivo *santosvega.txt*.

a) Construir una lista `p` de las palabras en el archivo (las palabras pueden incluir los signos de puntuación pero no los finales de renglón).

Hacerlo de dos formas y verificar si se obtienen los mismos resultados:

i) Usando `read` para leer todo el archivo de una vez.

ii) Leyendo renglón por renglón y usando `extend`.

Sugerencia: Empezar con `p = []` y usar `split`.

b) Encontrar la cantidad de palabras encontradas.

c) Usando `print` con segundo argumento `end='␣'` (dejando un espacio) cuando sea adecuado, imprimir el texto en un único renglón.

Ayuda: copiar la estructura en el ejercicio 11.1.*c*).

d) Encontrar las palabras con cinco o más letras en el archivo. ¶

Capítulo 12

Simulación

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.

12.1. Funciones de números aleatorios en Python

Análogamente al 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 12.1. Veamos ejemplos de las funciones en *random* que nos interesan (hay varias más que no veremos).

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 $[0.0, 1.0)$.

```
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)]
```

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

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

e) `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
random.shuffle(range(5))    # da error
```

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

Sugerencia: pasar a lista y usar `''.join(algo)`. ¶

Ejercicio 12.2. Podemos emular el comportamiento de las funciones `random.randint` y `random.choice` usando solamente `random.random`: hacer sendas funciones, digamos `mirandint(a, b)` (donde `a` y `b` son enteros) y `michoice(s)` (donde `s` es una sucesión).

✍ Es bastante más complicado construir una permutación aleatoria emulando el comportamiento de `random.shuffle`.

Por otra parte, poniendo `random.shuffle(lista)[:n]` (modificando `lista`) podemos emular `random.sample(lista, n)`, si bien en forma ineficiente. ¶

12.2. Números aleatorios

Aunque no es nuestra intención aquí 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.

✍ 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 *random*.

✍ Hay distintos algoritmos para generar números aleatorios, y los resultados pueden ser muy distintos según el algoritmo, a diferencia de lo que sucede con el seno o el logaritmo (que pueden calcularse con diferentes algoritmos pero los resultados son similares).

En general los sistemas operativos tienen sus propios generadores, no necesariamente iguales al que usa Python.

✍ Los algoritmos para construir números aleatorios son determinísticos, y como sólo ciertos números pueden representarse en la computadora, los números se repiten 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!

✍ 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.

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,

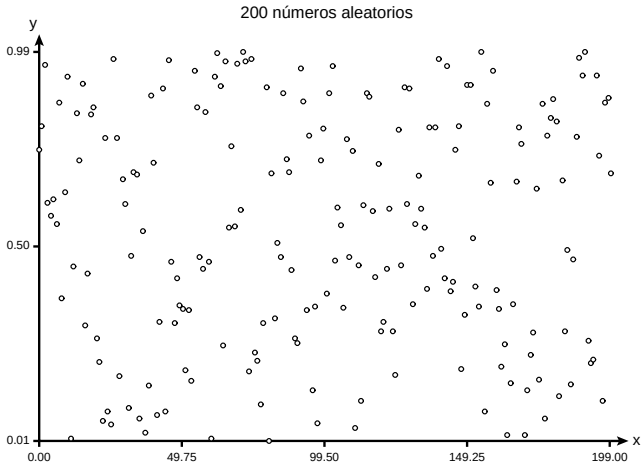


Figura 12.1: 200 números aleatorios entre 0 y 1 representados en el plano.

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)]
```

Como 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))]
```

✍ Estamos encontrando la transpuesta mencionada en el ejercicio 10.22.

Ahora hacemos el gráfico, que será similar al de la figura 12.1:

```
import grpc
grpc.puntos(pts2d(lista))
grpc.titulo = str(n) + ' números aleatorios'
grpc.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
grpc.reinicializar()   # borrar objetos gráficos
grpc.puntos(pts2d(clasif))
grpc.titulo = 'Los números clasificados'
grpc.mostrar()
```

✍ En vez de copiar la lista en `clasif` y luego ordenarla, podríamos haber puesto directamente `clasif = sorted(lista)`, como veremos en el ejercicio 13.1.

✍ 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

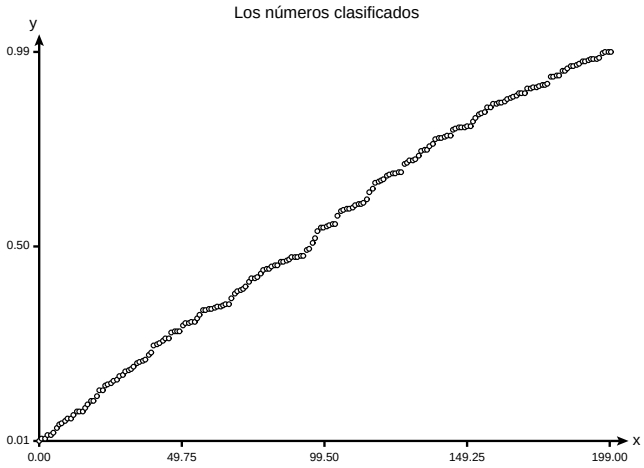


Figura 12.2: Los números clasificados.

bien usando `grpc.reinicializar` para no reinicializar IDLE (como hacemos acá), o bien usando `random.seed` con la misma semilla cuando generamos la lista.

Obtenemos un gráfico como el de la figura 12.2, 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 $[a,b] \subset [0,1]$ 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]
  grpc.reinicializar()
  grpc.puntos(pts2d(lista2))
  grpc.titulo = 'La mitad de los números'
  grpc.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
  grpc.reinicializar()
  ```
  ```

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

Ejercicio 12.3. Repetir los pasos anteriores tomando sólo los últimos $\lfloor n/2 \rfloor$ números, y luego los que estén en posición par, 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.

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

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

12.3. Aplicaciones

Ejercicio 12.4. 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) Modificarla función para simular tirar una moneda con resultados «cara» o «ceca». ¶

Ejercicio 12.5. 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 12.6. Hacer una función que diga cuántas veces debió tirarse un dado hasta que aparecieron k seis consecutivos, 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 un seis 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.

Suponiendo que el generador de números aleatorios es correcto, se puede demostrar matemáticamente que la probabilidad de que esto suceda es 0.

En ejemplos «chicos» como los que hacemos no hay problemas, pero puede suceder que haya problemas en la práctica en otros casos, 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, debido al comportamiento cíclico que mencionamos anteriormente (al principio de la sección 12.2).

Ejercicio 12.7. Recordando lo hecho en la sección 12.2:

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 12.8.

a) Hacer 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 parciales $.45, .45+.35, \ldots$, y no estaría de más recordar el ejercicio 10.17, en especial el apartado *d*).

b) Generalizar al caso en que en vez de la lista $(1,2,3,4)$ se dé la lista $(1,\ldots,n)$ y que en vez de las frecuencias $(.45,.35,.15,.5)$ 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 12.9 (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.

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 poner índices en las listas.*

Las funciones de hash *pueden verse en el libro de Wirth (1987), llamadas allí* transformaciones de llaves, *o con más detalle en el libro de Knuth (1998, vol. 3).* ¶

Ejercicio 12.10. 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 0 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 sabe 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 no importaba cambiar la elección, mientras que algunos pocos pensaban que era mejor cambiar.

a) ¿Qué te parece (intuitivamente)?

b) Para tener una idea de la solución al problema, hacer un función que simule:

i) el poner un auto detrás de una de las puertas (sin que nosotros sepamos el resultado),

ii) que nosotros hagamos una elección (usando `input`),

iii) que en base a esta elección, el «locutor» elija una segunda puerta detrás de la cual no está el auto,

iv) que nosotros mantengamos o cambiemos la elección, y

v) que diga si ganamos o no el auto.

c) Usar varias veces la función y luego proponer una estrategia con buenas probabilidades de ganar. Luego hacer una función para comprobar la estrategia en 1000 corridas, retornando la tasa de éxito. ¶

12.4. Métodos de Monte Carlo

Existen muchos métodos, llamados *de Monte Carlo* (en honor al famoso casino) para aproximar cantidades determinísticas mediante probabilidades.

Los dos ejercicios siguientes son ejemplos de esta técnica.

Ejercicio 12.11. Hacer 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 12.12. Hacer 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. ¶

Capítulo 13

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.

El método de clasificación que usa Python es bastante sofisticado y rápido. Para clasificar una lista de longitud n realiza del orden de $n \times \log n$ pasos (entre comparaciones y asignaciones). En contraposición, los métodos de clasificación elementales en general usan del orden de n^2 pasos, y no vale la pena, entonces, dedicar tiempo al estudio de estos métodos.

De cualquier forma, vamos a ver el *método de conteo* en el ejercicio 13.3 que extiende técnicas ya vistas, y es muy sencillo y rápido (del orden de n) en ciertas circunstancias.

Análogamente, con `in`, `index` y `count` (ejercicio 8.18) podemos buscar un objeto en una lista, dando eventualmente su posición, pero podemos mejorar la eficiencia si la lista está ordenada usando búsqueda binaria, lo que hacemos en la sección 13.3.

13.1. Clasificación

Ejercicio 13.1. Como sabemos, 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:

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

b)
```
a = 'mi mamá me mima'
a.sort()        # da error porque es inmutable
b = sorted(a)
b               # da una lista
a               # no se modificó
''.join(b)
```

✍ Es posible que la «á» (con tilde) no aparezca en el lugar correcto. Para que lo haga hay que cambiar algunas cosas, y no siempre funciona bien. Nuestra solución será sencillamente no usar tildes o diéresis cuando clasifiquemos palabras o letras.

c)
```
a = (4, 1, 2, 3, 5)
sorted(a)                # da una lista
sorted(a, reverse=True)  # orden inverso
```
¶

Ejercicio 13.2 (`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, de modo que se puedan comparar.

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 `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 la lista de nombres original,
```
def nombre(x):
    return x[0]
nombres = [nombre(x) for x in a]
```
agreguémosle algunos nombres más,
```
nombres = ['Ana', 'Luisa', 'Mateo', 'Adolfo',
   'Geri', 'Guille', 'Maggie'] + nombres
```
y llamemos `n` a la cantidad de nombres:
```
n = len(nombres)
```

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, i. e., `datos` es la transpuesta de `[nombres, edades, notas]` (ejercicio 10.22).

iv) Clasificar `datos` de manera que los datos aparezcan ordenados por nota inversamente, luego por edad, y finalmente por nombre (primero los que tienen más nota, a una misma nota primero los que tienen menor edad, a una misma edad, alfabéticamente). ¶

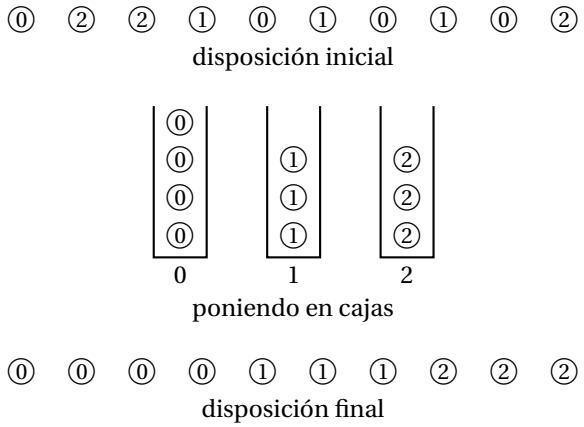


Figura 13.1: Ordenando por conteo.

Ejercicio 13.3 (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 $m-1$. Si

```
a = [0, 2, 2, 1, 0, 1, 0, 1, 0, 2]
```

las bolitas estarían alineadas como en la figura 13.1 (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,
# 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) Hacer 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 aleatorios entre 1 y 10.

b) Podemos pulir el esquema un poco.

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

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

iii) ¿Se podría mejorar el lazo `for k in range(m)...`?

iv) ¿Se podría eliminar completamente la lista `cuenta`?

✍ Se puede ver que el método usa del orden de $n+m$ pasos, asignaciones y comparaciones, lo que lo hace mejor que otros algoritmos generales cuando m es chico, digamos menor que n.

❦ *El método se generaliza al llamado* bucket sort *o* bin sort,[1] *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.* ¶

Ejercicio 13.4 (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 a_ℓ tal que la mitad de los datos son menores o iguales que a_ℓ y la otra mitad son mayores o iguales que a_ℓ. 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.

✍ En realidad no es necesario ordenar todos los datos para obtener la mediana.

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.

Hacer una función para generar aleatoriamente una lista de longitud n (argumento de la función) de números entre 0 y 9, y luego encontrar la media, mediana y moda/s. ¶

13.2. 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 en el curso.

Ejercicio 13.5 («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

```
b = []
for x in a:
    if x not in b:
        b.append(x)
b
```

Hacer una función con estas ideas y probarla en el ejemplo dado.

b) En base al esquema anterior, definir una función `purgar` de modo que `a` se modifique adecuadamente.

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.

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

✍ *Sugerencia si la anterior no alcanza*:

```
i = 0
for x in a:
    if x not in a[:i]:
        a[i] = x
        i = i + 1
del a[i:]
```

c) ¿Qué problemas tendría usar el esquema

```
i = 1
for x in a[1:]:  # no tocar el primer elemento
    if x not in a[:i]:
        a[i] = x
        i = i + 1
del a[i:]
```

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

Ejercicio 13.6 («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 13.5, pero una variante del apartado *c*) 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]
del 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 13.5 en general hace del orden de n^2 comparaciones, mientras que el de este ejercicio usa del orden de n. ¶

Ejercicio 13.7 (de lista a conjunto). Hacer 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 13.6) que purgar primero (según el ejercicio 13.5) y después clasificar. ¶

Ejercicio 13.8 («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[:]    # o c = a + b
c.extend(b)
```

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

b) Hacer 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
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.extend(a[i:n])        # copiar el resto de a
c.extend(b[j:m])        # 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.

c) ¿Podrían usarse listas por comprensión?, ¿cómo? ¶

Ejercicio 13.9 («intersección» de listas). Análogamente, queremos construir 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 13.8.*b*).

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

13.3. Búsqueda binaria

Ejercicio 13.10 (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 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 13.11. 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 13.10.

Ejercicio 13.12 (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

$$mucho = poco + 1, \tag{13.1}$$

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

$$medio = poco. \tag{13.2}$$

i) Verificar que (13.1) implica (13.2).

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 en una función para buscar un elemento en una lista (¡ordenada no decrecientemente!) retornando `False` o `True` y en este caso imprimiendo su posición.

d) Probar la función 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 (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 13.10)? 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 (ver el *manual de la biblioteca*). ¶

13.4. Ejercicios adicionales

Ejercicio 13.13 (generala). Hacer un programa para simular el juego de la generala en el que se tiran 5 dados (una única vez), y se tiene «full» cuando hay 3 dados con un mismo número y los otros 2 con otro número, «póker» cuando hay 4 dados iguales, «escalera» cuando hay 5 números consecutivos, y «generala» cuando los 5 dados son iguales.

Sugerencia: una vez «tirados» los dados, hacer alguna variante de «purga» para determinar si se ha obtenido alguno de los juegos. ¶

Ejercicio 13.14 (Pasos en búsqueda binaria). La función f que estudiamos en este problema está relacionada con la cantidad de pasos que se realizan en la búsqueda binaria cuando hay n elementos.

Consideremos $f:\mathbb{N}\to\mathbb{N}$ definida por $f(n)=\lfloor (n+1)/2\rfloor$, y para $k\in\mathbb{N}$,

$$F(k,n)=f^k(n)=\underbrace{f(\cdots(f}_{k\text{ veces}}(n))\cdots).$$

a) Hacer una función para calcular $F(k,n)$.

b) ¿Cuál es el valor de $F(k,2^k)$? (probar con algunos valores de k, hacer una conjetura y tratar de demostrarla).

c) Para cada $n>2$, existe $j=g(n)$ tal que $F(k,n)>1$ si $k<j$ y $F(k,n)=1$ para $k\geq j$. Por ejemplo, $g(3)=2$. Por simplicidad ponemos $g(1)=g(2)=1$. Hacer una función para calcular $g(n)$.

d) Si $m\leq n$ entonces $f(m)\leq f(n)$ y $g(m)\leq g(n)$.

e) ¿Cuál es el valor de $g(2^k)$?

f) Ver que si $2^k<n<2^{k+1}$, entonces $g(n)=k+1$. Por lo tanto, $g(n)=\lceil \log_2 n\rceil$ para todo $n\geq 2$. ¶

13.5. 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).

Capítulo 14

Números enteros y divisibilidad

A partir de ahora veremos pocas cosas nuevas de Python, concentrándonos en problemas matemáticos y cómo resolverlos con la computadora.

En este capítulo vemos problemas de matemáticas discretas, en general relacionados con teoría elemental de números, y empezamos con uno de los resultados más antiguos que lleva el nombre de algoritmo.

14.1. El algoritmo de Euclides

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

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

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

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

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

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

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

✍ Para nosotros, un número p es primo si $p \in \mathbb{N}$, $p > 1$, y los únicos divisores de p son ± 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 la escuela elemental nos enseñan a calcular el máximo común divisor efectuando primeramente la descomposición en primos. Sin embargo, la factorización en primos es computacionalmente difícil,[2] 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.

❦ *Euclides de Alejandría (alrededor de 325–265 a. C., aunque hay discusión sobre si se trata de una persona o un grupo) escribió una serie de libros de enorme influencia en las matemáticas, inclusive en las actuales. En* Los elementos *presenta los principios de*

[1] ¡Como su nombre lo indica!

[2] Ver también las notas después del problema 14.10.

lo que se llama «geometría euclidiana» a partir de un pequeño conjunto de axiomas. Estos axiomas perduraron hasta fines del siglo XIX (con las modificaciones de Pasch y la aparición de las geometrías no euclidianas), y en nuestro país los libros de texto de geometría en la escuela secundaria siguieron su presentación hasta bien avanzado el siglo XX (hasta la aparición de la «matemática moderna»).

En la época de Euclides los números representaban longitudes de segmentos, y de allí que Los elementos *tratara dentro de la geometría cuestiones que hoy consideraríamos como de teoría de números.*

En el original griego, Euclides *significa «renombrado» o «buena fama» (eu:* bien, *kleos:* fama*).*

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

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

✍ 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 14.1 (algoritmo de Euclides).

a) La función `mcd1` en el módulo *enteros* corresponde al algoritmo original (sólo para enteros positivos).

i) Probar la función para entradas positivas, e. g., `mcd1(612, 456)` da como resultado `12`.

ii) Ver que si algún argumento es nulo o negativo el algoritmo no termina (¡sin ejecutar la función!).

iii) 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.

b) La función `mcd2` (en el módulo *enteros*) es una versión para cuando $a, b \in \mathbb{Z}$, cambiando las restas sucesivas por el cálculo del resto. Verificar el funcionamiento tomando distintas entradas (positivas, negativas o nulas).

✍ En el ejercicio 9.6 hicimos el proceso inverso: para calcular el resto hicimos restas sucesivas.

c) 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.

❦ *En realidad el número de pasos nunca supera* 5 *veces la cantidad de cifras del número menor, lo que se demuestra... ¡usando los números de Fibonacci!*

d) Modificar la función de modo que a la salida escriba la cantidad de veces que realizó el lazo `while`.

e) ¿Qué pasa si se cambia la instrucción `while b != 0` por `while b > 0`?

f) Modificar la función de modo que a la salida escriba también los valores originales de a y b, por ejemplo que imprima `El máximo común divisor entre 12 y 8 es 4` si las entradas son $a = 12$ y $b = 8$.

g) Una de las primeras aplicaciones de mcd es «simplificar» números racionales, por ejemplo, escribir 12/8 como 3/2. Hacer un programa 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 $\operatorname{mcd}(m, n) = 1$.

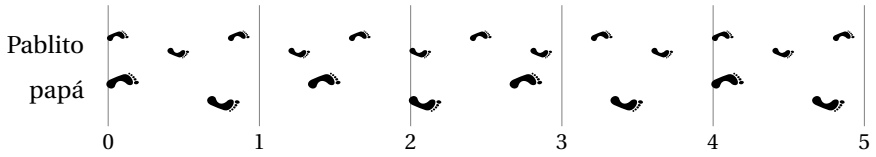


Figura 14.1: Pasos de Pablito y su papá.

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

Ejercicio 14.2. El *mínimo común múltiplo* de $a, b \in \mathbb{N}$, $\text{mcm}(a,b)$, se define en forma análoga al máximo común divisor: es el menor entero del conjunto $\{k \in \mathbb{N} : a \mid k \text{ y } b \mid k\}$.

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

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

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

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

Ejercicio 14.3 (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 14.1).

Respuesta: 4 y 20 metros respectivamente.

b) ¿Qué relación hay entre el máximo común divisor o el mínimo común múltiplo con 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 14.16.

¶

14.2. Ecuaciones diofánticas

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 14.3)— 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*.[(3)]

[(3)] Bah, ¡a lo bestia!

Ejercicio 14.4. Geri y Guille compraron botellas de vino para la 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) Hacer 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 $y\in\mathbb{Z}$, $y\geq 0$.

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. ¶

La estrategia de resolución del ejercicio 14.4 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 14.5. 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).

Hacer un programa 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
```
¶

También la técnica de «barrido» puede usarse para ecuaciones diofánticas no lineales.

Ejercicio 14.6. Hacer un programa para 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.

❦ *Se puede demostrar que todo entero positivo es suma de cuatro cuadrados (algunos eventualmente nulos), y que un entero positivo es suma de dos cuadrados si y sólo si es de la forma $a\times b$, donde a es una potencia de 4 (eventualmente $4^0=1$) y b tiene resto 1 al dividirlo por 4.* ¶

14.3. 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.

❦ *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 14.7 (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

$$\boxed{2} \quad 3 \quad \not{4} \quad 5 \quad \not{6} \quad \ldots$$

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

$$\boxed{2} \quad \boxed{3} \quad \not{4} \quad 5 \quad \not{6} \quad 7 \quad \not{8} \quad \not{9} \quad \not{10} \quad 11 \quad \not{12} \quad \ldots$$

y seguimos de esta forma, recuadrando y tachando múltiplos hasta agotar la lista.

a) La función `criba` en el módulo *enteros* 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.

✍ Si $a \times b = m$ y $\sqrt{m} < b < m$, debe ser $1 < a < \sqrt{m}$.

Por lo tanto, en el lazo principal podemos reemplazar `range(2, n+1)` por `range(2, s + 1)` donde $s = \lfloor \sqrt{n} \rfloor$, y también cambiar el lazo en `j` por:

```
for j in range(i*i, n+1, i):
```

Hacer estos cambios, viendo (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 14.12). Es decir, el trabajo que realiza la criba es casi lineal con respecto a la cantidad de elementos que encuentra. ❡

Ejercicio 14.8 (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. El problema es ver en qué posición debió colocarse Flavio Josefo dentro del círculo para quedar como último sobreviviente.

En vez de ser tan crueles y matar personas, vamos a suponer que tenemos un 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 4, como ilustramos en la figura 14.2 (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 resolver 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.

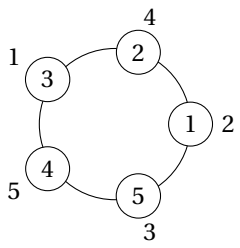


Figura 14.2: Esquema del problema de Flavio Josefo con $n=5$ y $m=3$.

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):  # n-1 vueltas
    cuenta = 0          # contamos m jugadores
    while cuenta < m:
        if s < n:       # incrementar 1
            s = s + 1                                   (14.1)
        else:           # empezar con 1
            s = 1
        if salio[s] < 0:    # si no salió
            cuenta = cuenta + 1
    salio[s] = vuelta   # la vuelta en que salió
s = salio.index(-1)     # el último que queda
salio[s] = n
```

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` ⟺ `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 14.9 (el problema de Flavio Josefo II). La criba según el esquema (14.1) 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.

Para atender a las observaciones anteriores, 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 vamos sacando jugadores de `quedan`, podemos ir formando las listas `salio` y `flavio` de antes.

Nos quedaría un esquema como el siguiente:

```
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
```
(14.2)

Hacer 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 esquema (14.2) es unas 5 veces más rápido que el (14.1) para $n = 1234$ y $m = 3$. ¶

14.4. Números primos

Los números primos son considerados como ladrillos para la construcción de los enteros, pero en cierto 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.

En años recientes, 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 vemos algunas preguntas, varias simples de enunciar, que no han sido respondidas completamente por los matemáticos, y miramos algoritmos elementales (¡e ineficientes!) para su resolución.

- 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 14.10 vemos métodos para resolver estos problemas, que son *extremadamente ineficientes* para números grandes (digamos de 100 cifras, los usados en criptografía tienen 200 o más).

- La función ϕ de Euler que calculamos en el ejercicio 14.11 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 es muy elemental, lejos de lo que se hace en la realidad. El ejercicio 15.30 da una idea de cosas que se pueden hacer para mejorar la eficiencia.

- Si bien se sabe que hay infinitos primos desde Euclides, no se sabe muy bien cómo se distribuyen.

  Recién hacia fines del siglo XIX pudieron resolverse algunas cuestiones muy generales que tratamos en el ejercicio 14.12 sobre la distribución de los primos.

  ❦ *El teorema de los números primos que enunciamos en el ejercicio 14.12.a) 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 $\mathrm{mcd}(a,b)=1$. *En el ejercicio 14.12.b), si pensamos que $b=10$, y a es cualquier número que no tenga como factores a 2 ni a 5, el teorema de Dirichlet dice que hay infinitos primos que terminan en 1, infinitos que terminan en 3, etc., pero no dice cómo es la distribución. Esto fue demostrado en forma general por Franz Carl Joseph Mertens (1840–1927) al refinar los resultados de Dirichlet, usando la función ϕ de Euler.*

  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.

- En el ejercicio 14.13 vemos propiedades relacionadas con la demostración de Euclides que hay infinitos primos, considerando los llamados *primos de Euclides*. No se sabe si hay infinitos primos de esta forma.
- Una pregunta sencilla sobre la distribución de los primos que los matemáticos no han podido resolver aún es la existencia de *primos gemelos* que consideramos en el ejercicio 14.14.
- Otra pregunta sencilla relacionada, y que tampoco se ha podido contestar es la conjetura de Goldbach que vemos en el ejercicio 14.15.

Ejercicio 14.10 (factorización de enteros). Como ya observamos en el ejercicio 14.7.*b*), el entero $a>1$ *no* es primo si y sólo si existen enteros b y c tales que $a=b\times c$ y $1<b\le\sqrt{a}$.

a) Usando este resultado (tomando ideas del ejercicio mencionado), hacer una función `esprimo` que determine si el entero $a>1$ es primo o no, viendo si los números $2,3,4,\ldots,\lfloor\sqrt{a}\rfloor$ lo dividen o no.

b) La función `factoresprimos` (en el módulo *enteros*) es una elaboración de la función anterior, retornando una lista de todos los factores primos del número $a>1$, o la lista vacía si $a=1$ (si la lista tiene longitud 1 el número es primo).

Probarla con algunas las entradas como `1`, `2`, `4`, `123456789`, `1234567891`, `12345678912345678 91`.

c) Modificar la función de modo que la lista no tenga repetido el mismo factor primo, sino que devuelva una lista de listas, donde cada sublista tenga el factor primo seguido de su multiplicidad. Por ejemplo, si $a=456=2^3\times3\times19$, la lista retornada debe ser `[[2, 3], [3, 1], [19, 1]]`. ¶

Ejercicio 14.11 (ϕ de Euler). Para $n\in\mathbb{N}$ se define la *función ϕ de Euler* como la cantidad de coprimos menores que n (y positivos), es decir,

$$\phi(n)=|\{m\in\mathbb{N}:1\le m\le n \text{ y } \mathrm{mcd}(m,n)=1\}|.$$

Una forma de entender ϕ es pensar que es la cantidad de fracciones entre 0 y 1 que simplificadas tienen denominador n. Por ejemplo, $\phi(15)=8$ y las fracciones son 1/15, 2/15, 4/15, 7/15, 8/15, 11/15, 13/15 y 14/15.

a) Probar que $n \in \mathbb{N}$ es primo $\Leftrightarrow \phi(n) = n-1$.

b) Desarrollar 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, basados en propiedades que no vemos. 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, $\text{mcd}(n,k) = 1 \Leftrightarrow \text{mcd}(n-k,n) = 1$.

d) Un primo p tal que $\phi(p)$ es una potencia de 2 se llama *primo de Fermat*, y debe ser de la forma $p = 2^{2^k} + 1$.
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, n debe ser el producto de una potencia de 2 y primos de Fermat. Pierre Wantzel demostró en 1837 que la condición también es necesaria.
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. ¶

Ejercicio 14.12 (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 todos 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 mencionados al principio de esta sección. ¶

Ejercicio 14.13 (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

$$x = p_1 \times p_2 \times \cdots \times p_n + 1, \tag{14.3}$$

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 (14.3). Por ejemplo, los primeros primos de Euclides son $3 = 2+1$, $7 = 2\times3+1$ y $31 = 2\times3\times5+1$.

Sin embargo, ni todos los primos son de esta forma (como 5), ni todos los números de esta forma son primos. Por ejemplo

$$9\,699\,691 = 2\times3\times5\times7\times11\times13\times17\times19+1 = 347\times27953.$$

Encontrar todos los números de la forma (14.3) menores que 9 699 691 que *no* son primos de Euclides. ¶

Ejercicio 14.14 (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[(4)] está formado por

$$3\,756\,801\,695\,685^{2\,666\,669} \pm 1,$$

[(4)] Marzo de 2012, `http://primes.utm.edu/top20/page.php?id=1#records`

(cada uno con 200700 cifras decimales), pero no se sabe si hay infinitos pares de primos gemelos.

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 14.15 (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) Hacer 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) 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 todos los pares entre 4 y 10 000, y encontrar el máximo y mínimo. ¶

14.5. Ejercicios adicionales

Ejercicio 14.16 (Pablito y su papá II). Hacer una función para resolver en general el problema de Pablito y su papá (ejercicio 14.3), donde las entradas son el número de pasos y la cantidad de metros recorridos tanto para Pablito como para su papá. El número de pasos debe ser entero positivo, mientras que los metros recorridos (en cada caso) son números fraccionarios representados por un par de números (numerador y denominador) ambos positivos.

✍ En el ejercicio 15.7 podemos apreciar por qué trabajamos con aritmética exacta en este problema. Ver también la nota al principio de la sección 15.1.

Ayuda: Supongamos que las entradas son:

$$\begin{aligned} p_b &: \text{ número de pasos de Pablito} \\ m_b = n_b/d_b &: \text{ metros recorridos por Pablito en } p_b \text{ pasos} \\ p_p &: \text{ número de pasos del papá} \\ m_p = n_p/d_p &: \text{ metros recorridos por el papá en } p_p \text{ pasos} \end{aligned}$$

de modo que las entradas a la función son p_b, n_b, d_b, p_p, n_p y d_p.

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

$$2 \times k_b \times \frac{m_b}{p_b} = 2 \times k_p \times \frac{m_p}{p_p}. \tag{14.4}$$

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

$$k_b \times n_b \times d_p \times p_p = k_p \times n_p \times d_b \times p_b \tag{14.5}$$

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 (14.5) 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 (14.4). ¶

Ejercicio 14.17 (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.

Vamos a construir una función que escribe la parte decimal (del desarrollo en base 10) de p/q, distinguiendo su período y anteperíodo. Para eso tenemos que ir mirando los restos sucesivos de la división. En cuanto un resto se repite, se repetirá el cociente, así como todos los restos y cocientes sucesivos.

Bastará asociar cada resto, r, con el cociente que produce, c, al hacer la división:

$$r \times 10 = c \times q + r',$$

donde r' es el nuevo resto.

En la función `periodo` (en el módulo *periodo*) hacemos este trabajo, conservando los restos (para ver si se repiten) y los cocientes (que nos dan los dígitos decimales).

a) Estudiar las instrucciones de la función, y probarla para distintos ejemplos, comparando con el valor de `p / q` dado por Python, por ejemplo para 1/23.

b) Agregar condiciones de modo de no ejecutar el lazo principal si `q` es cero.

c) Agregar también condiciones al principio para reducir el caso en que `p` o `q` son negativos al caso donde ambos son positivos.

d) 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.

e) 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 14.18. 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. ¶

Ejercicio 14.19 (Visibilidad en el plano). Consideremos el *reticulado* $\mathscr{C}$ definido por los puntos del plano con ambas coordenadas enteras. Diremos que el punto $P \in \mathscr{C}$ es *visible* (desde el origen O) si el segmento que une P y O no contiene otros puntos de $\mathscr{C}$ distintos de P y O.

a) Ver que si $a, b \in \mathbb{N}$, entonces (a, b) es visible $\Leftrightarrow \operatorname{mcd}(a, b) = 1$.

b) Desarrollar una función que dado $n \in \mathbb{N}$ calcule s_n, la cantidad de puntos visibles en el cuadrado $1 \leq a, b \leq n$.

c) Se puede demostrar que a medida que n crece, $p_n = s_n/n^2$ se aproxima a $p = 6/\pi^2 = 0.6079\ldots$ Calcular p_n para distintos valores de n para ver que este es el caso.

Sugerencia: empezar con n pequeño e ir aumentando paulatinamente su valor. ¶

14.6. Comentarios

- Los ejercicios 14.8, 14.18 y 14.19 están tomados de Engel (1993).

Capítulo 15

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. 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 == 0` con `a != b`,
- `(a + b) + c != a + (b + c)`, o sea, la suma no es asociativa.

En este capítulo empezamos mirando a estos inconvenientes, para pasar luego ver a técnicas efectivas de resolución de problemas.

15.1. La codificación de decimales

Como vimos en el ejercicio 4.6 al calcular 876^{123}, Python puede trabajar con cualquier número entero (sujeto a la memoria de la computadora), pero no con todos los números decimales, para los que 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 14.16 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

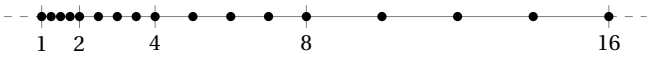


Figura 15.1: Esquema de la densidad variable en la codificación de números decimales.

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 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 15.1. 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 puede 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 != b + c + a
```

¶

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 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 15.1.

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 15.2. 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
```

Hay que tener mucho cuidado: 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 15.3 ($\varepsilon_{\text{máq}}$). 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 siguiente número a 1,
- $\varepsilon_{\text{máq}}$ 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`. 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 $2 = 2\times 1$, $3 = 2\times(1+1/2)$,..., $4-2^{-j} = 2\times(2-2^{-j-1})$,...).

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 15.2. ¿Es cierto que `epsmaq` es aproximadamente `1 / a`? (bastará multiplicar `a` y `epsmaq`). ¶

Ejercicio 15.4 ($\varepsilon_{\text{mín}}$). Otro indicador importante es $\varepsilon_{\text{mín}}$, el *épsilon mínimo*, que es el menor número decimal positivo que se puede representar.

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

a) Decir por qué falla el esquema de ejercicios anteriores:

```
c = 1.0
while c > 0:
    c = c / 2
c = 2 * c   # nos pasamos: volver para atrás
```

b) Construimos `epsmin` en el módulo *decimales* eliminando el problema al conservar el último valor no nulo: comprobar que su comportamiento es correcto. ¶

Ejercicio 15.5 (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 4.6, 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 5.7, 8.23 o 9.7) 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 15.2, 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) Usando el valor de `n` obtenido en el apartado anterior, 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
```

e) `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 simplemente float('inf')
b = 2*x      # -> inf
a == b       # -> verdadero
```

☞ *Python a veces da el valor* `inf` *cuando hace cálculos con decimales, pero sólo números no demasiado grandes se deben comparar con* `inf`. ¶

Ejercicio 15.6. 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 == 0` con `x` distinto de `y`.
Sugerencia: usar el anterior.

c) `(x + y) + z != x + (y + z)`.
Sugerencia: recordar el ejercicio 15.1, o tal vez el 15.2.*c*), con `y = 1`, `z = 1`.

d) `x + y + z != y + z + x`.
Sugerencia: usar el anterior. ¶

15.2. 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 15.1 vimos que podemos tener `a + b + c != 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 15.7 (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 homónimas del módulo *enteros*, `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 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 $\varepsilon_{máq}$, $\varepsilon_{mín}$ y otros números de la sección 15.1 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 entre *error absoluto* (relacionado con $\varepsilon_{mín}$) y *relativo* (relacionado con $\varepsilon_{máq}$), que no estudiaremos en el curso (y se estudian en cursos de estadística, física o análisis numérico). Nos contentaremos con poner una tolerancia «razonable», generalmente del orden de $\sqrt{\varepsilon_{máq}}$.

Ejercicio 15.8 (problemas numéricos en la ecuación cuadrática). Como sabemos, la ecuación cuadrática

$$ax^2+bx+c=0 \tag{15.1}$$

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

$$x_1=\frac{-b+\sqrt{d}}{2a} \qquad \text{y} \qquad x_2=\frac{-b-\sqrt{d}}{2a}. \tag{15.2}$$

a) Hacer 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 (15.2), y también calcule las cantidades $ax_i^2+bx_i+c$, $i=1,2$, 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 (15.1) en cada caso. ¶

15.3. 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 (salvo las 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 15.9 (punto fijo de la raíz cuadrada).

a) Utilizando la construcción

```
y = x
for i in range(n):
    y = math.sqrt(y)
```

hacer 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 $\sqrt{1}=1$.

Un punto x en el dominio de la función f se dice *punto fijo de* f si

$$f(x)=x,$$

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 15.2 (izquierda) cuando $f(x)=\cos x$.

Ejercicio 15.10. Haciendo un gráfico combinado de $f(x)=\sqrt{x}$ y de $y=x$ con *grpc*, encontrar otro punto fijo de f (distinto de 1). ¶

Ejercicio 15.11. 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 $x\geq 0$).

✍ Suponemos conocidas las definiciones de función continua y de función derivable, que generalmente se da en los cursos de análisis o cálculo matemático.

✍ Por supuesto, hay funciones continuas que «no se pueden dibujar» como

- $1/x$ 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

- signo(x) para $x\in\mathbb{R}$, pues «pega un salto» en $x=0$,
- la función de Dirichlet
  $$f(x)=\begin{cases} 1 & \text{si } x\in\mathbb{Q},\\ 0 & \text{si } x\in\mathbb{R}\setminus\mathbb{Q}, \end{cases}$$
  una función 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 15.2, 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 15.2 vemos cómo a partir de $x_0=0$, nos vamos aproximando al punto fijo, donde los trazos horizontales van desde puntos en el gráfico de f a la diagonal $y=x$ y los verticales vuelven al gráfico de f.

Ejercicio 15.12 (punto fijo de $f(x)=\cos x$). Con las notaciones anteriores para $f(x)=\cos x$ y x_i:

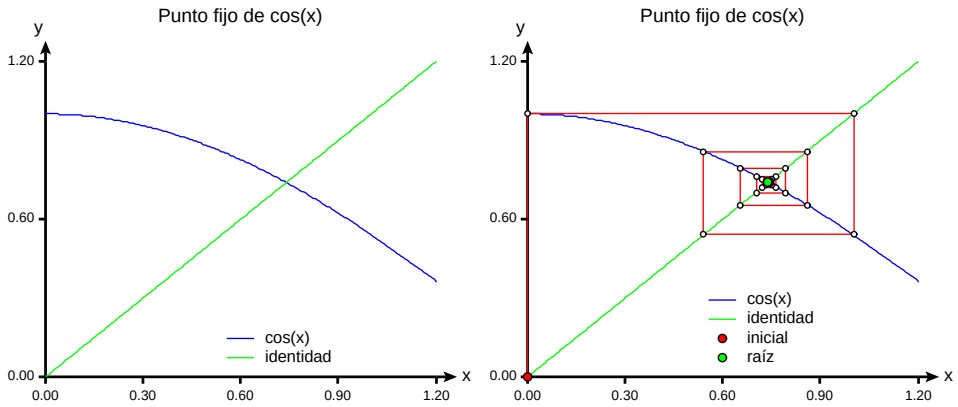


Figura 15.2: Gráfico de $\cos x$ y x (izquierda) y convergencia a punto fijo desde $x_0 = 0$ (derecha).

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 que no se hagan más iteraciones si

$$|x_{k+1} - x_k| < \varepsilon,$$

donde $\varepsilon > 0$ es un nuevo argumento (por ejemplo, $\varepsilon = 0.000\,01 = 10^{-5}$), 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 $|f(x_k) - x_k| < \varepsilon$. ¶

En el módulo *numerico* se define la función `puntofijo` que retorna un cero de la función con una tolerancia permitida, realizando un número máximo de iteraciones, sintetizando lo hecho en el ejercicio 15.12.

Ejercicio 15.13. La figura 15.2 fue hecha con el módulo *grpuntofijo*: ejecutarlo, comprobando que se obtienen resultados similares a la figura mencionada. ¶

Ejercicio 15.14. *grpuntofijo* es una «plantilla» (*template* en inglés) para ilustrar el método de punto fijo, y lo usaremos como «caja negra» cambiando la función y otros parámetros.

a) Usar *grpuntofijo* para ilustrar el método de punto fijo para la función $\sqrt{x}$ que consideramos anteriormente, tomando como intervalo $[0,3]$ y como puntos iniciales 0, 0.5, 1 y 2.

b) Repetir para la función x^2 (con el mismo intervalo y puntos iniciales). ¶

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, $|f(x_{n+1}) - f(x_n)| < \varepsilon_y$,
- 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.

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 15.15. 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.

☞ *Aún cuando un método iterativo converja a una solución, no siempre obtenemos el punto fijo que buscamos.*

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 15.16. 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.*

15.4. 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 desarrollado por Newton para funciones mucho más generales.

❦ *Los babilonios tomaron poder de la Mesopotamia (entre los ríos Tigris y Éufrates) alrededor de 2000 a. C., desalojando a los sumerios, quienes habían estado allí desde 3500 a. C. Fueron los sumerios los que desarrollaron el sistema sexagesimal (en base* 60*) que nos llega a la actualidad en la división de horas en minutos y segundos, y los*

babilonios continuaron con el desarrollo del sistema en base 60*, llegando a un sistema que en algunos sentidos era más avanzado que el decimal nuestro.*

Pero los babilonios también estudiaron la resolución de ecuaciones cuadráticas y cúbicas, llegando al equivalente de las soluciones (15.2) a la ecuación cuadrática (15.1) y al método que presentamos en esta sección para aproximar raíces cuadradas, que estaría descripto en la tableta «Yale», fechada entre 1800 y 1650 a. C.

Mucho más tarde, Isaac Newton (1642–1727) desarrolló un método basado en el análisis matemático para encontrar raíces de funciones muy generales, que tiene como caso particular al método babilónico. El método de Newton también es conocido como de Newton-Raphson, pues J. Raphson (1648–1715) publicó este resultado unos 50 años antes que se publicara el de Newton.

Variantes del método de Newton-Raphson, que se estudia en los cursos de cálculo numérico, son los que usan las computadoras y calculadoras para calcular funciones como el seno o el logaritmo.

La derivada de f en x, $f'(x)$ se define como el valor al que se parecen los cocientes $(f(x+h)-f(x))/h$ cuando h es muy pequeño, o en símbolos,

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \quad \text{si } h \text{ es suficientemente chico.} \tag{15.3}$$

Si la derivada existe, o sea, si los cocientes se parecen a algo, 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 es lo que se denomina como $f'(x)$.

«Despejando» en la relación (15.3), llegamos a

$$f(x+h) \approx f(x) + f'(x)\,h,$$

que usamos en lo que sigue.

La idea del método de Newton es que si x^* es una raíz de la función derivable f, y x es un punto próximo a x^*, tendremos

$$f(x^*) = 0 \approx f(x) + f'(x)(x^* - x),$$

y despejando x^*, suponiendo $f'(x) \neq 0$, queda

$$x^* \approx x - \frac{f(x)}{f'(x)}.$$

Esto establece un método iterativo considerando la sucesión

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \text{para } n = 0, 1, \ldots,$$

siempre que f' no se anule en los puntos x_n. Es decir, buscamos un punto fijo de la función

$$g(x) = x - \frac{f(x)}{f'(x)}. \tag{15.4}$$

Ejercicio 15.17. Encontrar la función g dada en la ecuación (15.4) cuando $f(x) = x^2 - a$, donde $a \in \mathbb{R}$ es una constante dada.

Ayuda: $f'(x) = 2x$.

Respuesta: $g(x) = (x + a/x)/2$. ¶

El método babilónico para aproximar la raíz cuadrada de $a \in \mathbb{R}$, $a > 0$, consiste en calcular sucesivamente las iteraciones

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right) \qquad \text{para } n = 1, 2, \ldots, \tag{15.5}$$

a partir de un valor inicial x_0 dado ($x_0 > 0$) (finalizando con algún criterio). En otras palabras, $x_n = g(x_{n-1})$, donde g es la función encontrada en el ejercicio 15.17.

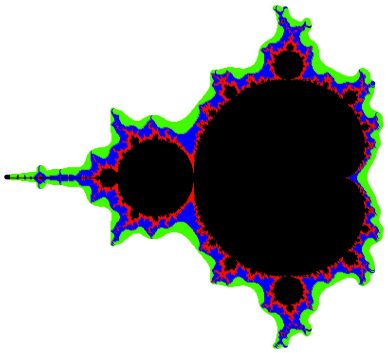


Figura 15.3: El conjunto de Mandelbrot.

Ejercicio 15.18 (método babilónico). La función `babilonico` (en el módulo *numerico*) es una versión para aplicar el método babilónico, finalizando si se alcanzó un máximo de iteraciones o un error aceptable (los criterios de parada).

a) Ejecutar la función para distintos valores de `a`, y luego modificarla para comparar el resultado con el valor de `math.sqrt(a)`.

b) Cambiar el lazo `while` por otro `for` usando `break` (de modo que el comportamiento sea el mismo).

c) En la función el valor inicial, el número máximo de iteraciones y la tolerancia están dados. Modificar la función para que estos datos sean argumentos.

d) Uno de los criterios de parada usados en la función es que la diferencia de dos valores consecutivos «en x» sea suficientemente pequeña. Agregar también un criterio para parar si la diferencia «en y», $|x^2 - a|$, es suficientemente pequeña, con tolerancia eventualmente distinta a la anterior.

e) Modificar la función para que imprima el criterio que se ha usado para terminar. Ver cuál es este criterio cuando $x_0 = 1$, el número máximo de iteraciones es 15, y ambas tolerancias son 10^{-5}, para $a = 10^{-2}, 10^4, 10^8$.

f) Comparar los resultados obtenidos al aproximar $\sqrt{2}$ y $\sqrt{200}$ con un número fijo de iteraciones (por ejemplo 5).

✍ 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 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 $\varepsilon_{\text{mín}}$.

g) A partir las iteraciones en (15.5), eventualmente imprimiendo unas pocas:

i) ¿Qué pasa si $a = 0$? ¿Y si $x_0 = 0$?

ii) ¿Qué pasa si $x_0 < 0$ y $a > 0$?

iii) ¿Qué pasa si $a < 0$ y $x_0 \in \mathbb{R}$?

✍ En el caso $a < 0$ se observa un comportamiento *caótico*, oscilando sin ninguna ley aparente. Si convergiera a algún número $\ell \in \mathbb{R}$, éste sería la raíz cuadrada de un número negativo, lo cual es absurdo.

❦ *El comportamiento caótico está relacionado con el conjunto de Mandelbrot (figura 15.3) y otros conjuntos «fractales».* ¶

No siempre conocemos la función explícitamente, por ejemplo si viene dada por datos experimentales. También puede suceder que la derivada tenga una expresión complicada. En ambos casos podemos usar la aproximación a la derivada dada en (15.3)

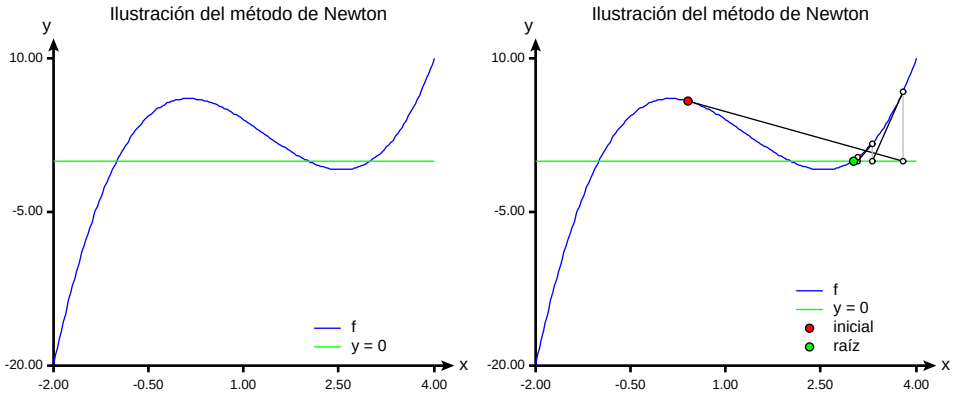


Figura 15.4: Gráfico de f en la ecuación (15.6) (izquierda) y convergencia del método de Newton desde $x_0 = 0.4$ (derecha).

si h es suficientemente chico (según f y x), y reemplazar a $f'(x)$ por el cociente a la derecha de esa ecuación.

Esto es lo que hacemos en la función `newton` (en el módulo *numerico*), que toma dos argumentos, la función f y el punto inicial x_0, y calcula internamente una aproximación de la derivada de f tomando Δx bien pequeño.

Consideremos por ejemplo

$$f(x) = (x+1)(x-2)(x-3), \tag{15.6}$$

que tiene ceros en -1, 2 y 3 y se ilustra a la izquierda en la figura 15.4. f tiene derivadas que se pueden calcular sencillamente, pero aplicamos el método de Newton usando derivadas aproximadas. El resultado puede observarse a la derecha en la figura 15.4, donde vemos los sucesivos puntos obtenidos cuando el punto inicial es 0.4.

✍ 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.

Ejercicio 15.19. La figura 15.4 fue hecha con el módulo *grnewton*, que es una «plantilla» para ilustrar el método de Newton. Ejecutar el módulo, viendo que se obtienen resultados similares a la figura mencionada. ¶

Ejercicio 15.20 (Newton con derivadas aproximadas). Usando la función `newton` y el módulo *grnewton* encontrar ceros de la función $\cos x$ tomando puntos iniciales 1.0 y 0.0. ¶

15.5. 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 15.5. 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 15.5.

✍ En los cursos de análisis o cálculo se demuestra que si f es continua en $[a,b]$, y tiene signos distintos en a y b, entonces f se anula en algún punto del intervalo.

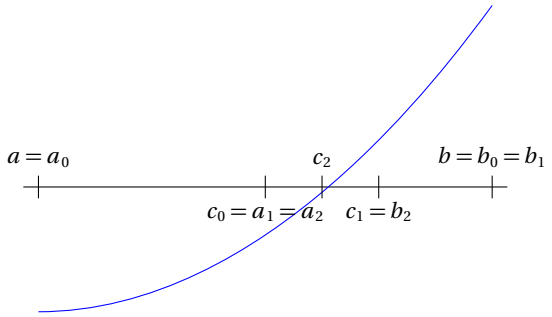


Figura 15.5: Función continua con distintos signos en los extremos.

Se finaliza según algún criterio de parada (como mencionados en la pág. 108), por ejemplo cuando se obtiene un valor de x tal que $|f(x)|$ es suficientemente chico, $|f(x)| < \varepsilon_y$, 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 $2^{10} = 1\,024 \approx 10^3$ y $2^{20} = 1\,048\,576 \approx 10^6$, por lo que el intervalo inicial se divide aproximadamente en 1000 después de 10 iteraciones y en $1\,000\,000 = 10^6$ 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 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 calculan los valores de la función en ambos extremos, `ypoco` y `ymucho`.
3. Si la condición de distinto signo en los extremos no se satisface inicialmente, el método no es aplicable y salimos poniendo un cartel apropiado.
4. Si ya el valor de `ypoco` es suficientemente chico, guardamos los valores `xpoco` y `ypoco` en `sol`, y de modo similar para el otro extremo.
5. El lazo principal se realiza mientras no se haya encontrado solución:
   - Se calcula el punto medio del intervalo, `x`, y el valor `y` de la función en `x`.
   - Si el valor absoluto de `y` es suficientemente chico, guardamos los valores de `x` y `y` en `sol`.
   - Si no, calculamos el nuevo intervalo, cuidando de que los signos en los extremos sean distintos.
   - Sólo necesitamos el signo de `ypoco` para determinar el nuevo intervalo, de modo que no conservamos el valor de `ymucho` (¿por qué?).
   - El criterio de parada es el obtener un valor de la función suficientemente chico.

Ejercicio 15.21 (método de la bisección para encontrar raíces). 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 15.6 (izquierda). Usando el método de la bisección para esta función tomando como intervalo inicial $[-1.2, 1.5]$, obtenemos una sucesión de intervalos dados por los puntos que se muestran en la misma figura a la derecha.

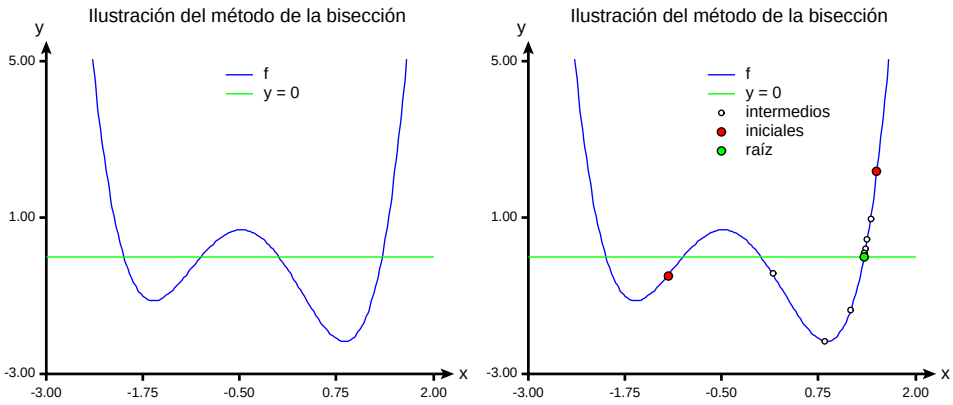


Figura 15.6: Método de bisección (ejercicio 15.21) función (izquierda) y puntos obtenidos para el intervalo inicial $[-1.2, 1.5]$ (derecha).

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) Agregar el criterio de parada por máximo número de iteraciones, digamos `maxit`, e imprimir un cartel diciendo por cuál criterio se ha terminado. Variar la tolerancia y el número de iteraciones para obtener que se sale por un criterio en varios casos y por el otro criterio en otros tantos casos.

c) ¿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.

d) La función `biseccion` no verifica si `poco < mucho`, y podría suceder que `poco > mucho`. ¿Tiene esto importancia?

e) Teniendo en cuenta las notas al principio de la sección (pág. 113), ¿tendría sentido agregar un criterio de modo de parar si los extremos del intervalo están suficientemente cerca? Si la nueva tolerancia fuera ε_x, ¿cuántas iteraciones deben realizarse para alcanzarla, en términos de ε_x y los valores originales de `mucho` y `poco`? ¶

Ejercicio 15.22. La figura 15.6 se hizo con el módulo *grbiseccion*, que —como los módulos *grpuntofijo* y *grnewton*— es una «plantilla» para ilustrar el método de la bisección. Ejecutar el módulo, viendo que se obtienen resultados similares a la figura mencionada. ¶

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, como vimos en el ejercicio 15.20 al tomar 0 como punto inicial.

Por supuesto que el método de la bisección y el de búsqueda binaria (sección 13.3) son esencialmente la misma cosa. Si tenemos una lista de números ordenada no decrecientemente $a_0 \le a_1 \le \cdots \le a_n$, 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 sólo valores enteros de x, y por eso hacemos

$$medio = \left\lfloor \frac{poco + mucho}{2} \right\rfloor,$$

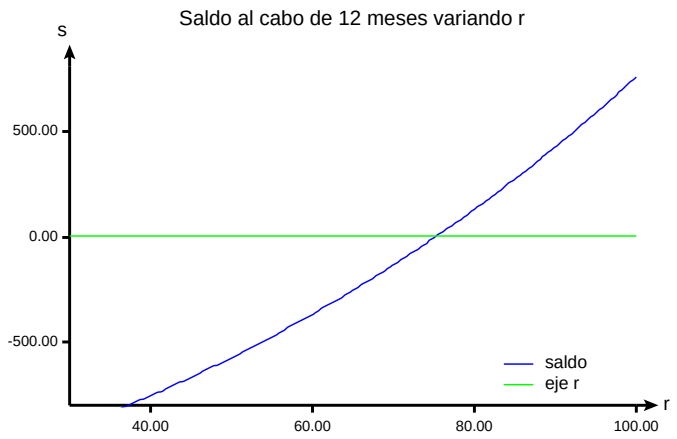


Figura 15.7: Gráfico de `saldo(c, r, p, m)` cuando r varía.

y terminamos el algoritmo en cuanto la diferencia en x es 1, que se traduce como

$$mucho - poco \leq 1.$$

Ejercicio 15.23. Con la ayuda del módulo *grbiseccion*, usar el método de la bisección para resolver los siguientes apartados:

a) 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.

b) Encontrar una solución aproximada de $\cos x = x$ y comparar con los resultados del ejercicio 15.12.

c) Obtener una solución aproximada de cada una de las ecuaciones

$$2 - \ln x = x \qquad \text{y} \qquad x^3 \operatorname{sen} x + 1 = 0.$$

✍ La primera ecuación tiene una raíz, y la segunda tiene infinitas. ¶

Otros casos donde no podemos usar el método de Newton directamente es cuando carecemos de una fórmula explícita de la función, como en el siguiente ejercicio.

Ejercicio 15.24 (interés sobre saldo). Maggie quiere comprar una «notebook» y en un folleto de propaganda vio que podía comprar una por $2799 de contado o en 12 cuotas de $339. Si dispone de $2799, ¿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, ya que hay muchas formas de calcularla.

Por ejemplo, una técnica que realizan algunas casas es poner un *precio de contado* (PC), que en el folleto que miró Maggie es $3199, un *precio total financiado* (PTF), que en el folleto es $4068, y luego calculan la *tasa efectiva anual* (TEA) como

$$TEA = \left(\frac{PTF}{PC} - 1\right) \times 100 \approx 27.16,$$

como aparece en el folleto que miró Maggie.

Podemos considerar el precio efectivo de la oferta, PE, como un descuento sobre PC (dando un 12.5% de descuento en este caso), o al revés, podemos pensar en PC como un precio ficticio sobre el real PE (dando un aumento del 14.29%).[(1)]

Finalmente, el PTF ($4068) se calcula tomando la cantidad de cuotas (12) por el monto de cada cuota ($339).

[(1)] Muchas casas hacen un descuento del 10% sobre el precio nominal cuando se paga en efectivo.

a) Hacer las cuentas descriptas en Python.[2]

Pasemos a describir el llamado *interés sobre saldo*.

Supongamos que 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:

$$b_1 = t\,c, \quad c_1 = b_1 - p, \quad b_2 = t\,c_1, \quad c_2 = b_2 - p, \ldots$$

y en general

$$c_m = b_m - p = t\,c_{m-1} - p, \tag{15.7}$$

donde inclusive podríamos poner $c_0 = c$.

b) 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: $c\,r/1200$.

c) Del mismo modo, encontrar una tasa crítica, $r_{\text{crít}}(c, p)$ (dependiendo sólo de c y m), tal que la deuda se mantenga constante, $c_{m+1} = c_m$ para $m = 0, 1, 2, \ldots$

Respuesta: $1200\,p/c$.

d) Programar una función `saldo` 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 $m = 0, 1, 2, \ldots$

Aclaración 1: no se pide encontrar una fórmula, sólo traducir a Python la ecuación (15.7).

Aclaración 2: suponemos $c > 0$, $r \geq 0$, $p > 0$ y $m \geq 0$.

✍ En el ejercicio 15.28 se enuncia una fórmula explícita, acá innecesaria.

e) Usar la función `saldo` con c es 2799 (el precio de la oferta), $r = 27.15\,\%$, $p = 339$ y $m = 12$. ¿Es razonable el resultado?

f) 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 el apartado *b*) porque cada vez se adeuda menos.

Construir una función que dados c, r y p (p mayor que el monto en el apartado *b*)) calcule el número total de cuotas m para llegar a saldo nulo, y luego calcular el valor correspondiente para $c = 2799$, $r = 27.16$ y $p = 339$.

Aclaración: todas las cuotas deben ser iguales, excepto tal vez la última que puede ser menor.

Sugerencia: ¿un lazo `while`?

g) ¿Cómo es el comportamiento de `saldo` cuando sólo varía c?, ¿y si sólo varía r?

Respuesta: crecientes en ambos casos.

h) Usando el módulo *grpc* hacer un gráfico de la función `saldo` cuando $c = 2799$ (lo que pagaría Maggie hoy), $p = 339$ (la mensualidad pedida), $m = 12$ (la cantidad de cuotas) variando r, confirmando lo visto en el apartado anterior. El gráfico debería ser similar al de la figura 15.7.

i) El gráfico anterior muestra que la curva corta al eje r, y podemos 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 *c*)?

Respuesta: podríamos poner `poco = 0` si $p \times m \leq c$, y `mucho = rcrit` siempre (suponiendo $c > 0$ y $p > 0$).

j) Hacer 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.

[2] Sí, se dice *descritas*.

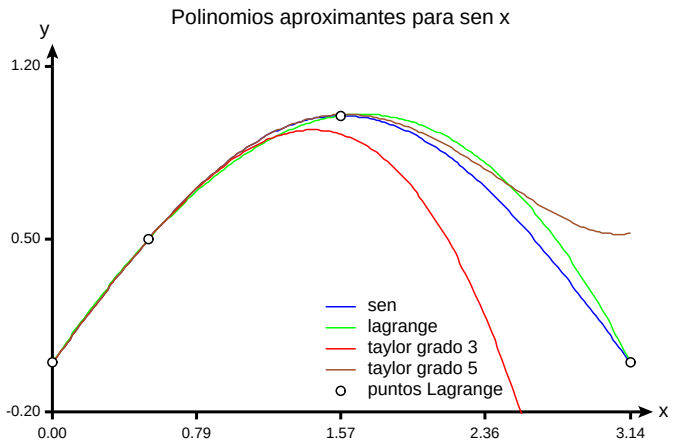


Figura 15.8: Distintas aproximaciones a sen x mediante polinomios.

k) Calcular la tasa (anual nominal) que pagaría Maggie si decide comprar en cuotas.
Respuesta: 75.36 %.

l) Como dispone ahora de $2799, Maggie podría poner ese dinero en certificados a plazo fijo cada mes, extrayendo $339 mensualmente para pagar la cuota. Si el banco ofrece una tasa de 12 % (nominal anual), ¿en cuántos meses quedaría saldo nulo de modo que no pueda renovar el certificado del banco?
Sugerencia: recordar el apartado *f*).
Respuesta: 9 meses.

m) 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 *k*).

n) Suponiendo que la tasa anual de inflación está entre 20 % y 30 %, que los bancos ofrecen una tasa no mayor al 12 % en certificados a plazo fijo, y teniendo en cuenta las tasas en el folleto (27.16 %) y la que encontramos, ¿Maggie debería hacer la compra en efectivo o en cuotas haciendo certificados a plazo fijo en el banco? ¶

15.6. Polinomios

Los polinomios son expresiones de la forma

$$P(x)=\sum_{k=0}^{n} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0, \qquad (15.8)$$

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,

$$1234 = 1\times 10^3 + 2\times 10^2 + 3\times 10 + 4.$$

También 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 15.8 mostramos cómo se podría aproximar a sen x mediante desarrollos de Taylor alrededor de 0 de grados 3 y 5, 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.

Nuestra primera tarea será evaluar un polinomio dado.

Ejercicio 15.25 (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 (15.8).

Hacerlo de tres formas:

a) Calculando la suma de términos como se indica en la ecuación (15.8), calculando x^k para cada k.

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*,

$$((\cdots((a_n\, x\, +\, a_{n-1})\, x\ \, +\, a_{n-2})\, +\cdots)\, x\, +\, a_1)\, x\, +\, a_0. \tag{15.9}$$

✍ En las dos primeras versiones 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 $n-1$. Finalmente, en la regla de Horner, tenemos sólo n sumas y n productos. ¶

Ejercicio 15.26 (polinomios interpoladores de Lagrange). Un polinomio $P(x)$ como en la ecuación (15.8), 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:

$$P(x) = \sum_{i=1}^{n+1} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}. \tag{15.10}$$

✍ 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 (15.10) 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 (15.10), donde los datos son (x_i, y_i), $1 \leq i \leq n+1$, 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 15.8.

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.

❦ *Aunque considerado como francés, Joseph-Louis Lagrange (1736–1813) nació en Turín (Italia) como Giuseppe Lodovico Lagrangia.*

Lagrange fue uno de los fundadores del «cálculo de variaciones» (área relacionada con la mecánica) y las probabilidades, e hizo numerosas contribuciones en otras áreas como astronomía y ecuaciones diferenciales.

Estudiantes de cálculo reconocen su nombre por los «multiplicadores» para encontrar extremos de funciones de varias variables. ¶

Ejercicio 15.27 (escritura en base entera). La idea de la regla de Horner se usa también en el caso de escritura en base b ($b \in \mathbb{Z}$, $b > 1$). Si $n \in \mathbb{Z}$ es positivo y queremos escribir

$$n = \sum_{i=0}^{k} a_i b^i, \quad \text{donde } a_i \in \mathbb{Z} \text{ y } 0 \le a_i < b, \tag{15.11}$$

entonces a_0 se obtiene como resto de la división de n por b, por lo que $b \mid (n - a_0)$, y $n_1 = (n - a_0)/b$ es un entero que tiene resto a_1 cuando dividido por b, entonces $b \mid (n_1 - a_1)$, etc.

✍ Tal vez queda más clara la idea mirando la ecuación (15.9) para la regla de Horner, cambiando x por b: el resto de dividir n por b es a_0, etc.

✍ Usamos este proceso en el ejercicio 9.7.

a) Hacer una función que dados n y b (ambos enteros, $n > 0$ y $b > 1$), encuentre los coeficientes $(a_0, a_1, \ldots, a_k)$ de la expresión (15.11). Por ejemplo, si `n = 10` y `b = 2`, se de obtener `[0, 1, 0, 1]`.

Aclaración: El último coeficiente debe ser no nulo (pedimos $n > 0$). En otras palabras, la longitud de la lista obtenida debe ser la cantidad de cifras de n cuando expresado en base b.

b) Hacer 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, hacer 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`.

Sugerencia: mirar el ejercicio 15.5.*a*).

d) En la ecuación (15.11), ¿cómo se relacionan k y $\log_b n$ (si $a_k \neq 0$)? ¶

Terminamos relacionando algunos conceptos que vimos.

Ejercicio 15.28 (interés sobre saldo II). En el ejercicio 15.24 usamos la expresión (15.7) para expresar la deuda al principio del mes m, c_m.

a) Interpretando (15.7) 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,

$$\sum_{k=0}^{n} x^k = (x^{n+1} - 1)/(x - 1) \quad \text{si } x \neq 1.$$

✍ La expresión explícita de c_m nos permite usar el método de Newton, sin apelar a la bisección. Si los pagos o los períodos no fueran uniformes, tendríamos que volver a la versión original. ¶

15.7. Ejercicios adicionales

Ejercicio 15.29 (métodos elementales de integración). Dos conceptos importantes que se estudian en cursos de cálculo o análisis matemático son el de derivada (como en la relación (15.3)) y el de integral o área encerrada por una curva. Ambos conceptos están relacionados y se pueden pensar como inversos uno del otro, lo que no es sorprendente si pensamos que derivar es como «desacumular» e integrar es como «acumular» en el ejercicio 10.17.

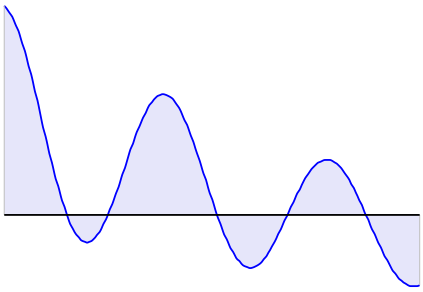


Figura 15.9: f y el área que delimita.

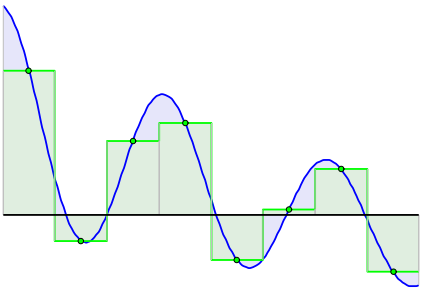


Figura 15.10: Regla del punto medio.

En la figura 15.9 vemos en color celeste el área que queda encerrada por el gráfico de una función y el eje x. Las partes por encima del eje son positivas, mientras las que quedan por debajo son negativas.

En este ejercicio nos preocupamos por establecer aproximaciones a la integral de una función f (continua) en un intervalo $[a,b]$:

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

Hay diversas formas de introducir el concepto de integral de Riemann, pero la idea es dividir el intervalo $[a,b]$ en intervalos más pequeños, $[a_0,a_1]$, $[a_1,a_2]$, … $[a_{n-1},a_n]$, donde $a=a_0<a_1<\cdots<a_n=b$, tomar un punto x_i en el intervalo $[a_{i-1},a_i]$ ($i=1,\ldots,n$) y finalmente aproximar el área determinada por f por la suma de las áreas de los rectángulos,

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n}(a_i-a_{i-1})\,f(x_i). \tag{15.12}$$

Vamos a ver tres reglas sencillas de aproximar la integral, donde supondremos que todos los subintervalos tienen igual longitud, i. e., poniendo $\Delta x=(b-a)/n$, queda

$$a_i = a+i\,\Delta x \quad \text{para } i=0,\ldots,n.$$

La primera regla es la del *punto medio*, donde tomamos

$$x_i=\frac{a_{i-1}+a_i}{2}=a+\frac{i-1+i}{2}\,\Delta x=a+\frac{\Delta x}{2}+(i-1)\Delta x \quad \text{para } i=1,\ldots,n,$$

y de (15.12) se simplifica a

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(a_i')\Delta x=\Delta x\sum_{i=1}^{n} f(x_i). \tag{15.13}$$

En la figura 15.10 hemos dividido el dominio de la f anterior en $n=8$ partes iguales, y los rectángulos con las alturas correspondientes a la función evaluada en los puntos medios.

a) Dadas f, a, b y n, podemos esquematizar la regla del punto medio en Python como:

```
dx = (b - a)/n  # delta x
xs = [a + dx/2 + i*dx for i in range(n)]
dx * sum(f(x) for x in xs)  # la aproximación
```

✍ Repetir n veces el cálculo de `a + dx/2` no es muy eficiente que digamos, pero lo hacemos para no complicar y pensando que n no será muy grande.

Hacer una función con este esquema implementando la regla del punto medio.

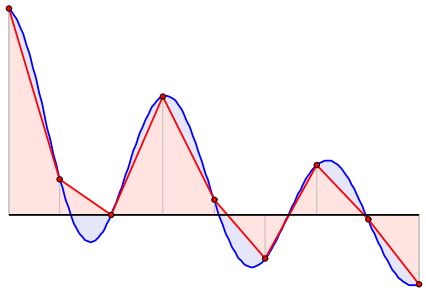


Figura 15.11: Regla del trapecio.

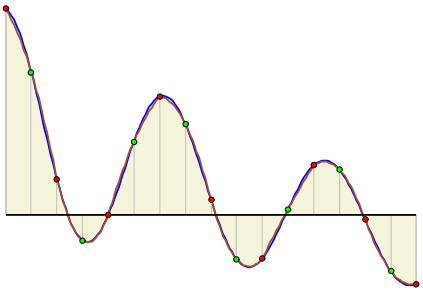


Figura 15.12: Regla de Simpson.

Al ver los rectángulos de la figura 15.10, inmediatamente pensamos en cambiarlos por trapecios, que deberían dar una mejor aproximación. Como vemos en la figura 15.11, no parece haber mejorado la cosa a pesar de las buenas intenciones.

❦ *Se puede demostrar que la regla del punto medio y la del trapecio dan aproximaciones igualmente buenas, siendo el error del orden de* $(\Delta x)^2$ *para funciones «suaves».*

✍ Es completamente distinto si estuviéramos aproximando la longitud de la curva determinada por f en vez del área: la poligonal asociada a los trapecios da mucho mejor aproximación que los segmentos en las bases de los rectángulos, ¡que siempre suman $b-a$!

Desde las matemáticas estamos haciendo la aproximación de la *regla del trapecio*:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \sum_{i=1}^{n} \frac{f(a_{i-1})+f(a_i)}{2}(a_i-a_{i-1}) \\ &= \Delta x \sum_{i=1}^{n} \frac{f(a_{i-1})+f(a_i)}{2},\end{aligned} \tag{15.14}$$

donde en la última igualdad usamos que los intervalos tienen todos longitud Δx. De otra forma, cambiamos $f(\frac{a_{i-1}+a_i}{2})$ en (15.13) por $\frac{f(a_{i-1})+f(a_i)}{2}$ en (15.14).

Podemos simplificar un poco la última expresión en (15.14):

$$\begin{aligned}\sum_{i=1}^{n} \frac{f(a_{i-1})+f(a_i)}{2} &= \frac{f(a_0)+f(a_1)}{2} + \frac{f(a_1)+f(a_2)}{2} + \cdots + \frac{f(a_{n-1})+f(a_n)}{2} \\ &= \frac{f(a_0)}{2} + f(a_1) + f(a_2) + \cdots + f(a_{n-1}) + \frac{f(a_n)}{2} \\ &= \frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(a_i).\end{aligned}$$

b) Hacer una función implementando la regla del trapecio.

Sin desanimarnos por el «fracaso» de los trapecios, intentamos con parábolas. Concretamente, agregamos el punto medio a los intervalos, y pasamos una parábola por estos tres puntos (los extremos y el punto medio de cada intervalo) y estimamos el área dada por la parábola. En la figura 15.12 vemos el efecto, que parece muy bueno.

El problema ahora es calcular el área en el intervalo $[a_{i-1}, a_i]$ determinada por la parábola que pasa por los puntos

$$(a_{i-1}, f(a_{i-1})), \quad (x_i, f(x_i)) \quad \text{y} \quad (a_i, f(a_i)) \qquad (\text{para } i = 1, \ldots, n),$$

donde recordamos que $x_i = (a_{i-1}+a_i)/2$. No es difícil ver que el área buscada es

$$(f(a_{i-1}) + 4f(x_i) + f(a_i))\,\frac{a_i - a_{i-1}}{6},$$

y tenemos la aproximación

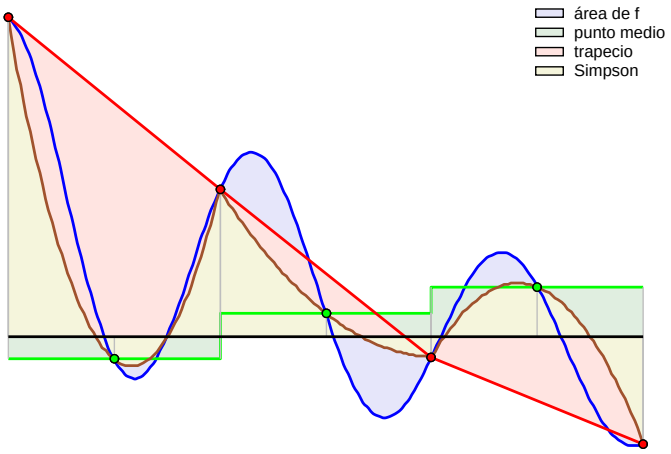


Figura 15.13: Aproximaciones a la integral.

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{6}\sum_{i=1}^{n}(f(a_{i-1})+4f(x_i)+f(a_i)) = \frac{\Delta x}{3}(2r+p), \tag{15.15}$$

donde r es la aproximación (15.13) de los rectángulos por la regla del punto medio y t es la aproximación (15.14) de la regla del trapecio.

La aproximación (15.15) se llama *regla de Simpson*, un promedio ponderado de las otras dos reglas. Se puede demostrar —usando un poco de cálculo— que da mejor aproximación que las otras dos por separado.

❦ *Tomás Simpson (1710–1761) es recordado especialmente por sus trabajos en métodos numéricos. Sin embargo, lo que hoy llamamos «regla de Simpson» se debe a Newton. En contrapartida, la versión actual del método de Newton-Raphson —tal como lo describimos en la sección 15.4— se debe a Simpson, quien lo publicó en 1740.*

En la figura 15.13 mostramos las tres aproximaciones simultáneamente, tomando 3 intervalos en vez de 8 para poder apreciar las diferencias.

c) Hacer una función implementando la regla de Simpson.

d) Se puede demostrar que

$$\int_0^4 \operatorname{sen} x\,dx = 1-\cos(4).$$

Comparar el valor teórico con las aproximaciones del punto medio, trapecio y Simpson para $n=4$ y $n=10$.

❦ *A diferencia de las reglas del punto medio y de los trapecios, la regla de Simpson da una aproximación mucho mejor, con errores del orden de $(\Delta x)^4$ para funciones «suaves».*

¶

Ejercicio 15.30 (potencias binarias). Otra variante de la idea de la regla de Horner (problema 15.25) es el cálculo de una potencia «pura» x^n, con $x\in\mathbb{R}$ y $n\in\mathbb{N}$.

Si no tuviéramos la función `x ** n` de Python, una primera posibilidad es hacer el producto según el esquema

```
p = 1
for i in range(n):
    p = p * x
p
```
(15.16)

usando $n-1$ multiplicaciones, pero podemos hacer algo mejor.

Si $n = 2^k$, tendremos

$$x^n = x^{2^k} = (\cdots((x^2)^2)^2 \cdots)^2,$$

donde en la expresión a la derecha se eleva al cuadrado k veces. Entonces hacemos $k \approx \log_2 n$ multiplicaciones en vez de las $n-1$ en el esquema (15.16).

En general, expresando a n en su desarrollo en base 2,

$$n = \sum_{i=0}^{k} a_i 2^i,$$

pondremos

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

a) Si los coeficientes se conocen y están guardados en `a`, un esquema posible es

```
if a[0] == 1:
    producto = x
else:
    producto = 1
potencia = x
for aj in a[1:]:    # aj = a[j]
    potencia = potencia * potencia  # = x**(2**j)
    if aj == 1:
        producto = producto * potencia
producto
```

Hacer una función con estas ideas (con argumentos `x` y `a`), y compararla con `x**n`.

b) Modificar la función anterior de modo que los argumentos sean `x` y `n` y la función va encontrando los coeficientes.

c) Comparando con el esquema (15.16), ¿cuántas operaciones aritméticas (productos y divisiones) se realizan en cada caso? (Recordar el ejercicio 15.27.*d*)).

✍ Debido a su alta eficiencia, la misma técnica se usa para encriptar mensajes usando números primos de varias decenas de cifras. ¶

Sabemos cómo representar los números naturales como sumas de potencias de 2 (binaria) o 10 (decimal), pero también hay otras formas simples de representar los naturales como sumas, como vemos en el próximo ejercicio. A veces se usan representaciones alternativas para ahorrar espacio y tiempo en los cálculos con números muy grandes.

Ejercicio 15.31 (Zeckendof (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

$$n = b_2 f_2 + b_3 f_3 + \cdots + b_m f_m, \tag{15.17}$$

donde $b_m = 1$, $b_k \in \{0,1\}$, y $b_k \cdot b_{k+1} = 0$ para $k = 2, \ldots, m-1$.

Por ejemplo, $10 = 8+2 = f_6 + f_3$, $27 = 21+5+1 = f_8 + f_5 + f_1$.

Hacer un programa para calcular esa representación (dando por ejemplo una lista de los índices k para los que $b_k = 1$ en la igualdad (15.17)). Comprobar con algunos ejemplos la veracidad de la salida. ¶

Ejercicio 15.32 (El Juego de Nim). Propongamos el siguiente juego:

> *Dos jugadores, A y B, colocan un número arbitrario de fósforos sobre una mesa, separados en filas o grupos. El número de filas y el número de fósforos en cada fila también son arbitrarios. El primer jugador, A, toma cualquier número de fósforos de un fila, pudiendo tomar uno, dos o hasta toda la fila, pero sólo debe modificar una fila. El jugador B juega de manera similar con los fósforos que quedan, y los jugadores van alternándose en sus jugadas. Gana el jugador que saca el último fósforo.*

Veremos que, dependiendo de la posición inicial, uno u otro jugador tiene siempre una estrategia ganadora. Para esto, digamos que una disposición de los fósforos es una *posición ganadora* para el jugador X, si *dejando* X los fósforos en esa posición, entonces no importa cuál sea la jugada del oponente, X puede jugar de forma tal de ganar el juego. Por ejemplo, la posición en la cual hay dos filas con dos fósforos cada una, es ganadora: si A deja esta posición a B, B debe tomar uno o dos fósforos de una fila. Si B toma 2, A toma los dos restantes. Si B toma uno, A toma uno de la otra fila. En cualquier caso, A gana.

a) Ver que la posición en donde hay 3 filas con 1, 2 y 3 fósforos respectivamente, es ganadora.

Para encontrar una estrategia ganadora, formamos una tabla expresando el número de fósforos de cada fila en binario, uno bajo el otro, poniendo en una fila final P si la suma total de la columna correspondiente es par, e I en otro caso. Por ejemplo, en las posiciones anteriores haríamos:

$$\begin{array}{cc} 1 & 0 \\ 1 & 0 \\ \hline P & P \end{array} \qquad \text{y} \qquad \begin{array}{cc} 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ \hline P & P \end{array}$$

En el último ejemplo, es conveniente poner 01 en vez de sólo 1 en la primera fila a fin de tener todas las filas con igual longitud.

Si la suma de cada columna es par decimos que la posición es *correcta*, y en cualquier otro caso decimos que la posición es *incorrecta*. Por ejemplo, la posición donde hay 1, 3 y 4 fósforos es incorrecta:

$$\begin{array}{cc} | & \to \quad 0\ 0\ 1 \\ | \ | & \to \quad 0\ 0\ 1 \\ | \ | \ | & \to \quad 0\ 0\ 1 \\ | \ | \ | \ | & \to \quad 1\ 0\ 0 \\ \hline & \quad I\ I\ P \end{array}$$

b) En este apartado, veremos que una posición en Nim es ganadora si y sólo si es correcta.

i) Si ninguna fila tiene más de un fósforo, la posición es ganadora $\Leftrightarrow$ hay en total un número *par* de fósforos $\Leftrightarrow$ la posición es correcta.

ii) Si un número $a \in \mathbb{N}$, expresado en binario por $(a_n, \ldots, a_0)$ (permitiendo $a_n = 0$) es reemplado por otro menor b (entero ≥ 0), expresado en binario como $(b_n, \ldots, b_0)$, entonces para algún i, $0 \leq i \leq n$, las paridades de a_i y b_i son distintas.

iii) Por lo tanto, si el jugador X recibe una posición correcta, necesariamente la transforma en incorrecta.

iv) Supongamos que estamos en una posición incorrecta, es decir, al menos la suma de una columna es impar. Para fijar ideas supongamos que las paridades de las columnas son

$$P \quad P \quad I \quad P \quad I \quad P$$

Entonces hay al menos un 1 en la tercera columna (la primera con suma impar). Supongamos, otra vez para fijar ideas, que una fila en la cual esto pasa es (en binario)

$$0 \quad 1 \quad \overline{1} \quad 1 \quad \overline{0} \quad 1$$

donde marcamos con $-$ que los números debajo están en columnas de suma impar. Cambiando 0's y 1's por 1's y 0's en las posiciones marcadas, obtenemos el número (menor que el original, expresado en binario):

$$0 \quad 1 \quad \overline{0} \quad 1 \quad \overline{1} \quad 1$$

Está claro que este cambio corresponde a un movimiento permitido, haciendo la suma de cada columna par, y que el argumento es general.

v) Por lo tanto, si el jugador X recibe una posición incorrecta, puede mover de modo de dejar una posición correcta.

vi) Si A deja una posición correcta, B necesariamente la convierte en incorrecta y A puede jugar dejando una posición correcta. Este proceso continuará hasta que cada fila quede vacía o contenga un único fósforo (el caso *i*).

c) En base a los apartados anteriores, y suponiendo que A y B siempre juegan de modo óptimo, ¿quién ganará?

d) Ver que la «jugada ganadora» en *b.iv*) no siempre es única.

e) Desarrollar una función que, ingresada una posición inicial en la forma de una lista con el número de fósforos en cada fila, decida si la posición es correcta o incorrecta, y en este caso encuentre una jugada ganadora.

f) ¿cuál es la estrategia si el que saca el último fósforo es el que *pierde*? ¶

15.8. Comentarios

- El juego de Nim (ejercicio 15.32) está tomado de Hardy y Wright (1975).
- La mayoría de las figuras en este capítulo fueron hechas con el módulo *grpc*.

Capítulo 16

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).

16.1. Notaciones y cuestiones previas

Empezamos dando una descripción de los términos y propiedades que usaremos, dejando para los cursos de matemática discreta las definiciones rigurosas y las demostraciones. En el proceso estableceremos nomenclatura y notaciones que pueden diferir entre autores.

✍ En esta sección hay una «ensalada» de conceptos, varios de los cuales pueden no ser familiares. 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.

Los vértices se indican por V y las 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, $V = \{1, 2, \ldots, n\}$. 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 de la forma $\{v, v\}$ uniendo un vértice con sí mismo, ni aristas «paralelas» uniendo los mismos 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 $m \leq \binom{n}{2}$.

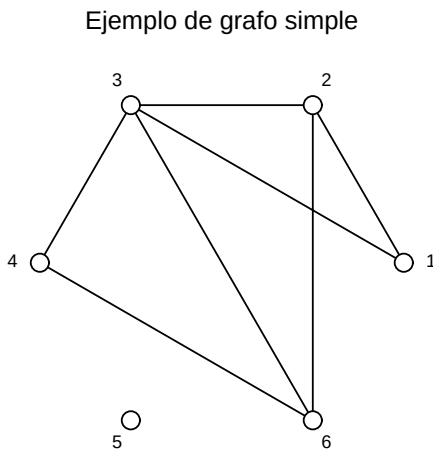


Figura 16.1: Un grafo con 6 vértices y 7 aristas. El vértice 5 es aislado.

En la figura 16.1 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 $(v_1, v_2, \ldots, v_k)$, donde $\{v_i, v_{i+1}\} \in E$ para $i = 1, \ldots, k-1$. Claro que podemos describir un camino tanto por los vértices como por las aristas intermedias, y en vez de poner $(v_1, v_2, \ldots, v_k)$ podríamos considerar $(\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\})$. La *longitud* del camino $(v_1, v_2, \ldots, v_k)$ es $k-1$, la cantidad de aristas que tiene (y *no* la cantidad de vértices k).

Si $u = v_1$ y $v = v_k$, decimos que el camino $(v_1, v_2, \ldots, v_k)$ es un *camino de u a v*, o que *une u y v*, o sencillamente *camino u–v*.

Un camino en principio puede tener vértices repetidos, y si se cierra sobre sí mismo de modo que $v_1 = v_k$, decimos que se trata de un camino *cerrado*, mientras que si no tiene vértices repetidos decimos que es un camino *simple*. Un *ciclo* es un camino cerrado sin aristas repetidas (pero puede tener varios vértices repetidos).

Por ejemplo, en la figura 16.1:

- $(3, 2, 6, 3, 4)$ es un camino 3–4 de longitud 4,
- $(1, 2, 3)$ es un camino simple,
- $(1, 2, 3, 6, 2, 3, 1)$ es un camino cerrado (pero no ciclo),
- $(4, 3, 2, 1, 3, 6, 4)$ 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 « $\sim$ » definida en $V \times V$ por $u \sim v$ si y sólo si $u = v$ o existe un camino u–v, es una relación de equivalencia, y las clases de equivalencia se llaman *componentes conexas* o simplemente *componentes* del grafo. Por ejemplo, el grafo de la figura 16.1 no es conexo, pues tiene un vértice aislado, y sus componentes son $\{1, 2, 3, 4, 6\}$ y $\{5\}$.

Por otra parte, si un grafo es conexo pero no tiene ciclos, decimos que es un *árbol*.

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 $n^2/2$.[(1)]

Dada su estructura, es más sencillo trabajar con árboles que con grafos. Como hemos dicho, 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):

(1) Más precisamente, entre $n-1$ y $n(n-1)/2$.

16.1. Teorema (Caracterizaciones de árboles). Un grafo simple $G=(V,E)$ con $|V|=n$ es un árbol si y sólo si se cumple alguna de las siguientes condiciones:

a) Para cualquier $a,b\in V$ *existe* un *único* camino que los une.
b) G es conexo y $|E|=n-1$.
c) G no tiene ciclos y $|E|=n-1$.
d) G es conexo, y si se agrega una arista entre dos vértices cualesquiera, se crea un único ciclo.
e) G es conexo, y si se quita cualquier arista —pero no los vértices en los que incide— 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* 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 $E'=\{\{u,v\}\in E : u,v\in V'\}$.

16.2. 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 $1,2,\dots,n$, y nos falta guardar la información de las aristas. Hay muchas formas de hacerlo, pero nosotros 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.

✍ Otras dos formas usuales son la *matriz de adyacencias*, una matriz cuyas entradas son sólo 0 o 1 y de modo que la entrada ij es 1 si y sólo si $\{i,j\}\in E$, y la *matriz de incidencias*, una matriz de $m\times n$, también binaria, donde la entrada ij es 1 si y sólo si la arista i-ésima es incidente sobre el vértice j.

Las más de las veces el ingreso de datos es más sencillo mediante la lista de aristas, pues la matriz de adyacencias tiene n^2 elementos y en general $m\ll n^2$ (y siempre $m\le n(n-1)/2$ para grafos simples).

Ejemplo 16.2. Para el grafo de la figura 16.1, 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 `[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 13 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.

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 16.3. Hacer una función para que dada la lista de vecinos, se imprima para cada vértice, los vértices que son adyacentes.

Por ejemplo, si la entrada es el grafo del ejemplo 16.1, 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 16.4. Como es útil pasar de una representación a otra, definimos las funciones `dearistasavecinos` y `devecinosaaristas` en el módulo *grafos*.

Comprobar el comportamiento de estas funciones con las entradas del ejemplo 16.2, pasando de una a otra representación (en ambos sentidos).

Observar que:

- 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.
- Las inicializaciones de `vecinos` y `aristas` en cada caso.
- La construcción `for u, v in aristas` en `dearistasavecinos`, aún cuando cada arista está representada por una lista. ¶

Ejercicio 16.5. A fin de evitar errores cuando ingresamos datos y no escribir tanto, es conveniente guardar los datos del grafo en un archivo de texto. Por comodidad (y uniformidad), supondremos que el archivo de texto contiene en la primera línea el número de vértices, y en las restantes los vértices de cada arista: *por lo menos debe tener un renglón* (correspondiente al número de vértices), y *a partir del segundo debe haber exactamente dos datos* por renglón.

Por ejemplo, para el grafo del ejemplo 16.2, el archivo tendría (ni más ni menos):

```
6
1  2
1  3
2  3
2  6
3  4
3  6
4  6
```

Hacer una función que toma como argumento el nombre de un archivo de textos 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 16.6 (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).

Por ejemplo, en el grafo del ejemplo 16.2, los grados son $\delta(1)=2, \delta(2)=3, \delta(3)=4, \delta(4)=2, \delta(5)=0, \delta(6)=3$.

a) Hacer una función que dado un grafo 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

$$\sum_{v \in U} \delta(v) \quad \text{es par.}$$

Hacer una función para hacer este cálculo y verificar el teorema. ¶

Ejercicio 16.7. Es claro que si $(u = v_1, v_2, \ldots, v_k = v)$ es un camino u–v, entonces $(v_k, v_{k-1}, \ldots, v_1, v_0)$ es un camino v–u, y lo mismo para un ciclo.

Hacer una función que ingresando un grafo (de un archivo de texto) y una sucesión de vértices $(v_1, v_2, \ldots, v_k)$ (ingresada interactivamente, $k > 1$):

a) decida si $(v_1, v_2, \ldots, v_k)$ es un camino, es decir, si $\{v_{i-1}, v_i\} \in E$ para $i = 2, \ldots, k$,

b) en caso afirmativo, imprima el camino inverso, $(v_k, v_{k-1}, \ldots, v_1, v_0)$,

c) verifique si $(v_1, v_2, \ldots, v_k)$ es un ciclo. ¶

Ejercicio 16.8. El módulo *grgr* (gráficos de grafos) permite hacer ilustraciones de grafos como el de la figura 16.1. El comportamiento es similar al del módulo *grpc*, aunque una diferencia importante es que en *grgr* se pueden mover los vértices y etiquetas con el ratón.

Como otros módulos gráficos que vimos, *grgrsimple* es una plantilla para realizar ilustraciones, en este caso con *grgr*. 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. ¶

16.3. 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. 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`,[(2)] y cualquier otro vértice tendrá `padre[v]` $\neq$ `v`.

[(2)] Como siempre, no ponemos tildes en los identificadores para evitar problemas.

Algoritmo *recorrido*

Entrada: un grafo $G = (V, E)$ y un vértice $r \in V$ (la raíz)
Salida: la lista de vértices que se pueden alcanzar desde r «visitados»

Poner $Q = \{r\}$
mientras $Q \neq \emptyset$:
  sea $i \in Q$
  sacar i de Q
  «visitar» i
  para todo j adyacente a i:
    si j no está «visitado» y $j \notin Q$:
      agregar j a Q
      poner $padre(j) = i$

Cuadro 16.1: Esquema del algoritmo `recorrido`.

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 16.1 mostramos un esquema informal, donde la cola se llama Q y la raíz r.

Hay distintos tipos de cola, y a nosotros nos interesarán las siguientes:

Cola *lifo* (last in, first out) o *pila*: el último elemento en ingresar (*last in*) es 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:

```
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
x = cola.pop(0)  # para colas fifo
```

  Más adelante veremos cómo elegir el elemento a sacar en las colas de prioridad que usaremos.

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` en un vértice si y sólo si no se puede llegar a él desde la raíz.

También podemos ver que la cola se ha implementado como pila pues sale el último en ingresar.

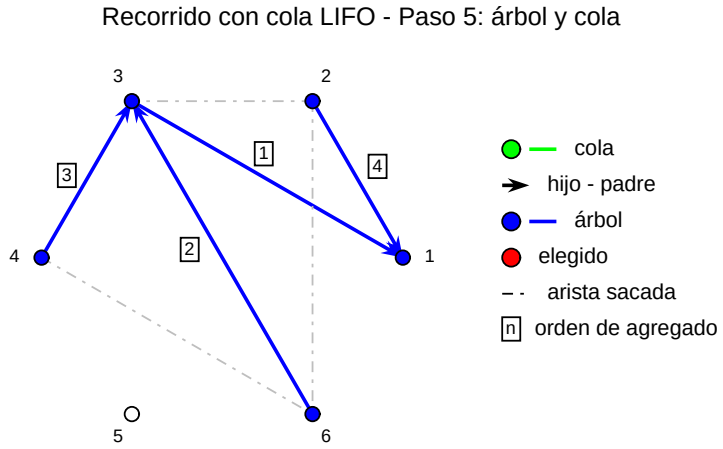


Figura 16.2: Recorrido *lifo* del grafo del ejemplo 16.2 tomando raíz 1.

Ejercicio 16.9 (recorrido de un grafo).

a) Estudiar la función `recorrido` y comprobar el comportamiento para el grafo del ejemplo 16.2 tomando distintos vértices como raíz, por ejemplo 1 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 en este caso. La figura 16.2 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 in vecinos[u] if padre[v] == None]
cola.extend[lista]
for v in lista:
   padre[v] = u
```

? ¶

Ejercicio 16.10. 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 16.2 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 16.11 (componentes). En el ejercicio 16.9 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 16.2 las aristas $\{3,5\}$ y $\{4,5\}$, de modo que ahora 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. Hacer 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`, hacer una función que retorne una lista de las componentes de un grafo. En el grafo original del ejemplo 16.2 el resultado debe ser algo como `[[1, 2, 3, 4, 6], [5]]`. ¶

En la figura 16.2 podemos ver que las aristas elegidas por la función `recorrido` forman un árbol. Usando la raíz 1 en el grafo del ejemplo 16.2, 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 16.12. Agregar instrucciones a `recorrido`, de modo que en vez de retornar los vértices del árbol obtenido, se retorne la lista de aristas que forman el árbol. ¶

Ejercicio 16.13. Hacer sendas funciones para los siguientes apartados dado un grafo G:

a) Ingresando 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 16.1 y el ejercicio 16.11.*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. ¶

Ejercicio 16.14 (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 16.6, hacer 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 16.21).

❦ *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 el 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 fue quien originó el estudio de teoría de grafos y la topología al resolver en 1736 el famoso problema de los puentes de Königsberg (hoy Kaliningrad, en Rusia), donde el río Pregel se bifurcaba dejando dos islas (y dos costas), y las islas y las costas se unían por siete puentes. Euler resolvió el problema, demostrando que no se podían recorrer todos los puentes pasando una única vez por ellos, demostrando el teorema que mencionamos en el problema. ¶

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*

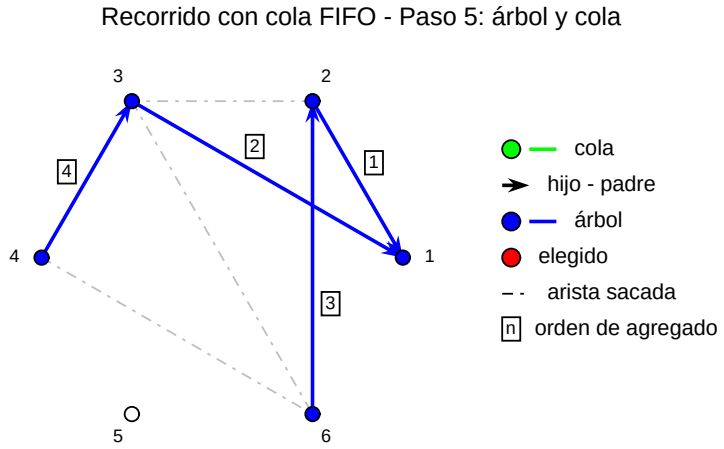


Figura 16.3: Recorrido *fifo* del grafo del ejemplo 16.2 tomando raíz 1.

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. La programación es bastante más complicada en ese caso, y podríamos ponerla como:

```
while len(cola) > 0:
    u = cola[-1]    # tomar el último de la cola
    while vecinos[u] != []:  # si queda algún vecino
        v = vecinos[u].pop() # sacarlo de vecinos[u]
        # y no considerar u como vecino de v
        vecinos[v].remove(u)
        if padre[v] == None: # si v nunca estuvo en
            cola.append(v)   # la cola, incorporarlo
            padre[v] = u     # guardar de dónde vino
            break            # y salir de este lazo
    if u != cola[-1]:  # si no se agregaron vecinos
        cola.pop()     # eliminarlo de la cola
```

En el caso del grafo del ejemplo 16.2 tomando raíz 1 en esta versión el orden de visita $(1,3,6,4,2)$. En la página del libro hay una «película» que ilustra el algoritmo.

El algoritmo es un poco más ineficiente que usar una cola lifo porque tenemos que sacar a `u` de `vecinos[v]` (recorriendo la lista `vecinos[v]` hasta encontrar `u`), y tenemos que acceder a `vecinos[u]` varias veces hasta vaciarlo (mientras que en nuestra presentación se accede una única vez).

✍ 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 16.15 (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.

✍ 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 cualquier po-

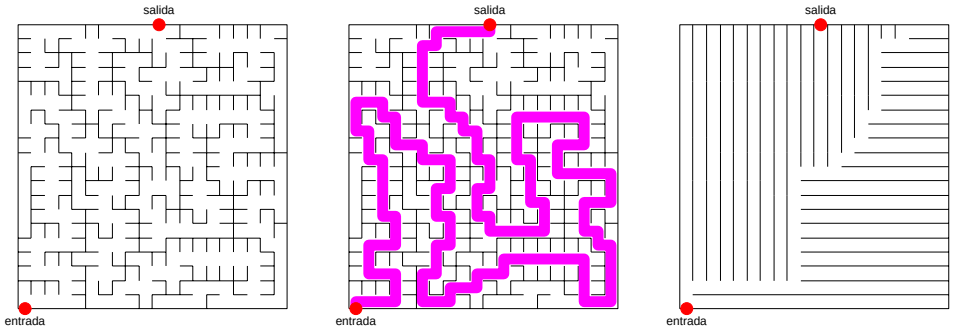


Figura 16.4: Laberinto (izquierda) y solución (centro) usando pila (lifo), y laberinto usando cola fifo (derecha).

sición (usando las funciones `lista.insert(lugar, x)`, `lista.remove(x)` o `lista.pop(lugar)`).

Para los ejemplos del curso no hace diferencia. El módulo estándar *collections* implementa listas fifo eficientemente en *collections.deque* (ver el *manual de la biblioteca*). Nosotros no lo veremos en el curso.

b) Repetir el ejercicio 16.10, comparando las diferencias entre el recorrido en profundidad y el recorrido a lo ancho.

✍ En la página del libro se muestra una «película» (archivo pdf) de cómo se va construyendo el árbol. La figura 16.3 es la última página de ese archivo, destacando el orden en que las aristas se incorporan a él cuando la raíz es 1 (comparar con la figura 16.2). ¶

Una forma dramática de mostrar las diferencias de recorridos lifo o fifo es construyendo laberintos como los de la figura 16.4.

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.

Se dan numeraciones aleatorias a las listas de vecinos después de construirlas (con `random.shuffle`), y la raíz (entrada) se toma aleatoriamente sobre el borde inferior. Finalmente, se construyen los árboles correspondientes usando el algoritmo `recorrido`.

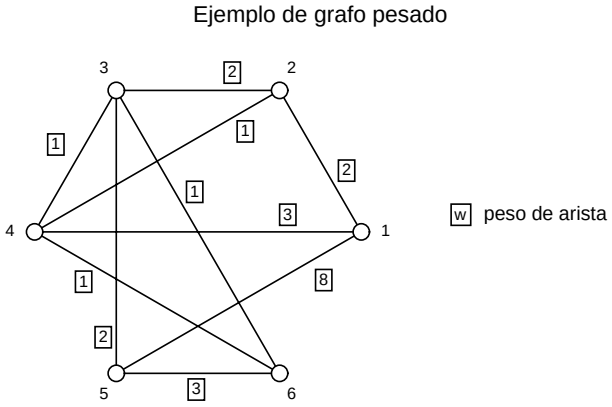
Como se trata de árboles, sabemos que hay un único camino entre cualquier par de vértices. Tomando aleatoriamente un punto de salida sobre el borde superior, podemos construir el único camino entre la entrada y la salida, lo que constituye la «solución» del laberinto.

En la figura 16.4 a la izquierda mostramos el árbol obtenido para cola lifo y a la derecha para cola fifo, para $m = n = 20$, usando una misma semilla (con `random.seed`) para tener las mismas listas de vecinos. En la parte central de la figura mostramos la «solución» del laberinto a la izquierda (cola lifo).

Observamos una diferencia notable en la calidad de los árboles. El recorrido con cola fifo o *ancho primero* produce un árbol mucho más «regular». Por supuesto, los árboles tienen la misma cantidad de aristas, pero los caminos obtenidos con recorrido en profundidad primero produce caminos mucho más largos.

16.4. Grafos con pesos

Algunas veces la información interesante de un grafo está en los vértices, y las aristas nos dicen que, por alguna razón, dos vértices están relacionados y nada más. Otras veces, las aristas tienen información adicional que queremos considerar. Por ejemplo si las aristas indican rutas entre ciudades, la información podría ser la distancia entre las ciudades. Cuando cada arista $e \in E$ de un grafo tiene un *costo* o *peso* asociado, w_e,



Arista e	peso w_e
{1,2}	2
{1,4}	3
{1,5}	8
{2,3}	2
{2,4}	1
{3,4}	1
{3,5}	2
{3,6}	1
{4,6}	1
{5,6}	3

Figura 16.5: Un grafo con pesos en las aristas.

decimos que se trata de un grafo *con pesos* o *pesado*. Consideraremos aquí sólo el caso $w_e > 0$ para todo $e \in E$.

En la figura 16.5 vemos un ejemplo de grafo con pesos, marcados con recuadros en las aristas correspondientes, y que por comodidad y claridad reproducimos al costado.

Pensando que los vértices representan ciudades y los costos de las aristas representan distancias, si tuviéramos que ir de una ciudad a otra, teniendo distintas rutas alternativas para elegir, es razonable preguntarse cuál de ellas será la más corta (o la más barata). Éste es el *problema del camino más corto*: en un grafo pesado, y dados dos vértices $s, t \in V$, encontrar un camino $(s = v_1, v_2, \ldots, v_k = t)$ con peso total mínimo, donde el peso total de un camino se define como

$$w_{\{v_1,v_2\}} + w_{\{v_2,v_3\}} + \cdots + w_{\{v_{k-1},v_k\}} = \sum_{i=1}^{k-1} w_{\{v_i,v_{i+1}\}}.$$

Observemos que el valor de k no está fijo: no nos interesa si tenemos que usar una arista o cien, sólo nos interesa que la suma de las distancias para ir de s a t sea mínima.

Por ejemplo, en el grafo de la figura 16.5 podemos usar varios caminos para ir del vértice 1 al 5: el camino $(1,5)$ usa una única arista $(k = 2)$ y tiene peso 8, el camino $(1,2,3,5)$ usa 3 aristas y tiene costo total $2+2+2 = 6$, y en realidad no hay otro con menor costo. Sin embargo, el camino $(1,4,3,5)$ también tiene costo $3+1+2 = 6$, es decir, puede haber más de un camino con distancia mínima.

Siguiendo con el ejemplo de las ciudades, otro problema interesante es pensar que inicialmente las ciudades no están unidas, los pesos de las aristas representan ahora los costos de construcción de los caminos correspondientes, y queremos construir carreteras de modo de que todas las ciudades puedan unirse entre sí mediante estas nuevas rutas, pero el costo de la construcción sea lo menor posible.

El problema es entonces encontrar un subconjunto de aristas que, junto con los vértices originales, formen un subgrafo generador conexo, y tal que la suma de los costos de ese subconjunto de aristas sea lo menor posible.

Claro que si el grafo original no es conexo, no podremos encontrar un subgrafo conexo (y generador). Por otro lado, no pueden formarse ciclos porque podríamos sacar del ciclo la arista con mayor costo manteniendo la conexión.

> ✍ Este razonamiento es correcto si en el ciclo hay alguna arista con costo estrictamente positivo, pero podríamos tener problemas si tienen costo cero o negativo (nosotros suponemos que todas las aristas tienen costo positivo). Por ejemplo, si hay un ciclo de costo negativo, recorriendo el ciclo reduciremos el costo total, y si lo repetimos muchas veces...

Es decir, queremos un subgrafo generador, conexo y sin ciclos, y por lo tanto el conjunto de aristas a elegir debe forman un árbol generador, que haga mínima la suma

de los pesos de las aristas que lo componen,

$$\sum_{e \text{ en el árbol}} w_e.$$

Un árbol con estas propiedades se llama *árbol generador mínimo*. Podría haber más de un árbol con costo mínimo, por ejemplo si todos los costos de todas las aristas son 1 y el grafo no es un árbol.

Volviendo al grafo de la figura 16.5, podemos formar un árbol generador con las aristas $\{1,2\},\{1,4\},\{2,3\},\{3,6\},\{5,6\}$ (siempre un árbol generador debe tener $n-1$ aristas), con peso total $2+3+2+1+3=11$, y si reemplazamos la arista $\{5,6\}$ por la arista $\{3,5\}$, reducimos el costo en 1. En este ejemplo, dados los datos, se forma un ciclo donde todas las aristas tienen peso 1, y por lo tanto hay más de un árbol mínimo (de peso total 7): ¿podrías encontrar dos de ellos a simple vista?

Antes de resolver estos problemas (el de la ruta más corta y el del mínimo árbol generador), hacemos cambios necesarios para incorporar los pesos en las aristas.

Ejercicio 16.16.

a) En el ingreso de datos el único cambio es agregar el peso a las aristas como tercera componente. Para el grafo de la figura 16.5, podríamos poner

```
ngrafo = 6  # cantidad de vértices
aristas = [[1, 2, 2], [1, 4, 3], [1, 5, 8],
           [2, 3, 2], [2, 4, 1], [3, 4, 1],
           [3, 5, 2], [3, 6, 1], [4, 6, 1],
           [5, 6, 3]]
```

b) Como cada vez que probemos algún programa tendremos que volver a ingresar datos, es conveniente guardarlos en un archivo de texto, como hicimos en el ejercicio 16.5. Modificar lo hecho en ese ejercicio para incorporar los pesos. Por ejemplo, el grafo de la figura 16.5 tendría asociado un archivo con las entradas:

```
6
1  2  2
1  4  3
1  5  8
2  3  2
2  4  1
3  4  1
3  5  2
3  6  1
4  6  1
5  6  3
```

¶

Ejercicio 16.17. La figura 16.5 se hizo con el módulo *grgrpesado*, similar al *grgrsimple* que vimos en el ejercicio 16.8 pero para ilustrar grafos pesados. Ejecutar el módulo y comprobar su funcionamiento, moviendo vértices y etiquetas o cambiando el grafo. ¶

16.5. Camino más corto: Dijkstra

Tal vez el algoritmo más conocido para resolver el problema del camino más corto sea el de Dijkstra, que sigue la estructura de la función `recorrido`: se comienza desde un vértice, en este caso s, se lo coloca en una cola, y se visitan los vértices de la cola.

✍ Hay otros algoritmos para encontrar el camino más corto entre un par de vértices. En particular, el de Floyd-Warshall encuentra todos los caminos entre cualquier par de vértices aún en presencia de aristas con costos negativos (o determina que hay un ciclo de peso negativo).

❦ *E. W. Dijkstra (1930–2002) nació y murió en Holanda. Fue uno de los más grandes intelectos que contribuyeron a la lógica matemática subyacente en los programas de computación y sistemas operativos. Entre sus muchas y destacadas contribuciones está el algoritmo para el camino más corto que presentamos, publicado en 1959.*

Nuevamente habrá tres clases de vértices: los visitados, los que están en la cola y no han sido aún visitados, y los que nunca estuvieron en la cola. Como novedad, para cada $v \in V$ consideraremos el valor $\text{dist}(v)$ de la distancia más corta de s a v mediante caminos de la forma $(s = v_1, v_2, \ldots, v_k, v)$ donde todos los vértices excepto v ya han sido visitados. A fin de indicar que no existe tal camino, pondremos $\text{dist}(v) = \infty$, siendo éste el valor inicial para todo v (cuando no hay vértices visitados).

A diferencia de los recorridos de grafos que hicimos anteriormente, donde sale el último o el primero de la cola, en el algoritmo de Dijkstra elegimos el vértice de la cola con menor distancia $\text{dist}(v)$. Así, la cola no es de tipo lifo o fifo sino *de prioridad*.

✍ No vamos a implementar colas de prioridad eficientemente, nos limitaremos a encontrar el mínimo de dist en la cola y extraerlo, agregando elementos a la cola siempre al final.

La estructura de *montón* (*heap* en inglés) es más apropiada para este tipo de colas. Como es de esperar, Python tiene un módulo estándar implementando esta estructura.

El algoritmo termina cuando t es el vértice que se visita, y entonces $\text{dist}(t)$ es la distancia del menor camino de s a t, o cuando la cola es vacía, en cuyo caso no existe un camino desde s hacia t y necesariamente el grafo no es conexo.

Al visitar un vértice u y examinar un vértice vecino v, verificamos si

$$\text{dist}(u) + w_{\{u,v\}} < \text{dist}(v). \tag{16.1}$$

Si esta desigualdad es válida, quiere decir que el camino más corto para ir desde s a v (sólo por vértices ya visitados) es ir desde s a u con el camino para u, y luego usar la arista $\{u, v\}$. Por lo tanto, actualizamos $\text{dist}(v)$ poniendo $\text{dist}(v) = \text{dist}(u) + w_{\{u,v\}}$. También agregaremos v a la cola si no se ha agregado aún (o, equivalentemente, si $\text{dist}(v) = \infty$).

Es una propiedad del algoritmo (que no demostraremos) que si v es un vecino de u —el vértice que se está visitando— y v ya se ha visitado, la desigualdad (16.1) no puede darse.

✍ Para demostrar esta propiedad —y que el algoritmo es correcto— se usa de forma esencial que $w_e > 0$ para todo $e \in E$. Nosotros dejaremos estas propiedades para cursos de matemática discreta o teoría de grafos.

Ejercicio 16.18. La función `dijkstra` (en el módulo *grafos*) implementa el algoritmo de Dijkstra para encontrar el camino más corto entre s y t.

a) Verificar que para el grafo de la figura 16.5, el camino más corto entre 1 y 5 tiene distancia total 6.

✍ En la página del libro hay una «película» (archivo pdf) de cómo se va construyendo el árbol y luego el camino. La figura 16.6 muestra la última página de ese archivo, donde podemos observar el árbol en azul, las flechas indican el sentido *hijo-padre*, y en magenta observamos el camino t–s que hay que revertir.

b) Agregar instrucciones para construir el camino s–t usando `padre`.

Sugerencia: recordar el ejercicio 16.13. En el ejemplo del apartado anterior, el camino recorrido es $(1, 2, 3, 5)$.

c) Cambiar la función de modo que calcule las distancias mínimas de s (ingresado por el usuario) a todos los otros vértices del grafo (sólo las distancias, no los caminos).

d) Modificar la función de modo que imprima una tabla de todas las distancias de u a v, $1 \le u, v \le n$. ❡

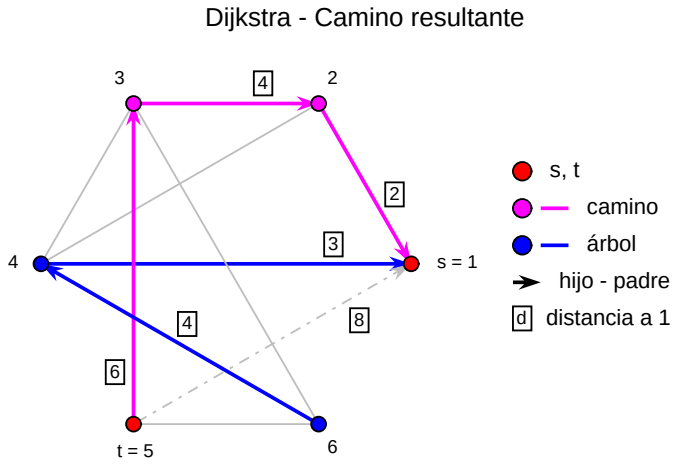


Figura 16.6: Resultado del algoritmo de Dijkstra.

16.6. Mínimo árbol generador: Prim

Hay varios algoritmos para encontrar un árbol generador mínimo. Nosotros veremos aquí el debido a Prim, pues podemos ponerlo dentro de un esquema similar a la función `recorrido`.

✍ En la mayoría de los casos prácticos, el algoritmo de Kruskal es más eficiente que el de Prim. No lo veremos pues su implementación es bastante más elaborada.

❦ *El algoritmo de Kruskal fue publicado en 1956, mientras que el de Prim fue publicado en 1959. Dijkstra también obtuvo en forma independiente el algoritmo de Prim y lo publicó en 1959, en el mismo trabajo donde presenta su algoritmo para el camino más corto, lo que no es sorpresa dada la similitud.*

Prácticamente todos los algoritmos para encontrar un árbol generador de mínimo peso se basan en ir agregando a un conjunto de aristas, una de menor costo que no forme un ciclo con las ya elegidas.

En el algoritmo de Prim, empezamos con un vértice r (la raíz), y vamos agregando aristas manteniendo siempre un árbol $T = (V(T), E(T))$. Al principio, T tiene un único vértice (r) y no tiene aristas.

Durante el algoritmo se van modificando T y el conjunto Q de vértices que se conectan a T pero no están en T. Q es una cola, de donde se elige un nuevo elemento u para incorporar a T, y luego se examinan los vecinos de u para eventualmente incorporarlos a Q.

El algoritmo termina cuando se han examinado todos los vértices conectados con r y $Q = \emptyset$.

A fin de hacer eficiente el algoritmo, se considera una función dist, similar a la del algoritmo de Dijkstra, que para $v \in Q \setminus V(T)$ es el peso de la arista con menor peso que conecta v con T. Una vez que v ingresa a $V(T)$, dist(v) no cambia.

Podemos caracterizar los vértices que no están en $V(T) \cup Q$ poniendo dist(v) $= \infty$, de modo que decir dist(v) $< \infty$ es lo mismo que decir que v está o estuvo en Q.

Esta estructura es similar a la de la función `recorrido`, en el que se forman tres clases de vértices: los visitados, los que están en la cola, y los que nunca ingresaron a la cola. En el algoritmo de Prim se sigue la misma idea, los vértices visitados son los de $V(T)$, y «visitar» el vértice u significa agregar u a $V(T)$ y actualizar otros datos de T, Q y dist que se necesiten.

Como en el algoritmo de Dijkstra, la cola es de prioridad. Sin embargo, la actualización de la distancia para los vecinos del vértice u visitado en el algoritmo de Dijkstra se hace de acuerdo a la desigualdad (16.1), mientras que ahora tendremos que analizar la

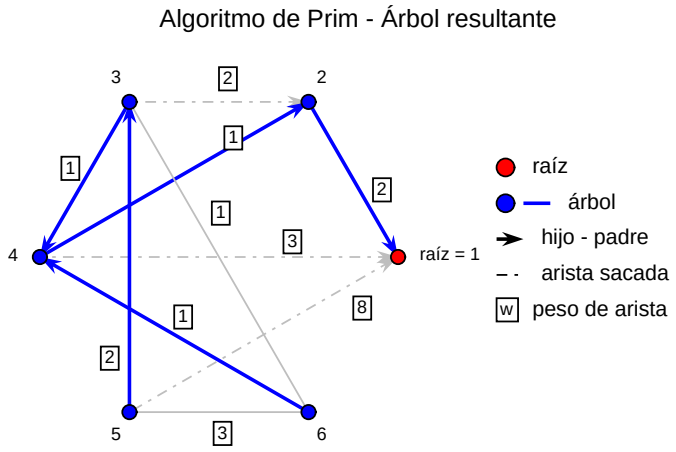


Figura 16.7: Resultado del algoritmo de Prim.

desigualdad

$$w_{\{u,v\}} < \text{dist}(v), \tag{16.2}$$

y cambiar dist(v) a $w_{\{u,v\}}$ en caso de ser válida.

Otra diferencia importante es que, en el algoritmo de Dijkstra, si v ya se ha visitado y se está visitando u, la desigualdad (16.1) no puede darse, pero en el algoritmo de Prim podría darse la desigualdad (16.2). De modo que para cada $v \in V$ debemos mantener información sobre si $v \in V(T)$ o no.

Veamos otras similitudes y diferencias:

- El valor *infinito* y la construcción de vecinos son idénticos.
- En `prim` «no existen» s y t, ni por supuesto el camino que los une, en cambio tenemos una raíz. En `dijkstra` s es el primero en ingresar a la cola mientras que ahora es la raíz.
- La distancia se actualiza comparando `dist[v]` con el peso de la arista $\{u, v\}$ en vez de con la suma entre ese peso y `dist[u]`.
- Al terminar el algoritmo de Prim, debemos decidir si T es un árbol generador, en cuyo caso será de mínimo costo, o no (y será $|V(T)| < |V|$).

Ejercicio 16.19. La función `prim` (en el módulo *grafos*) implementa el algoritmo de Prim para encontrar el mínimo árbol generador, a partir de un vértice inicial `raiz`.

a) Verificar que el árbol mínimo generador tiene peso 7 para el grafo de la figura 16.5, tomando como vértices iniciales 1 y 6.

b) Modificar la función para que retorne también la lista de aristas que forman el árbol mínimo. Por ejemplo, en el grafo del apartado anterior tomando `raiz = 1` se obtiene el árbol de aristas $\{1,2\}$, $\{4,3\}$, $\{2,4\}$, $\{3,5\}$ y $\{4,6\}$.
Sugerencia: recordar el ejercicio 16.12.
Verificar que tomando $r = 6$, el árbol mínimo obtenido es distinto (pero los costos son iguales, claro).

✍ En la página del libro hay una «película» de cómo se va construyendo el árbol cuando el vértice inicial es 1. La figura 16.7 es la última página de ese archivo. ¶

16.7. Ejercicios Adicionales

Ejercicio 16.20. Construir laberintos de $m \times n$ como los de la figura 16.4 (ver descripción correspondiente). ¶

Ejercicio 16.21 (ciclo de Euler II). En el ejercicio 16.14 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 y 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».

Hacer una función para decidir si un grafo es conexo y todos los vértices tienen grado par (ejercicio 16.14), y en caso afirmativo construir e imprimir un ciclo de Euler. ¶

16.8. Comentarios

- Hemos tratado de presentar `recorrido`, `dijkstra` y `prim` lo más parecidas entre sí a fin de resaltar las semejanzas. Es muy posible que las veas con un disfraz bastante distinto en otras referencias.
- Las figuras de los laberintos (figura 16.4) se hicieron con el módulo *grpc*. Las restantes figuras en este capítulo se hicieron con el módulo *grgr*, así como las «películas» en la página del libro para ilustrar los distintos algoritmos.

Capítulo 17

Recursión

17.1. Introducción

Recordando las sumas de Gauss (ejercicio 10.12),

$$s_n = 1+2+\cdots+n,$$

supongamos que queremos encontrar todas las sumas

$$s_1 = 1, \quad s_2 = 1+2, \quad s_3 = 1+2+3, \quad \ldots \quad s_n = 1+2+\cdots+n. \tag{17.1}$$

✍ Este tipo de trabajo lo hicimos en el ejercicio 10.17.

Podríamos calcular cada una de ellas separadamente, por ejemplo usando la fórmula de Gauss $s_n = n\times(n+1)/2$, pero también podríamos poner

$$s_1 = 1, \quad s_2 = s_1+2, \quad s_3 = s_2+3, \quad \ldots \quad s_n = s_{n-1}+n, \tag{17.2}$$

Claro que no habría mucha diferencia con el esquema que vimos en el ejercicio 10.12, en el que poniendo la variable `suma` como lista tendríamos

```
suma = [0 for i in range(n+1)]
for i in range(1, n+1):
    suma[i] = suma[i-1] + i
```

calculando en cada paso del lazo `for` la cantidad s_i.

Cambiando suma por producto en (17.2) (y el valor inicial de 0 a 1 porque es un producto), obtenemos el factorial

$$1! = 1, \quad 2! = 1!\times 2, \quad 3! = 2!\times 3, \quad \ldots \quad n! = (n-1)!\times n,$$

donde también reemplazamos s_k por $k!$.

En realidad *es usual definir* $n!$ como

$$0! = 1 \qquad \text{y} \qquad n! = n\times(n-1)! \quad \text{para } n\in\mathbb{N}. \tag{17.3}$$

Estas dos condiciones, la de valor inicial y la «fórmula» para obtener los siguientes, nos permiten calcular $n!$ para cualquier $n\in\mathbb{N}$. Por ejemplo, si queremos calcular 4! usando la ecuación 17.3, tendríamos sucesivamente:

$$\begin{aligned} 4! &= 4\times 3! = 4\times(3\times 2!) = 4\times 3\times(2\times 1!) = \\ &= 4\times 3\times 2\times(1\times 0!) = 4\times 3\times 2\times 1\times 1 = 24. \end{aligned}$$

Cuando se dan uno o más valores iniciales y una «fórmula» para calcular los valores subsiguientes, decimos que se ha dado una *relación de recurrencia*. Estas relaciones dan lugar a lo que se llama *inducción* en matemática y *recursión* en programación.

Ya hemos visto otras relaciones de recurrencia además de las sumas de Gauss y el factorial. Por ejemplo, el cálculo de la función $c_m = \mathit{saldo}(c, r, p, m)$ en el ejercicio 15.24, según la ecuación (15.7) que repetimos por comodidad:

$$c_m = \begin{cases} t\, c_{m-1} - p & \text{si } m > 0, \\ c & \text{si } m = 0. \end{cases} \tag{17.4}$$

Los números de Fibonacci también satisfacen una relación de recurrencia, dada por

$$f_1 = 1, \qquad f_2 = 1, \qquad \text{y} \qquad f_n = f_{n-1} + f_{n-2} \quad \text{para } n \geq 3,$$

que es la ecuación (10.20).

Veamos cómo se traducen estas ideas a la programación.

17.2. Funciones definidas recursivamente

La función `factorial` (en el módulo *recursion*), es una implementación recursiva del factorial, basada en la ecuación (17.3): se da el valor inicial para $n = 1$ y el valor para n mayores depende del valor anterior y n.

El esquema dentro del bloque es

```
if n == 1:      # valor inicial o base
    return 1
return n * factorial(n - 1) # depende del anterior
```

y es equivalente a

```
if n == 1:
    return 1
else:
    return n * factorial(n - 1)
```

o

```
if n > 1:
    return n * factorial(n - 1)
return 1
```

Ejercicio 17.1. Explicar por qué los tres esquemas anteriores son equivalentes. ¶

Veamos ejemplos parecidos que antes resolvíamos con un lazo `for` o similar, y que ahora podemos resolver usando recursión.

Ejercicio 17.2. En cada uno de los siguientes casos, definir funciones que usen recursión en vez de lazos `for` o `while` y compararlas con las definiciones con lazos.

a) Calcular x^n para n entero no negativo, usando

$$x^0 = 1, \quad x^n = x^{n-1} \times x \quad \text{para } n \in \mathbb{N}.$$

b) Calcular el mínimo de una lista (vector) $(a_1, \ldots, a_n)$ definiéndolo para $n = 2$ con `if` y usando que

$$\text{mín}(a_1, a_2, \ldots, a_n) = \text{mín}\{\text{mín}(a_1, a_2, \ldots, a_{n-1}), a_n\}$$

para $n > 2$.

En este caso el valor inicial corresponde a $n = 2$ y no a $n = 1$.

c) Calcular la función `saldo` (del ejercicio 15.24) según (17.4). ¶

Ejercicio 17.3. Usar la siguiente función recursiva para imprimir una lista (un elemento por renglón):

```
def imprimir(a, n):
    """Imprimir hasta n elementos de una lista."""
    if n > 1:
        imprimir(a, n-1)
    if 0 < n <= len(a):
        print(a[n-1])
```

Observar que no es claro cuál es el paso base, ya que no hay un `else` en el bloque.

a) ¿Qué pasa si se cambia el orden de las instrucciones (o sea, primero el grupo `if n <=...` y después el grupo `if n >...`)?

b) ¿Qué pasa si `n` es mayor que `len(a)`?, ¿y si `n` es `0`? ¶

La recursión se puede usar con más de un argumento, es decir, *no siempre se llama a la función con* $n-1$:

- La función `fibonacci` (en el módulo *recursion*) hace dos llamadas con argumentos $n-1$ y $n-2$.
- La función `mcd` (en el módulo *recursion*) es una implementación de la versión original de Euclides para encontrar el máximo común divisor entre dos enteros positivos, y hace llamadas con argumentos que disminuyen en más de 1 (en general).

17.3. Variables no locales

Muchas veces en recursión vamos a tener definida una función dentro de otra, y —por ejemplo— para saber cuán eficiente es un algoritmo nos gustaría cuántas veces se llama a la función interna.

Ejercicio 17.4. Supongamos que tenemos una función definida dentro de otra, y que queremos ver las veces que se usa esta función. Un primer intento es

```
def llamar(n):
    """Llamar n veces a una función interna."""

    def aumenta():  # función interna
        c = c + 1   # que incrementa el contador

    c = 0           # inicializar el contador
    for i in range(n):  # y hacer n llamadas
        aumenta()

    # acá debería ser c == n
    print('Se hicieron', c, 'llamadas')
```
(17.5)

a) Ver que el esquema anterior da error.

b) Para arreglarlo, podríamos declarar a `c` como `global` en algún lado.
Probar haciendo la declaración `glocal c`:

i) Dentro de la función `aumenta`, pero no en `llamar`.

ii) Dentro de la función `llamar`, pero no en `aumenta`.

iii) Dentro de ambas funciones.

¿Cuáles versiones no dan error?

c) El problema con la única posibilidad del apartado anterior que no da error es que tenemos que poner la variable como global, el identificador puede coincidir con el de otra variable, y se puede modificar accidentalmente, como vimos en el ejercicio 6.7.
Por ejemplo, declarar `c` como `global` en ambas funciones y poner:

```
c = -1
llamar(5)
c
```
(17.6)

comprobando que el valor de `c` dejó de ser -1.

d) La solución que ofrece Python es declarar la variable `c` como *no local*. A diferencia de la declaración `global`, al poner `nonlocal c` se busca el contexto más pequeño en donde se ha hecho una asignación a `c` (o se la ha declarado como `global`).

Así, cambiamos la definición de la función interna en el esquema (17.5) por

```
def aumenta():  # función interna
    nonlocal c  # nueva declaración
    c = c + 1   # que incrementa el contador
```

La variable `c` se asigna en el contexto de `llamar` (y no está declarada allí como global), que es el contexto más pequeño que contiene al contexto de `aumenta`.

Hacer este cambio y volver a ejecutar las instrucciones en (17.6), comprobando que no hay errores y el valor final de `c` es -1.

✍ El esquema final está guardado en el módulo *nolocal*. ¶

17.4. 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 6.1.

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 la copia de la función que ha sido llamada por la recursión termina su tarea, el espacio 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 $n!$, 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, siguiendo la misma estructura de las pilas o colas lifo que vimos en sección 16.3. Python impone un límite de unas 1000 llamadas para esa pila.

❦ *Un resultado de computación teórica dice que toda función recursiva puede reescribirse sin usar recursión con lazos* `while` *y listas que se usan como pilas para ir guardando los datos intermedios.*

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:

$$1,\ 1,\ 2(=1+1),\ 3(=2+1),\ 5(=3+2),\ 8(=5+3),\ldots$$

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:

$$\begin{aligned} f_5 &= f_4 + f_3 \\ f_4 &= f_3 + f_2 \\ f_3 &= f_2 + f_1 \\ f_2 &\to 1 \\ f_1 &\to 1 \end{aligned}$$

$$\begin{aligned} f_2 &\to 1 \\ f_3 = f_2 + f_1 & \\ f_2 &\to 1 \\ f_1 &\to 1 \end{aligned}$$

realizando un total de 8 llamadas (haciendo otras tantas copias) además de la inicial.

Ejercicio 17.5. Agregar un contador a `recursion.fibonacci` para averiguar cuántas veces se la llama, y encontrar la cantidad de llamadas al calcular f_5, f_{10} y f_{20}.

✍ Recordar el ejercicio 17.4. ¶

En los ejemplos que vimos hasta ahora es mucho más eficiente y sencillo usar un lazo (`while` o `for`). La ventaja de la recursión es que nos permite resolver algunos problemas muy complicados en forma sencilla.

Por ejemplo, las versiones recursivas de los números de Fibonacci o del algoritmo de Euclides podrían considerarse como más naturales, si no más sencillas, que las versiones con lazos. Veamos otros ejemplos, donde la alternativa de recursión parece más conveniente.

17.5. Los Grandes Clásicos de la Recursión

En todo curso de programación que se precie de tal, entre los ejemplos de recursión se encuentran el *factorial*, los *números de Fibonacci*, y las *torres de Hanoi*. Ya vimos el factorial y los números de Fibonacci... o sea...

Ejercicio 17.6 (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 los discos estaban todos 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 hacer 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, tenemos que pasar los $n-1$ 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 c a b, debemos pasar $n-2$ discos de c a a, pasar el disco $n-1$ a b, y luego pasar $n-2$ discos de a a b. Y así sucesivamente.

Esto lo podemos expresar con una función `pasar` que podríamos poner como:

```
def pasar(n, x, y, z):
    """Pasar los discos 1...n de x a y usando z."""

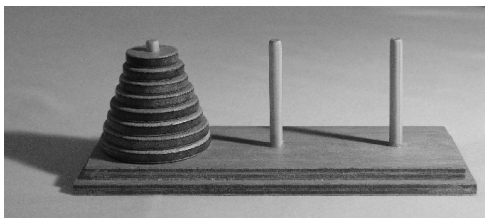
    if n > 1:
        pasar(n - 1, x, z, y)
        print('pasar el disco', n, 'de', x, 'a', y)
        pasar(n - 1, z, y, x)
    else:   # único disco
        print('pasar el disco 1 de', x, 'a', y)
```

donde genéricamente denotamos las agujas por `x`, `y` y `z`.

En la función `hanoi` (en el módulo *recursion*), pusimos a `pasar` como una función interna.

a) Estudiar la función `pasar` y su uso dentro de `hanoi`.

Tapa del juego original



Fotografía del juego

Figura 17.1: Las torres de Hanoi.

b) Agregar un contador (no local) para contar la cantidad de veces que se transfiere un disco de una aguja a otra (en `pasar`), e imprimirlo 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: $2^n - 1 = 1 - 2 + 2^n = 1 + 2(2^{n-1} - 1)$.

c) Suponiendo que transfieren un disco por segundo, ¿cuánto tardarán los monjes en transferir los 64 discos? ¿Cuántos años tardaría una computadora en calcular la solución para $n = 64$, suponiendo que tarda un nanosegundo por movimiento[1] (nano = dividir por mil millones)? Bajo la misma suposición sobre la velocidad de la computadora, ¿cuál es el valor máximo de n para calcular los movimientos en 1 minuto?

❦ *«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 17.1, usualmente se les da a los chicos con entre 4 y 6 discos, a veces de distintos colores. Notablemente, variantes del juego se usan en tratamiento e investigación de psicología y neuro-psicología.
Lucas es más conocido matemáticamente por su test de primalidad —variantes del cual son muy usadas en la actualidad— para determinar si un número es primo o no.

✍ Python tiene una ilustración animada del problema con 6 discos en el módulo *minimal_hanoi*.

✍ 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 17.1 fueron tomadas respectivamente de[2]

- `http://www.cs.wm.edu/~pkstoc/`
- `http://en.wikipedia.org/wiki/Tower_of_Hanoi` ¶

17.6. Contando objetos combinatorios

Ejercicio 17.7. Para $m, n \in \mathbb{N}$, consideremos una cuadrícula rectangular de dimensiones $m \times n$ (4×3 en la figura 17.2), e imaginémosnos que se trata de un mapa, donde los segmentos son calles y los puntos remarcados son las intersecciones.

Nos preguntamos de cuántas maneras podremos ir desde la esquina más hacia el sudoeste, de coordenadas $(0,0)$, a la esquina más hacia el noreste, de coordenadas (m,n), si estamos limitados a recorrer las calles únicamente en sentido oeste–este o sur–norte, según corresponda.

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

[1] ¡Y que no hay cortes de luz!
[2] Enlaces válidos a marzo de 2012.

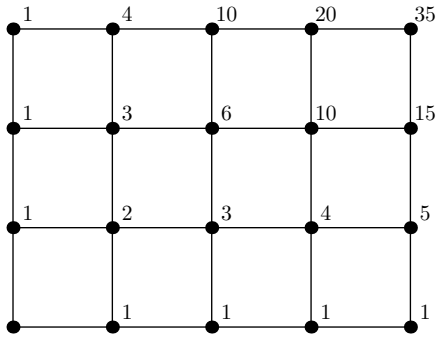


Figura 17.2: Contando la cantidad de caminos posibles.

de caminos llegando desde el sur (si se puede). Los números en la figura 17.2 indican, para cada intersección, la cantidad de caminos para llegar allí desde $(0,0)$ mediante movimientos permitidos.

a) Hacer una función para calcular recursivamente la cantidad $h(m,n)$ de caminos para llegar desde $(0,0)$ a (m,n), donde m y n son enteros positivos.

b) En cursos de matemática discreta se demuestra que

$$h(m,n) = \frac{(m+n)!}{m!\,n!} = \binom{m+n}{m} = \frac{(m+n)\times(m+n-1)\times\cdots\times(m+1)}{n\times(n-1)\times\cdots\times 1},$$

el número combinatorio que vimos en el ejercicio 10.16.

Comparar los valores de la función definida allí con los resultados del apartado anterior.

✍ Podemos comparar el rectángulo de la figura 17.2 con el triángulo de Pascal, del ejercicio 11.6 y figura 11.2, pensando que el triángulo es rectángulo en el vértice superior, y luego rotándolo de modo que el lado izquierdo quede «apoyado» horizontalmente y el ángulo recto a la izquierda.

c) 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 $h(r,s)=0$.

d) 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)). Hacer una función para esta nueva posibilidad.

Sugerencia: puedo armar un camino de $(0,0)$ a (m,n) tomando cualquier camino de $(0,0)$ a (r,s) y después cualquier camino de (r,s) a (m,n).

e) De acuerdo a *b)*, la cantidad de caminos en *d)* es $h(r,s)\times h(m-r,n-s)$. Verificar que esto es cierto.

f) Análogamente, la cantidad de caminos en *c)* es $h(m,n)-h(r,s)\times h(m-r,n-s)$. Verificar que esto es cierto. ¶

Ejercicio 17.8. 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 $0\le y<n$). ¶

17.7. Generando objetos combinatorios

Un problema muy distinto al de *contar* objetos, como hicimos con la cantidad de caminos del ejercicio 17.7, es *generarlos*, por ejemplo para encontrar alguno o todos los que satisfacen cierto criterio, y aquí es donde recursión muestra toda su potencia.

Empezamos por encontrar todos los subconjuntos y permutaciones de $\{1,\dots,n\}$, poniendo en lo que sigue $I_n = \{1,\dots,n\}$ por comodidad. El número de objetos a generar puede ser muy grande, como 2^n o $n!$, e independientemente de la eficiencia del algoritmo que usemos para generarlos, debemos tener en cuenta:

- difícilmente tenga sentido tener todos los objetos simultáneamente en una única y enorme lista, que llamaremos la «gran lista»,
- en general basta inspeccionar los objetos uno por uno,

por lo tanto:

> *no deben usarse construcciones como las de los siguientes tres ejercicios.*

Ejercicio 17.9 (subconjuntos I). Para poner todos los subconjuntos de I_n en una «gran lista», podemos 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.

✍ Es la idea que se usa normalmente para demostrar que I_n tiene 2^n subconjuntos.

```
def subconjuntosNO(n):
    """Generar los subconjuntos de {1,..., n}."""

    if n == 0:          # un único subconjunto
        return [[]]     # el conjunto vacío

    subs = subconjuntosNO(n-1)
    return subs + [s + [n] for s in subs]
```

¶

Ejercicio 17.10 (subconjuntos con k elementos I). Para ponerlos todos en una «gran lista», observamos 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 $0 < k < n$.

```
def subconjuntosnkNO(n, k):
    """Generar los subconjuntos de {1,..., n}
    con k elementos."""

    if k == 0:          # no hay elementos
        return [[]]     # conjunto vacío

    if n == k:          # un único subconjunto
        return [list(range(1, n+1))]    # el total

    # acá es 0 < k < n
    a = subconjuntosnkNO(n-1, k-1)
    b = subconjuntosnkNO(n-1, k)
    return b + [s + [n] for s in a]
```

¶

Ejercicio 17.11 (permutaciones I). Para poner todas las permutaciones en una «gran lista», 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, al final, o entremedio).

✍ 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.

```
def permutacionesNO(n):
    """Generar las permutaciones de {1,...,n}."""

    if n == 1:
        return [[1]]

    ps = permutacionesNO(n-1)
    return [p[:i] + [n] + p[i:] # intercalar n
            for p in ps         # en cada permutación de n-1
            for i in range(n)   # variando posición
            ]                                                    ¶
```

Ejercicio 17.12. Los algoritmos en los ejercicios anteriores son interesantes en cuanto se parecen a demostraciones de matemáticas, pero sólo deben usarse en condiciones extremas.

a) Una forma de entender por qué insistimos en lo absurdo de tener a todos los objetos en una «gran lista» es calcular los valores de 2^n, $\binom{n}{\lfloor n/2 \rfloor}$ y $n!$ para $n = 5, 10, 20$.

b) Probar las funciones de los ejercicios anteriores para $n = 0, 4, 8$ y $k = n/2$, comprobando que las longitudes dan los valores correctos (2^n, $\binom{n}{k}$ y $n!$, respectivamente). ¶

Antes que generar todos los objetos simultáneamente y ponerlos en la «gran lista», conviene ir generándolos de a uno y hacer con el resultado alguna acción, que genéricamente llamamos «hacer algo». La idea es usar una única lista que se modifica a medida que se construyen los objetos. Una estructura particularmente útil para esta tarea es la de *pila*, que vimos en la sección 16.3.

✍ En el módulo estándar *itertools* de Python se generan permutaciones, combinaciones, y otros objetos combinatorios que veremos en esta sección como iteradores no explícitos, similares a `range`, justamente para no tener la «gran lista». No vemos este módulo en el curso.

Ejercicio 17.13 (cadenas de bits). La función `cadenasdebits` (en el módulo *recursion*), genera todas las cadenas de bits (listas de 0 y 1) de longitud n, $n \in \mathbb{N}$, para «hacer algo» con cada una de ellas, por ejemplo imprimirlas.

En la función consideramos una (única) lista `a` donde ponemos las cadenas que construimos (y que se destruyen al construir nuevas cadenas). Dado que las cadenas de bits de longitud n se obtienen agregando un 0 o un 1 a las cadenas de longitud $n-1$, para construirlas podemos usamos una función interna `cadena` en las que se agrega el 0 o el 1, llamándose a sí misma en cada caso si hay más lugares (bits) para poner. En la parte principal se hace una única llamada a `cadena(1)` y hay que inicializar `a = []`.

Observar que usamos a como pila. Para k fijo, el elemento en la posición k tiene un valor de 0 cuando $i = 0$, que se mantiene para valores superiores a k pero luego es cambiado para $i = 1$, y obviamente cambia varias veces cuando es llamado desde valores menores a k.

a) Usando `cadenasdebits`, imprimir todas las cadenas de 1, 2, 3 y 4 bits (llamando, por ejemplo, a `cadenasdebits(3, print)`).

b) Para entender mejor el comportamiento de `cadenasdebits` podemos imprimir `k` y `a` en cada llamada de `cadena`. Hacer este cambio (en `cadena`) y ver la salida para $n = 1, 2, 3$.

c) Definir una función apropiada para usar como segundo argumento de la función `cadenasdebits` de modo de contar las cadenas de longitud n (en vez de imprimirlas con `print`). Verificar que hay 2^n cadenas, por ejemplo para $1 \le n \le 10$.

d) En la versión propuesta de `cadenasdebits` se va «hacia adelante», llamando a `cadena(1)` en el cuerpo principal, y aumentando el valor del argumento `k` en cada llamada de la función hasta llegar a $k = n$. Esto contrasta con el uso de recursión en, por ejemplo, el factorial, donde vamos «hacia atrás» disminuyendo el valor del argumento en cada llamada.

Redefinir `cadena` de modo de comparar `k` con 0 (o 1) en vez de n, y se haga la llamada `cadena(n)` en el cuerpo principal. ¶

La técnica que acabamos de ver es bastante general y puede usarse en muchos problemas. Sin embargo, para encontrar las cadenas de bits no es necesario usar recursión.

Ejercicio 17.14. Resolver el ejercicio 17.13 sin usar recursión, usando que las cadenas de bits de longitud n pueden pensarse como los coeficientes en base 2 de los números entre 0 y $2^n - 1$.

Sugerencia: para $k = 0, \ldots, 2^n - 1$, construir la lista de coeficientes en base 2, recordando el ejercicio 15.27. ¶

Ejercicio 17.15 (subconjuntos II). En el ejercicio 17.13 (o el 17.14) esencialmente se construyen todos los subconjuntos de I_n, ya que las cadenas de bits de longitud n se pueden considerar como *vectores característicos*. Dado un conjunto $A \subset \{1, 2, \ldots, n\}$ definimos su vector característico, $b(A) = (b_1, b_2, \ldots, b_n)$, mediante

$$b_i = \begin{cases} 1 & \text{si } i \in A, \\ 0 & \text{si no.} \end{cases}$$

✍ Es claro que dos conjuntos distintos tienen vectores característicos distintos, y que por otro lado, dada una cadena de bits podemos encontrar un conjunto A tal que $b(A)$ sea esa cadena. Por lo tanto, hay una correspondencia biunívoca entre cadenas de bits de longitud n y subconjuntos de I_n.

Construir una función adecuada de modo que ingresada como segundo argumento a `cadenasdebits` haga que se imprima el subconjunto correspondiente en I_n, representando al conjunto vacío con una raya « `-` ». Por ejemplo, para $n = 2$ debería imprimirse algo como (el orden no es importante):

```
-
2
1
1 2
```
¶

Una vez que sabemos cómo generar una familia de objetos, es más sencillo generar o contar objetos de la familia con características particulares, como en el siguiente ejercicio.

Ejercicio 17.16. Supongamos que queremos calcular la cantidad $h(n)$ de cadenas de n bits *que no contienen dos ceros sucesivos*:

a) Usando `cadenasdebits`, hacer una función para evaluar $h(n)$ para $n \in \mathbb{N}$.

✍ Recordar los ejercicios 12.6 y 17.4.

b) Comparar el número $h(n)$ obtenido anteriormente con los números de Fibonacci y hacer una nueva función para calcular $h(n)$ directamente.

✍ Para demostrarlo, podemos usar que las cadenas de n bits consideradas que terminan en 1 se pueden poner en correspondencia biunívoca con las cadenas de $n-1$ bits sin ceros consecutivos (agregándo o sacando 1), y las que terminan en 0 se pueden poner en correspondencia biunívoca con las de $n-2$ bits (agregando o sacando 10). ¶

Habiendo adquirido experiencia con las pilas para generar objetos combinatorios, en el módulo *recursion* definimos las funciones `subconjuntos`, `subconjuntosnk` y `permutaciones`. En todas usamos la estructura de pila —como hicimos en la función

`cadenasdebits`— para ir generando los objetos de a uno y «hacer algo» con ellos. Así, reemplazamos las funciones de los ejercicios 17.9, 17.10 y 17.11, respectivamente, por otras que siguen básicamente el mismo algoritmo pero son más eficientes en el uso de la memoria.

Veamos las características de cada una, observando que las llamadas en el cuerpo principal son del tipo `f(1)`.

`subconjuntos`: aquí la función sobre la que se realiza la recurrencia es `ponerono(m)`, que decide si agregar (o no) m al subconjunto. Reemplaza a la función `cadena` en `cadenasdebits`: en vez de poner 0 o 1, ponemos o no al elemento m.

`subconjuntosnk`: Nuevamente tenemos la función `ponerono(m)`, que es como la anterior pero miramos además si el cardinal del conjunto obtenido nos permite agregar algo.

✍ De alguna forma imitamos lo hecho en el ejercicio 17.16: se agrega (o cuenta en ese ejercicio) sólo si satisface cierta condición.

`permutaciones`: Ahora la función se llama `poner(m)`, que coloca a m en las m posiciones de una permutación de $m-1$ elementos.

Ejercicio 17.17. Comparar estas funcionas con las similares de los ejercicios 17.9, 17.10 y 17.11:

a) Estudiando si se sigue un algoritmo similar en cada caso.

b) Imprimiendo (con `print` como segundo argumento) y comparando con los resultados obtenidos en el ejercicio 17.12.*b*). ¶

Ejercicio 17.18. Considerando la función `permutaciones`:

a) Construir una función que entrada como segundo argumento haga que se imprima cada permutación con su número de orden. Por ejemplo, si $n=3$ que se imprima algo como:

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

Hay 6 permutaciones
```

b) Modificar la función original de modo que aparezca primero la permutación ordenada y al final la ordenada al revés. Por ejemplo, para $n=3$ (con segundo argumento `print`) que se imprima

```
[1, 2, 3]
[1, 3, 2]
[3, 1, 2]
[2, 1, 3]
[2, 3, 1]
[3, 2, 1]
```

Aclaración: ¡no debe generarse la «gran lista» de $n!$ permutaciones!
Sugerencia: cambiar el rango (`range`) en `poner`. ¶

Ejercicio 17.19 (el problema del viajante). Supongamos que un viajante tiene que recorrer n ciudades (exactamente) volviendo a la de partida, sin repetir su visita a ninguna (salvo la inicial), y que el costo de viajar desde la ciudad i a la ciudad j es $c_{ij} \geq 0$. El *problema del viajante* es encontrar una permutación $(a_1, a_2, \ldots, a_n)$ de $(1, 2, \ldots, n)$ de modo que el costo total del recorrido, $c_{a_1 a_2} + c_{a_2 a_3} + \cdots + c_{a_{n-1} a_n} + c_{a_n a_1}$, sea mínimo. Como se recorre un ciclo, es suficiente tomar $a_n = 1$.

a) Usando la función `permutaciones` o una variante, hacer una función para resolver el problema cuando el costo c_{ij} es:

i) $i+j$.
ii) $|i-j|$.
iii) $i \times j$.
iv) dado como entero aleatorio entre 1 y 10.

Sugerencia: definir una variable `costomin` y una lista `opt` donde se guardarán el costo más chico obtenido y la permutación correspondiente, y poner inicialmente `costomin` a un valor muy grande e inalcanzable. A medida que se van recorriendo las permutaciones, calcular el costo de la permutación y compararlo con `costomin`, cambiando `costomin` y `opt` acordemente. Tomar n pequeño, 4 o 5 al principio, mientras se prueba la función, y no mayor que 10–12 en cualquier caso.

b) Ver que en el caso *i)* del apartado anterior, el costo es el mismo para cualquier permutación.
Respuesta: $n \times (n+1)$.
Ayuda: comparar con las sumas de Gauss (ejercicio 10.12).

c) ¿Podrías decir cuál es el óptimo en el caso *ii)* para cualquier n?
Respuesta: $2 \times (n-1)$.
Ayuda: podemos pensar que los puntos $1, \ldots, n$ están sobre una recta y los costos son las distancias.

✍ Como estamos trabajando con grafos simples, los costos son simétricos, es decir, $c_{ij} = c_{ji}$ (como en los ejemplos *i)*–*iii)*). Como recorrer un ciclo en uno u otro sentido no cambia el costo, estamos trabajando (al menos) el doble de lo necesario con la propuesta.

❦ *El problema del viajante y otros similares son sumamente importantes y se aplican a problemas de recorridos en general. Por ejemplo, para «ruteo» de vehículos, máquinas para perforar o atornillar, circuitos impresos y «chips», etc., donde el costo puede medirse en dinero, tiempo, longitud, etc. Consecuentemente, hay toda una área de las matemáticas y computación dedicada al estudio de su resolución eficiente.*

Hasta el momento no se conocen algoritmos verdaderamente buenos, resolviéndose exactamente el caso de unos pocos miles de «ciudades» y costos arbitrarios. Aunque se sospecha que no hay algoritmos «eficientes», aún no se ha demostrado que no los hay (para esto, hay que desarrollar toda una teoría matemática de qué significa «eficiente»).

El algoritmo propuesto acá es quizás el menos *eficiente, pues —como la técnica de barrido en la sección 14.2— analiza todas las soluciones posibles, y por eso se lo llama* de búsqueda exhaustiva. ¶

Ejercicio 17.20. Hacer una función para imprimir todos los caminos que hemos contado en el ejercicio 17.7, con el número de orden en que se van construyendo.

Por ejemplo, cuando $m = 2$ y $n = 1$ se imprima

```
1:  (0,0) (1,0) (2,0) (2,1)
2:  (0,0) (1,0) (1,1) (2,1)
3:  (0,0) (0,1) (1,1) (2,1)

Hay 3 caminos
```
¶

17.8. Ejercicios adicionales

Ejercicio 17.21 (preferencias). Es muy común que al hacer un estudio de mercado se pregunte a los encuestados sus preferencias entre una serie de productos.

Si se pidieran preferencias estrictas —prohibiendo empates— y hay n productos, habría $n!$ preferencias posibles (para cada encuestado), pero si se permiten empates, hay muchas más.

Por ejemplo, para 2 productos a y b hay 3 posibilidades:

$$a \prec b, \quad a \sim b, \quad b \prec a,$$

donde $a \prec b$ significa que a se prefiere a b, y $a \sim b$ significa que a y b están empatados. Del mismo modo, para 3 productos a, b y c hay 13 posibilidades,

$$\begin{gathered} a \prec b \prec c, \quad a \prec c \prec b, \quad b \prec a \prec c, \quad b \prec c \prec a, \quad c \prec a \prec b, \quad c \prec b \prec a, \\ a \prec b \sim c, \quad b \sim c \prec a, \quad a \sim c \prec b, \quad b \prec a \sim c, \quad a \sim b \prec c, \quad c \prec a \sim b, \\ a \sim b \sim c, \end{gathered}$$

siendo cada una de las $6 = 3!$ primeras un orden estricto.

a) Hacer una función para contar la cantidad de órdenes posibles.

✍ Es la sucesión A000670 en The On-Line Encyclopedia of Integer Sequences.
Allí se describen distintos lugares donde aparecen estos números, así como los primeros términos de la sucesión.

b) ¿Podrías generar todos los órdenes posibles sin hacer la «gran lista»? ¶

Bibliografía

A. Engel, 1993. *Exploring Mathematics with your computer.* The Mathematical Association of America. (*págs. 4 y 100*)

E. Gentile, 1991. *Aritmética elemental en la formación matemática.* Red Olímpica. (*pág. 4*)

G. H. Hardy y E. M. Wright, 1975. *An Introduction to the Theory of Numbers.* Oxford University Press, 4 ed. (*pág. 125*)

B. W. Kernighan y D. M. Ritchie, 1991. *El lenguaje de programación C.* Prentice-Hall Hispanoamericana, 2 ed. (*pág. 4*)

D. E. Knuth, 1997a. *The art of computer programming. Vol. 1, Fundamental Algorithms.* Addison-Wesley, 3 ed. (*pág. 4*)

D. E. Knuth, 1997b. *The art of computer programming. Vol. 2, Seminumerical algorithms.* Addison-Wesley, 3 ed. (*págs. 4 y 75*)

D. E. Knuth, 1998. *The art of computer programming. Vol. 3, Sorting and searching.* Addison-Wesley, 2 ed. (*págs. 4, 80 y 88*)

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. 4 y 17*)

N. Wirth, 1987. *Algoritmos y Estructuras de Datos.* Prentice-Hall Hispanoamericana. (*págs. 4, 80 y 88*)

S. Wolfram, 1988. *Mathematica - A System for Doing Mathematics by Computer.* Addison-Wesley, 1 ed. (*pág. 4*)

Apéndices

Apéndice A

Módulos y archivos mencionados

A.1. En el capítulo 5

A.1.1. *holamundo* (ejercicio 5.8)

```
"""Imprime 'Hola Mundo'.

Es un módulo sencillo para ilustrar cómo se trabaja
con módulos propios (y también cómo se documentan).

"""
print('Hola Mundo')
```

A.1.2. *holapepe* (ejercicio 5.9)

```
"""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')
```

A.1.3. *sumardos* (ejercicio 5.11)

```
"""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)
```

A.1.4. *nada* (ejercicio 5.12)

```
"""No hace nada.

Sirve para establecer el directorio de trabajo cuando
ejecutado (no importado) desde la terminal de IDLE.

"""
pass
```

A.2. En el capítulo 6

A.2.1. *holas* (ejercicio 6.1)

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

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

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)
```

A.2.2. *globyloc* (ejercicio 6.6)

```
"""Ilustración de variables locales y globales.

Este módulo no debe importarse o ejecutarse, pues algunas
funciones tienen definiciones incorrectas y dará error.

La idea es copiar (y pegar en otro lado) los distintos ejemplos.

"""

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

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

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

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

def f(x):
    """Variable local a con problemas."""
    b = a   # b es local y a podría ser global
    a = 1   # a es local porque se asigna
            # y entonces estamos en problemas
            # porque la usamos antes de asignar
    return x + a + b

def f(x):
    """Trata de cambiar la variable global a."""
    global a
```

```
    a = 5           # a es... ¡global!
    return x + a

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

def f(x):
    """Trata de cambiar el argumento en la función."""
    x = 2       # x es local pues es el argumento
    print(' dentro de f el valor de x es', x)
    return x

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

A.2.3. *flocal* (ejercicio 6.7)

```
"""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)
```

A.2.4. *fargumento* (ejercicio 6.8)

```
"""Ejemplo de 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
```

A.3. En el capítulo 7

A.3.1. *grseno* (ejercicio 7.3)

```
"""Gráfico del seno entre 0 y pi."""

import math
```

```
import grpc

grpc.funcion(math.sin, 0, math.pi)

grpc.mostrar()
```

A.3.2. *grexplog* (ejercicio 7.5)

```
"""Gráficos de exp, log y la identidad."""

import math

import grpc

grpc.titulo = 'Comparación de la exponencial y el logaritmo'

xmin = -2
xmax = 5
ymin = -5
ymax = xmax
eps = 0.001 # log no está definida en 0

grpc.ymin = ymin
grpc.ymax= ymax

grpc.funcion(math.exp, xmin, xmax, leyenda='exp(x)')
grpc.funcion(math.log, eps, xmax, leyenda='log x')

# identidad para comparar
def identidad(x):
   return x
grpc.funcion(identidad, xmin, xmax, leyenda='identidad')

# ejes coordenados en negro
grpc.poligonal([(xmin, 0), (xmax, 0)], estilo={'fill': 'black'})
grpc.poligonal([(0, ymin), (0, ymax)], estilo={'fill': 'black'})

grpc.leyendaspos = 'NO'

grpc.mostrar()
```

A.3.3. *gr1sobrex* (ejercicio 7.7)

```
"""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 grpc

# titulo de la ventana
grpc.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

grpc.ymax = ymax
grpc.ymin = ymin

# mismo color para ambos tramos
grpc.funcion(f, a, -eps, estilo={'fill': 'blue'})
grpc.funcion(f, eps, b, estilo={'fill': 'blue'})

# eje x
grpc.poligonal([(a, 0), (b, 0)], estilo={'fill': 'black'})

# eje y
grpc.poligonal([(0, ymin), (0, ymax)], estilo={'fill': 'black'})

grpc.mostrar()
```

A.4. En el capítulo 9

A.4.1. *ifwhile* (capítulo 9)

```
"""Ejemplos sencillos de <<if>> y <<while>>."""

def espositivo(x):
    """Decidir si el número es positivo o no."""
    if x > 0:
        print(x, 'es positivo')
    else:
        print(x, 'no es positivo')

def esentero(x):
    """Decidir si un número es entero o decimal."""
    if (not isinstance(x, bool)) and (int(x) == x):
        print(x, 'es entero')
        return True
    else:
        print(x, 'no es entero')
        return False

def pisotecho(x):
    """Encontrar el piso y el techo de un número."""
    y = int(x)      # int redondea hacia cero
    if y < x:       # x debe ser positivo
        print('el piso de', x, 'es', y, 'y el techo es', y + 1)
    elif x < y:     # x debe ser negativo
        print('el techo de', x, 'es', y, 'y el piso es', y - 1)
    else:           # x es entero
        print('el piso y el techo de', x, 'es', y)

def resto(a, b):
    """Resto de la división entre los enteros positivos a y b.

    Usamos restas sucesivas.
```

```
    """
    r = a
    while r >= b:
        r = r - b
    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 n si es negativo
    c = 0   # contamos las cifras

    # lazo principal
    while True:         # repetir...
        c = c + 1
        n = n // 10
        if n == 0:      # ... hasta que n es 0
            return c    # salida
```

A.5. En el capítulo 10

A.5.1. *fibonacci* (ejercicios 10.19 y 10.20)

```
"""Construir números de Fibonacci."""

def listafib(n):
    """Lista con los primeros n números de Fibonacci."""

    lista = [1, 1]              # f1 y f2
    for k in range(2, n):       # para k = 2,...
        (a, b) = lista[-2:]     # tomamos los dos últimos
        lista.append(a + b)     # los sumamos y
                                # agregamos a la lista
    return lista

def fibonacci(n):
    """Calcular el n-ésimo número de Fibonacci."""

    a = 1
    b = 1
    for k in range(n-2):
        a, b = b, a + b
    return b
```

A.6. En el capítulo 11

A.6.1. *pascal* (ejercicio 11.6)

```
def pascal(n):
    """Filas del triángulo de Pascal."""

    fila = [1]   # n = 0
    filas = [fila[:]]
    for k in range(1, n+1):  # hacer n-1 veces
        a = 1       # guardar el valor anterior de la nueva fila
        for i in range(1, k):
            fila[i], a = a + fila[i], fila[i]
        fila.append(1)  # agregar 1 al final
        filas.append(fila[:])
```

```
    return filas
```

A.6.2. *tablaseno* (ejercicio 11.7)

```
"""Hacer una tabla del seno para ángulos entre 0 y 90 grados."""

import math

aradianes = math.pi/180

archivo = open('tablaseno.txt', 'w')    # abrirlo

for grados in range(0, 91):
    archivo.write('{0:3}   {1:<15.8g}\n'.format(
        grados, math.sin(grados*aradianes)))

archivo.close()     # y cerrarlo
```

A.6.3. *dearchivoaconsola* (ejercicio 11.8)

```
"""Leer un archivo de datos e imprimirlo en la consola."""

entrada = input('Ingresar el nombre del archivo a leer: ')

lectura = open(entrada, 'r')    # abrirlo

for renglon in lectura:
    print(renglon, end='')

lectura.close()     # y cerrarlo
```

A.6.4. *santosvega.txt* (ejercicio 11.8)

```
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.
```

A.7. En el capítulo 12

A.7.1. *dados* (ejercicios 12.4 y 12.5)

```
"""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):
    """Contar las veces que se tira un dado hasta que sale 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
```

A.8. En el capítulo 14

A.8.1. *enteros* (capítulo 14)

```
"""Funciones de teoría de números."""

def mcd1(a, b):
    """Máximo común divisor entre los enteros a y b.

    - Versión original usando diferencias.

    - a y b son positivos.

    """

    # lazo principal
    while a != b:   # mientras sean distintos
        # restar el menor del mayor
        if a > b:
            a = a - b
        else:
            b = b - a

    # acá a == b

    # salida
    return a

def mcd2(a, b):
    """Máximo común divisor entre los enteros a y b.

    Versión usando 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:
        a, b = b, a % b
    # acá b == 0

    # salida
    return a

def criba(n):
    """Criba de Eratóstenes: lista de primos  n."""

    # inicialización
    # observar que hacemos que todos los elementos de
    # la lista tengan un mismo valor inicial
```

```
    # usando una lista por comprensión
    # las posiciones 0 y 1 no se usan, pero conviene ponerlas
    esprimo = [True for i in range(0, n+1)]

    # lazo principal
    for i in range(2, n+1):
        if esprimo[i]:
            for j in range(2*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]]

def factoresprimos(a):
    """Encontrar factores primos de un entero positivo."""

    import math
    noesprimo = True
    b = 2
    factores = []
    while noesprimo:
        s = int(math.sqrt(a))
        while ((a % b) != 0) and (b <= s):
            b = b+1
        if (b > s):
            noesprimo = False   # o sea es primo
        else:   # a % b es 0 y b es factor
            factores.append(b)
            a = a // b  # a se modifica
    if a > 1:   # si las divisiones dejan algo
        factores.append(a)
    return factores

def esprimo(n):
    """Determinar si el entero positivo n es primo.

    Hacemos divisiones por los menores a n.

    """

    # podríamos poner directamente
    #    return len(factoresprimos(n)) == 1
    # pero puede ser ineficiente si hay factores pequeños

    if n < 2:
        return False

    import math

    s = int(math.sqrt(n))
    primos = criba(s)
    for p in primos:
        if n % p == 0:
            return False
    return True
```

A.8.2. *periodo* (ejercicio 14.17)

```
def periodo(p, q):
    """Encontrar los decimales de la fracción p/q (en base 10).
```

```
    - Se supone que p y q son enteros positivos.

    - Se destaca período y anteperíodo.

    """

    # Vamos encontrando los cocientes y restos sucesivos
    # terminando en cuanto un resto se repite.

    # inicialización
    restos = []
    cocientes = []
    rnuevo = p % q      # nos ponemos en el caso p < q

    # lazo principal
    while rnuevo not in restos:
        r = rnuevo
        c, rnuevo = divmod(r * 10, q)  # 10 r = c q + rnuevo
        cocientes.append(c)
        restos.append(r)

    # salida
    i = restos.index(rnuevo)
    print('Los decimales son:', cocientes)
    print('El anteperíodo es:', cocientes[:i])
    print('El período es:    ', cocientes[i:])
```

A.9. En el capítulo 15

A.9.1. *decimales* (capítulo 15)

```
"""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 representable.

"""

def epsilon(inic):
    """Menor potencia de 2 que sumada a inic supera a inic.

    - Suponemos inic > 0.
    - epsilon(1.0) es el épsilon de máquina para Python, epsmaq.

    """
    x = 1.0
    while inic + x > inic:
        x = x / 2
    return 2 * x    # volver para atrás

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
```

A.9.2. *euclides2* (ejercicio 15.7)

```
"""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 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
```

A.9.3. *numerico* (capítulo 15)

```
"""Varias funciones de cálculo numérico."""

def puntofijo(func, xinic):
    """Encontrar un punto fijo de func dando un 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)

    # salida
    print('Resultados:')
    print(' inicial:   ', xinic)
    print(' f(inicial):', yinic)
    print(' final:     ', x)
    print(' f(final):  ', y)
    print(' Cantidad de iteraciones:', it + 1)
    if it + 1 == maxit:
        print(' *** Iteraciones máximas alcanzadas')
        print('     el resultado puede no ser solución')

def babilonico(a):
    """Raíz cuadrada aproximada del número real positivo a.

    Usa el método «babilónico».

    """

    maxit = 10      # máximo número de iteraciones
    tol = 10**(-7)  # error permitido
    x0 = 1.0        # punto inicial
```

```
    # inicialización
    it = 1
    x = x0
    y = (x + a/x)/2

    # lazo principal
    while ((it <= maxit) and (abs(x - y) > tol)):
        x = y
        y = (x + a/x)/2
        it = it + 1

    # salida
    print('Iteraciones realizadas:', it - 1)
    if it > maxit:
        print('*** Iteraciones máximas alcanzadas')
    print('Solución aproximada obtenida:', 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.

    """

    # 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

    print('Iteraciones realizadas:', it + 1)
    if it + 1 ==  maxit:
        print('*** Iteraciones máximas alcanzadas')
    print('Solución aproximada obtenida (x):', x)
    print('        Valor de la función en x:', y)

def biseccion(func, poco, mucho):
    """Método de la bisección para encontrar ceros de una función.

    '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
    xpoco = poco
    xmucho = mucho
    ypoco = func(xpoco)
    ymucho = func(xmucho)

    sol = []    # guardamos solución, por ahora ninguna

    # casos particulares
    if ypoco * ymucho > 0:
        print('*** La función debe tener', end=' ')
        print('signos opuestos en los extremos')
        return []
    if abs(ypoco) < eps:
        sol = [xpoco, ypoco]
    elif abs(ymucho) < eps:
        sol = [xmucho, ymucho]

    # a partir de acá debe haber solución si la función es continua
    # lazo principal
    while len(sol) == 0:    # mientras no haya solución
        x = (xpoco + xmucho) / 2
        y = func(x)
        if abs(y) < eps:
            sol = [x, y]
        elif y * ypoco < 0:
            xmucho = x
        else:
            xpoco, ypoco = x, y

    # salida
    print('Resultados:')
    print('  Puntos iniciales')
    print('    poco:    ', poco)
    print('    f(poco): ', func(poco))
    print('    mucho:   ', mucho)
    print('    f(mucho):', func(mucho))
    print('  raíz:   ', sol[0])
    print('  f(raíz):', sol[1])
```

A.9.4. *grpuntofijo* (ejercicio 15.13)

```
"""Graficar la técnica de punto fijo iterando una función.

Plantilla para usar el método de punto fijo
con gráficos interactivos: cambiar apropiadamente.

Se imprime genéricamente 'f' en los resultados por terminal.

"""

import math     # si se usan funciones trascendentes
```

```
import grpc

# 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
grpc.titulo = 'Ilustración del método de punto fijo'
grpc.leyendaspos = 'S'

# ejecución
grpc.puntofijo(f, a, b, eps, maxit, leyendaf)
```

A.9.5. *grnewton* (ejercicio 15.19)

```
"""Graficar el método de Newton para encontrar ceros de una función.

Se debe elegir (interactivamente) un punto inicial.

La derivada se estima poniendo un incremento bien pequeño.

Plantilla para usar el método de Newton
con gráficos interactivos: cambiar apropiadamente.

Se imprime genéricamente 'f' en los resultados por terminal.

"""

# import math # si se usan funciones trascendentales

import grpc

# 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
grpc.titulo = 'Ilustración del método de Newton'
grpc.leyendaspos = 'SE'
```

```
# limitar si no se ven bien positivos/negativos
grpc.ymin = -20
grpc.ymax = 10

# ejecución
grpc.newton(f, a, b, eps, maxit, leyendaf)
```

A.9.6. *grbiseccion* (ejercicio 15.22)

```
"""Graficar el método de la bisección para ceros de una función.

Se deben elegir (interactivamente) los extremos de un intervalo
que encierra al menos un cero de la función.

Plantilla para usar el método de la bisección
con gráficos interactivos: cambiar apropiadamente.

Se imprime genéricamente 'f' en los resultados por terminal.

"""

# import math # si se usan funciones trascendentales

import grpc

# 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
grpc.titulo = 'Ilustración del método de la bisección'
grpc.leyendaspos = 'N'
# limitar si no se ven bien positivos/negativos
grpc.ymin = -3
grpc.ymax = 5

grpc.biseccion(f, a, b, eps, leyendaf)
```

A.10. En el capítulo 16

A.10.1. *grafos* (capítulo 16)

```
"""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(ngrafo, vecinos):
    """Pasar de lista de vecinos a lista de aristas.

    Los índices para los vértices empiezan desde 1.

    """
    aristas = []
    for v in range(1, ngrafo + 1):
        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,
    retornando 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]

def dijkstra(ngrafo, aristas, s, t):
    """Algoritmo de Dijkstra para camino más corto.

    - Los datos son:

        * la cantidad de vértices,

        * lista de aristas:
            cada arista en la forma [u, v, w], donde w es el costo,

        * s, t: puntos a conectar.

    - Los costos deben ser positivos.

    """

    # función a usar como llave (key) al encontar mínimo
    def fdist(x):
        return dist[x]
```

```
    # constantes
    INFINITO = float('infinity')
    vertices = range(ngrafo + 1)    # incluimos v = 0

    # pasar a vecinos con pesos
    adys = [[] for u in vertices]
    adys[0] = None      # previene errores
    for a in aristas:
        u, v, w = a
        adys[u].append([v, w])
        adys[v].append([u, w])

    dist = [INFINITO for v in vertices]
    dist[s] = 0
    padre = [None for v in vertices]
    padre[s] = s
    enarbol = [False for v in vertices]     # si v está en V(T)
    cola = [s]
    while len(cola) > 0:            # mientras la cola es no vacía
        u = min(cola, key=fdist)    # elegir el que sale
        cola.remove(u)              # sacarlo de la cola
        enarbol[u] = True           # y ponerlo en el árbol
        if u == t:  # ya está
            break
        # actualizar cola y distancias
        for v, w in adys[u]:        # examinar vecinos
            if dist[u] + w < dist[v]:
                if dist[v] == INFINITO:
                    cola.append(v)
                dist[v] = dist[u] + w
                padre[v] = u

    # salida
    if u != t:  # sonamos
        print('El grafo no es conexo.')
        return False
    return dist[t]  # distancia de s a t

def prim(ngrafo, aristas, raiz):
    """Algoritmo de Prim para mínimo árbol generador.

    - Los datos son:

        * la cantidad de vértices,

        * lista de aristas:
            cada arista en la forma [u, v, w], donde w es el costo,

        * raíz donde empezar el algoritmo

    - Los costos deben ser positivos.

    """

    # función a usar como llave (key) al encontar mínimo
    def fdist(x):
        return dist[x]

    # constantes
    INFINITO = float('infinity')
    vertices = range(ngrafo + 1)    # incluimos v = 0
```

```
    # pasar a vecinos con pesos
    adys = [[] for u in vertices]
    adys[0] = None      # previene errores
    for a in aristas:
        u, v, w = a
        adys[u].append([v, w])
        adys[v].append([u, w])

    dist = [INFINITO for v in vertices]
    dist[raiz] = 0
    padre = [None for v in vertices]
    padre[raiz] = raiz
    enarbol = [False for v in vertices]     # si v está en V(T)
    cola = [raiz]
    while len(cola) > 0:            # mientras la cola es no vacía
        u = min(cola, key=fdist)    # elegir el que sale
        cola.remove(u)              # sacarlo de la cola
        enarbol[u] = True           # y ponerlo en el árbol
        # actualizar cola y distancias
        for v, w in adys[u]:        # examinar vecinos
            if (not enarbol[v]) and (w < dist[v]):
                if dist[v] == INFINITO:
                    cola.append(v)
                dist[v] = w
                padre[v] = u

    # salida
    vdet = [v for v in vertices if enarbol[v]]      # V(T)
    # ver si es generador
    if len(vdet) < ngrafo:
        print('El grafo no es conexo.')
        return False
    # costo de árbol generador mínimo
    return sum([dist[v] for v in vdet])
```

A.10.2. *grgrsimple* (ejercicio 16.8)

```
"""Ilustración del uso del módulo grgr para un grafo simple.

Recordar que se pueden mover los vértices y etiquetas del grafo
con el ratón.

"""

import grgr

#-----------------------------------------------
# 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]]

#-----------------------------------------------
# usando grgr
```

```
grgr.titulo = 'Ejemplo de grafo en los apuntes'

for i in range(1, ngrafo + 1):
    grgr.vertice(texto=str(i))
for u, v in aristas:
    grgr.arista(grgr.vertices[u], grgr.vertices[v])

grgr.mostrar()
```

A.10.3. *grgrpesado* (ejercicio 16.17)

```
"""Ilustración del uso del módulo grgr para un grafo pesado.

Recordar que se pueden mover los vértices y etiquetas del grafo
con el ratón.

"""

import grgr

#-----------------------------------------------
# descripción del grafo, modificar a gusto
# ejemplo en los apuntes
ngrafo = 6
aristas = [[1, 2, 2],
           [1, 4, 3],
           [1, 5, 8],
           [2, 3, 2],
           [2, 4, 1],
           [3, 4, 1],
           [3, 5, 2],
           [3, 6, 1],
           [4, 6, 1],
           [5, 6, 3]]

#-----------------------------------------------
# usando grgr

grgr.titulo = 'Ejemplo de grafo pesado en los apuntes'

# leyendas
grgr.leyenda(textob='w', texto='peso de arista')

for i in range(1, ngrafo + 1):
    grgr.vertice(texto=str(i))

for u, v, w in aristas:
    grgr.arista(grgr.vertices[u], grgr.vertices[v], texto=str(w))

grgr.mostrar()
```

A.11. En el capítulo 17

A.11.1. *recursion* (capítulo 17)

```
"""Ejemplos 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

def hanoi(n):
    """Solución recursiva a las torres de Hanoi."""

    def pasar(n, x, y, z):
        """Pasar los discos 1...n de x a y usando z."""

        if n > 1:
            pasar(n - 1, x, z, y)
            print('pasar el disco', n, 'de', x, 'a', y)
            pasar(n - 1, z, y, x)
        else:   # sólo un disco: el 1
            print('pasar el disco 1 de', x, 'a', y)

    pasar(n, 'a', 'b', 'c')

def cadenasdebits(n, haceralgo):
    """Hacer algo con cada una de las cadenas de n bits.

    Usamos una pila generando las cadenas de a una.

    """

    def cadena(k):
        """Agregar un 0 o un 1 a la lista a."""

        for i in range(2):
            a.append(i) # poner 0 o 1 en el lugar k
            if k < n:
                cadena(k+1) # seguir agregando
            else:    # hacer algo con la cadena
                haceralgo(a)
            a.pop()     # sacar lo agregado

    a = []      # inicialización
```

```
    cadena(1)   # ejecucion

def subconjuntos(n, haceralgo):
    """Hacer algo con cada subconjunto de {1,...,n}.

    Usamos una pila, generando los subconjuntos de a uno.

    """

    def ponerono(m):
        """Agregar o no m al conjunto."""
        if m == n + 1:      # me pasé, usar lo que hay
            haceralgo(pila) # hacer algo
        else:
            # acá es m <= n
            pila.append(m)   # agregar m al conjunto
            ponerono(m + 1)  # probar con m + 1
            pila.pop()       # y sacarlo
            ponerono(m + 1)  # probar con m + 1

    pila = []
    ponerono(1)

def subconjuntosnk(n, k, haceralgo):
    """Hacer algo con cada subconjunto de k elementos de {1,...,n}.

    Usamos una pila, generando los subconjuntos de a uno.

    """

    def ponerono(m):
        """Agregar o no m al conjunto."""

        if len(pila) == k:  # no se pueden agregar más
            haceralgo(pila) # hacer algo
        elif m <= n:
            pila.append(m)  # agregar m al conjunto
            ponerono(m + 1) # probar con m + 1
            pila.pop()      # y sacarlo
            ponerono(m + 1) # probar con m + 1

    if (k < 0) or (k > n):  # nada que hacer
        return None

    pila = []
    ponerono(1)

def permutaciones(n, haceralgo):
    """Hacer algo con cada permutación de 1,..., n.

    Usamos una pila, generando las permutaciones de a uno.

    """

    def poner(m):
        """poner m en las posiciones 0,...,m-1."""

        if m == n+1:        # me pasé
            haceralgo(pila) # hacer algo
        else:
            for k in range(m):      # para k = 0,..., m-1
```

```
                pila.insert(k, m)   # poner m en el lugar k
                poner(m + 1)        # poner siguiente
                pila.pop(k)         # volver al original

    pila = []
    poner(1)
```

A.11.2. *nolocal* (ejercicio 17.4)

```
def llamar(n):
    """Llamar n veces a una función interna."""

    def aumenta():  # función interna
        nonlocal c
        c = c + 1   # que incrementa contador

    c = 0               # inicializar contador
    for i in range(n):  # hacer n llamadas
        aumenta()

    # acá debería ser c == n
    print('La función interna se llamó', c, 'veces')
```

Apéndice B

Algunas notaciones y símbolos

Ponemos aquí algunas notaciones, abreviaciones y expresiones usadas (que pueden diferir de algunas ya conocidas), sólo como referencia: deberías mirarlo rápidamente y volver cuando surja alguna duda.

B.1. Lógica

$\Rightarrow$ *implica* o *entonces*. $x > 0 \Rightarrow x = \sqrt{x^2}$ puede leerse como *si x es positivo, entonces...*

$\Leftrightarrow$ *si y sólo si*. Significa que las condiciones a ambos lados son equivalentes. Por ejemplo $x \geq 0 \Leftrightarrow |x| = x$ se lee *x es positivo si y sólo si...*

$\exists$ *existe*. $\exists\, k \in \mathbb{Z}$ tal que... se lee *existe k entero tal que...*

$\forall$ *para todo*. $\forall\, x > 0$, $x = \sqrt{x^2}$ se lee *para todo x positivo,...*

$\neg$ La negación lógica *no*. Si p es una proposición lógica, $\neg p$ se lee *no p*. $\neg p$ es verdadera $\Leftrightarrow$ p es falsa.

$\wedge$ La conjunción lógica *y*. Si p y q son proposiciones lógicas, $p \wedge q$ es verdadera $\Leftrightarrow$ tanto p como q son verdaderas.

$\vee$ La disyunción lógica *o*. Si p y q son proposiciones lógicas, $p \vee q$ es verdadera $\Leftrightarrow$ o bien p es verdadera o bien q es verdadera.

B.2. Conjuntos

$\in$ *pertenece*. $x \in A$ significa que x *es* un elemento de A.

$\notin$ *no pertenece*. $x \notin A$ significa que x *no es* un elemento de A.

$\cup$ *unión* de conjuntos. $A \cup B = \{x : x \in A \text{ o } x \in B\}$.

$\cap$ *intersección* de conjuntos. $A \cap B = \{x : x \in A \text{ y } x \in B\}$.

$\setminus$ *diferencia* de conjuntos. $A \setminus B = \{x : x \in A \text{ y } x \notin B\}$.

$|A|$ *cardinal del conjunto A*. Es la cantidad de elementos de A. No confundir con $|x|$, el *valor absoluto del número x*.

$\emptyset$ El *conjunto vacío*, $|\emptyset| = 0$.

B.3. Números: conjuntos, relaciones, funciones

$\mathbb{N}$ El conjunto de números naturales, $\mathbb{N} = \{1, 2, 3, \dots\}$. Para nosotros $0 \notin \mathbb{N}$.

$\mathbb{Z}$ Los enteros, $\mathbb{Z} = \{0, 1, -1, 2, -2, \dots\}$.

$\mathbb{Q}$ Los racionales p/q, donde $p, q \in \mathbb{Z}$, $q \neq 0$.

$\mathbb{R}$ Los reales. Son todos los racionales más números como $\sqrt{2}$, 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 $n = k\,m$.
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 $z \in \mathbb{Z}$.

$[x]$ La *parte entera* de x, $x \in \mathbb{R}$, $[x] = \lfloor x \rfloor$. 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*, $\tan x = \operatorname{sen} x / \cos x$.

$\operatorname{arcsen} x$,
$\arccos x$,
$\arctan x$ Funciones trigonométricas inversas respectivamente de sen, cos y tan.
En Python, se indican como `asin`, `acos` y `atan` (resp.).

$\operatorname{signo}(x)$ Las función *signo*, definida para $x \in \mathbb{R}$ por

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

✍ 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.

✍ Señala notas de índole más técnica. Están en letras más pequeñas y en general se pueden omitir en una primera lectura.

❦ Señala comentarios históricos, curiosidades, etc. Están en letras cursivas más pequeñas, y se pueden omitir en la lectura.

⚯ Explicación, conclusión o «moraleja» especialmente interesante que se puede obtener de lo que se acaba de ver. El texto está en cursiva para distinguirlo.

💣 Cosas que hay que evitar. *En general quitan puntos en los exámenes.*

 El pasaje señalado con este símbolo tiene conceptos que pueden llevar a confusión y debe leerse cuidadosamente.

> ❦ *El símbolo de «curva peligrosa» fue ideado por Bourbaki y popularizado por Knuth en sus libros de TEX.*

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.

Resumen de comandos

Índice de figuras y cuadros

Índice de autores

Índice alfabético