Ciencias e Ingeniería para estudiantes de Python

Miguel Ángel Suárez

ISBN: 978-980-12-9190-9
Primera Impresión, Septiembre 2014
Este libro se editó en LaTeXmodificando una plantilla originalmente propuesta por Mathias Legrand (legrand.mathias@gmail.com). License: CC BY-NC-SA 3.0 (http://creativecommons.org/licenses/by-nc-sa), la cual se puede descargar libremente de http://www.LaTeXTemplates.com.

Índice general

Prefacio

En la actualidad la computadora se ha vuelto una herramienta importante e imprescindible en el ámbito de las ciencias y en particular en la física. No es posible hacer física sin recurrir a la computadora en algún momento, sin embargo la introducción de las herramientas computacionales en los pensa de estudios ha sido lenta y tímida sin un criterio unificado. Que se debe enseñar, como se debe enseñar y a que nivel, es todavía materia de controversia. Lo ideal sería que se llegara a un consenso parecido al que existe con las matemáticas, poca discusión existe al respecto, ya que está mas o menos establecido que parte del inmenso reino de las matemáticas es relevante para cada área de la ciencia o ingeniería, lamentablemente esto está muy lejos de ser cierto en la *computación para ciencias*.

Generalmente el primer contacto que tiene el estudiante con una herramienta computacional es con una hoja de cálculo, ya sea Excel, LibreOfice, OpenOfice, etc. Las hojas de cálculo son amigables, pero están más orientadas hacia el mundo financiero y pronto se notan sus limitaciones a pesar de su aceptable capacidad para las gráficas. Pero si de graficación se trata, no hay nada como Gnuplot o Xmgrace. La gran mayoría de las gráficas que aparecen las revistas científicas especializadas en física están hechas con Gnuplot, además con ambos paquetes se pueden realizar análisis de datos básicos, como regresiones lineales por ejemplo. Claro que si lo que se desea es realizar un análisis de datos mas serio, se puede recurrir a Sigma Plot, QtPlot, SciDAVIS, entre otros, todos ellos muy utilizados entre los físicos experimentales. Pero si hablamos del software técnico preferido por los experimentalistas, sin duda el que se lleva las palmas es MatLab seguido de cerca por sus competidores Scilab y Octave. Sin embargo para los científicos más inclinados al análisis estadístico de los datos tenemos SPSS, S-Plus, Statistical, o R. Y por supuesto no podemos dejar de lado las herramientas de computación simbólica preferidas por los científicos teóricos como Maple, Matematica, MuPAD (que es utilizada por Matlab), Maxima, etc. Finalmente también existen los viejos pero robustos caballitos de batalla que son los lenguajes de programación. El número de lenguajes existentes es asombroso, pero en el ámbito científico e ingenieril descollan pocos. El más veterano de todos es Fortran con un bagaje muy extenso de programas y librerías

de alta calidad, pero en los últimos años el C/C++ ha estado peleando duramente por la supremacía y por supuesto el nuevo en el campo de batalla, Python.

Por lo tanto, ¿Cuál paquete de software debe aprender el estudiante de ciencias o ingeniería? ¿Qué debe enseñarse? ¿Cálculo numérico? ¿Física computacional? ¿Herramientas para el procesamiento da datos? ¿Álgebra simbólica?

Este pequeño tutorial de Python pretende responder en parte a estas preguntas al menos en lo referente a la temática y al software que deben utilizarse a un nivel introductorio en una facultad de ciencias o ingeniería.

En lo concerniente al software, pienso que en primer lugar la universidad debe de mantenerse alejada de las patentes y del software privativo lo mas que se pueda, ya que pagar licencias es muy oneroso para el escaso presupuesto que se maneja. En la práctica las computadoras de los laboratorios, las de las oficinas de los profesores y las de las secretarias, están repletas de software "pirateado", creo firmemente que esto es inaceptable, sobretodo cuando existen a la disposición alternativas de software libre de altísima calidad. Esta consideración reduce enormemente el abanico de posibilidades en lo que a escogencia de software se refiere.

Otra característica importante es que el software utilizado en la enseñanza debe ser amigable y de fácil aprendizaje. Si lograr el dominio de la herramienta computacional es un reto en si mismo deja de ser una herramienta práctica, al menos a un nivel introductorio. Esto saca de las opciones a lenguajes como C/C++ o Java.

El lenguaje de programación Python cumple con estas características, pero adicionalmente es un lenguaje poderoso y altamente flexible. Con el se pueden hacer pequeños guiones (script) para automatizar tareas, manipular datos, etc., permite emular a paquetes como Matlab, con una capacidad gráfica notable. Adicionalmente con la importación de las librerías adecuadas permite realizar cálculo numérico avanzado, controlar hardware e incluso aplicar álgebra simbólica.

La presente guía esta orientada a los estudiantes de los primeros semestres de la licenciatura de las carreras de ciencias o ingeniería, aunque alumnos de semestres superiores que quieran aprender Python también podrán sacarle provecho. Se ha tratado en todo momento que sea de ágil lectura, con ejemplos relevantes en el área de la física o matemáticas.

Miguel Ángel Suárez
Septiembre, 2014

1 — Introducción

Antes de meternos de lleno a programar en Python es necesario aclarar algunas pocas cosas. En general todo software posee lo que se llama licencias. No es el objetivo de este libro adentrarse en el interesante mundo de las licencias y patentes [FSF; GNU] y de como estas influyen en el mundo de la tecnología, pero de una forma vaga e imprecisa, pudiéramos decir que las licencias de software son un instrumento legal que se relaciona con el modo en el que el software se distribuye y la forma en que el usuario puede utilizarlo. Python utiliza la licencia "Open Source"[OP] que es compatible con la "Licencia Pública General"(o GPL por sus siglas en inglés)[GPL]. A mi entender, según como se vea el software puede trasmitirse como conocimiento, o como un objeto. Los objetos suelen tener un precio asociado a él y algunas veces (como es el caso de ciertos software) son muy caros. El conocimiento por otra parte es invaluable. La licencia de Python garantiza que se transmita como conocimiento y por lo tanto no se puede pagar. Lo que si podemos hacer es ir a su página web [1] y descargárnoslo gratuitamente. Por supuesto una vez descargado debemos proceder a instalar Python en nuestra computadora, este no es un procedimiento difícil, y en Internet se encuentra abundante información al respecto. Lo único que hay que tomar en cuenta es el sistema operativo que se esté utilizando.

1.1 ¿Qué versión de la historia prefieres?

Antes de descargar Python e instalarlo, se tiene que tomar una decisión acerca de cuál versión elegir. Por supuesto en general la última es la más adecuada, pero con Python tenemos un pequeño problema, no todas las versiones son compatibles. Actualmente estamos en la rama del Python 3 cuya versión a la hora de escribir este libro es la Python 3.3.3. Esta rama no es compatible con la 2 que tuvo un éxito considerable. Por lo tanto existen en la actualidad una gran cantidad de programas escritos para Python 2 cuya última versión fue la Python 2.7.5. Afortunadamente podemos tener mas de una versión instalada en el mismo dispositivo. Por lo que las diferentes versiones no

[1] http://www.python.org

son un problema a la hora de ejecutar el código, pero para programar hay que tener claro cual versión se está utilizando. Esta incompoatibilidad entre las versiones de Python es un inconveniente, pero es necesario si se quiere tener un lenguaje ágil y moderno. Efectivamente la computación es una ciencia en evolución constante, nuevos paradigmas surgen, las técnicas se mejoran, los estilos cambian, etc. Adicionalmente existen actualmente programas creados para "traducir" de la rama 2 a la 3, además de extensa información al respecto[Pyt]. Este libro no requiere ningún conocimiento previo de Python, y como se utilizará en todo momento la versión 3.3, no hará falta ninguna clase de referencia a versiones más antiguas.

1.2 Interpretando Python

Antes de seguir debemos aclarar un pequeño detalle técnico, una cosa el lenguaje de programación llamado Python y otra cosa es el interprete de Python. El interprete es un programa que ejecuta otros programas. Lenguajes como Fortran o C/C++ necesitan ser compilados, en ellos un programa llamado compilador, traduce a lenguaje de máquina el código creado por el programador y produce un ejecutable. En Python por el contrario el interprete lee la primera línea el código escrito por el programador, lo traduce a lenguaje de máquina y la ejecuta, luego lee la siguiente y así sucesivamente. Esto hace que la ejecución sea mas lenta que en lenguajes como C pero se gana en flexibilidad y permite trabajar en forma interactiva.

1.3 Lo primero es aprender a ser iteractivo

Se podría decir que hay dos formas de utilizar Python una de ellas es genial, la otra aún lo es más. La primera es la manera iteractiva, la otra creando un programa (con la ayuda de un editor de texto) que después tiene que ser ejecutado. Comenzaremos explicando la forma iterativa.

Uno de los talentosos actores de Monty Python's Flying Circus se llamaba Eric Idle pero IDLE también significa Integrated DeveLopment Environment. De IDLE hablaremos un poco mas adelante, por ahora solo diremos que es una espectacular y poderosísima herramienta diseñada originalmente por el propio Guido van Rossum para escribir programas en Python y que en este libro no se utilizará en lo absoluto. De hecho no utilizaremos de momento ningún editor de texto, por el contrario empezaremos simplemente invocando a una cónsola, consola, terminal, o interprete de órdenes, como usted desee. Esto es muy fácil de hacer en cualquier variante del sistema operativo Linux, simplemente a través del menú principal. Por supuesto que en otros sistemas operativos también es fácil de hacer, pero usted será el encargado de averiguar como.

En la forma iterativa se utiliza el hecho de que Python es un lenguaje interpretado. En un terminal se escribe una instrucción, esta es ejecutada, luego se escribe la otra, es ejecutada, y así sucesivamente. La ventaja de este método es que tenemos una respuesta instantánea de lo que estamos haciendo. De esta forma se trabaja en MatLab, un software muy utilizado por ingenieros, técnicos y científicos y que se puede emular totalmente con Python. Por lo tanto, nuestro primer paso será abrir un terminal. Se dejará para mas adelante la segunda forma de utilizar Python haciendo uso de editores de texto, tanto genéricos como los dedicados a Python como lo es el propio IDLE.

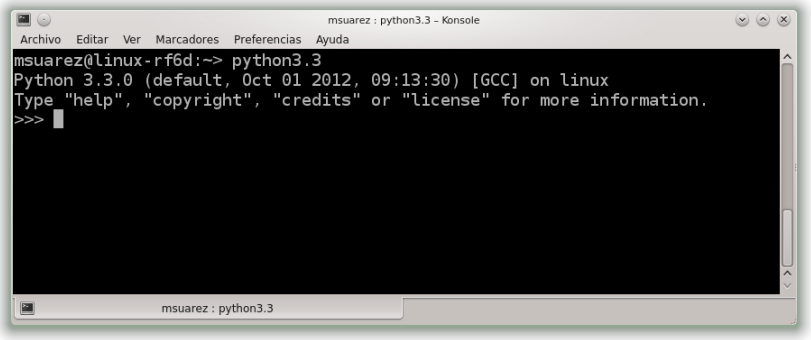


Figura 1.1: *Una típica ventana con un terminal en linux. En la linea de comandos se ha invocado a Python versión 3.3*

Una vez creada la ventana del terminal se observa en ella una cosa llamada prompt y que significa que el terminal está listo para recibir ordenes. En el caso que se muestra en la figura 1.1 es:

```
msuarez@linux-rf6d:~>
```

mientras que en windows el prompt es el (antiguamente) famoso "C:\ >". La orden que hemos dado es ejecutar python, y para ello se ha escrito simplemente python3.3 (donde el número 3.3 indica la versión que se está utilizando). Luego de apretar la tecla *intro* python responde con tres líneas. En la primera se confirma la versión de Python, la fecha de lanzamiento y el compilador que se ha utilizado para compilar al propio Python, en este caso se está usando linux y el compilador es GCC.

La segunda línea nos sugiere que escribamos una serie de comandos que se explican por si mismos, y me ahorran el trabajo de hacerlo a mi. En particular el comando "help()" puede ser de mucha utilidad mas adelante, cuando no se entienda nada de lo que estoy explicando. Finalmente la última línea ">>>" es una invitación a escribir algo. ¡Pero no lo haga! ¡No sea atrevido! ¡No sea curioso! ¡No haga nada por usted mismo! Espere a que yo le explique todo, y si no me entiende, pues entonces si escriba help().

Cuando haya tenido suficiente y quiera olvidarse de todo, entonces puede escribir quit() o exit() o apretar simultáneamente las teclas *Ctrl* y *d*.

2 — Primeros Tropiezos

Es inevitable cuando se comienza, cuando se continua, y después de haber terminado un programa, darse cuenta que se han cometido algunas decenas, cientos o tal vez miles de errores. Esto es totalmente natural, lo importante es corregirlos. Como Python es un lenguaje compilado, los errores van apareciendo a medida que se ejecuta el programa, lo cual es bastante útil, sobretodo en una sesión interactiva porque nos permite corregirlos a medida que se cometen y con eso evitamos tener que desenredar laboriosamente una madeja de errores anidados. En la medida que el aburrimiento nos lo permita se resaltarán los errores mas frecuentes y se explicará la solución, si es que el autor la conoce.

2.1 Utilice su nueva calculadora

Bueno, comencemos a andar para poder así dar nuestros primeros tropiezos, como ya se explicó anteriormente abriremos una sesión iterativa en una consola o interprete de órdenes, e invocaremos a Python escribiendo simplemente pyton3.3 (o python3, o python2.7 o la versión que usted tenga). Lo primero será hacer una suma, como estamos empezando no vamos a forzar mucho nuestro intelecto, así que sumaremos dos mas dos, para eso tenemos que escribir en la cónsola o terminal simplemente “2+2”. En el terminal se tiene que ver:

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

¿Todo bien? Ahora apretaremos la tecla “enter” (o “intro” según su teclado y preferencia) y el extraordinario y poderoso compilador de Python nos da como por arte de magia la respuesta.

```
>>>2+2
4
>>>
```

Fantástico! ¿Verdad que valió la pena? Es extraordinario lo que se puede hacer con la tecnología hoy en día! De hecho Python le ahorra una cantidad enorme de dinero evitándole comprar una calculadora ya que no solo sabe sumar 2+2, vea por ejemplo el siguiente ejemplo.

```
>>> ((2*(3528)/24)+8931)-9221
4.0
>>>
```

El cual a pesar de las apariencias, *no* da el mismo resultado que el del ejemplo anterior. La diferencia radica en que 4 es un número entero mientras que 4.0 no lo es[1], tampoco es un número real, no, tampoco es imaginario. Es un número *punto flotante*. Más adelante volveremos sobre este punto (flotante), por ahora fijémonos en el siguiente ejemplo:

```
>>> 2*3528/24+8931-9221
4.0
>>>
```

Si se parece mucho al ejemplo anterior es porque efectivamente es lo mismo, bueno casi. La diferencia obviamente está en los paréntesis. Para hacer una operación como esta: $5+25/2$ el interprete debe saber de antemano cuales operaciones realizar primero y cuales después. No es lo mismo realizar la división de 25 entre 2 y luego sumar 5, que sumar primero 5 + 25 y luego dividir entre 2. es decir:

```
>>> 5+25/2
17.5
>>>
```

en cambio

```
>>> (5+25)/2
15.0
>>>
```

Por eso en este libro se utilizará constantemente y redundantemente el uso de paréntesis. De esta manera estamos seguros que el interprete de Python hará las operaciones de la forma en que nosotros queremos y no a su manera.

Además de la suma, resta, multiplicación, y división, Python reconoce tres operaciones más. Con "**" se realiza la potenciación

```
>>> 5**2
25
>>> 2**5
32
```

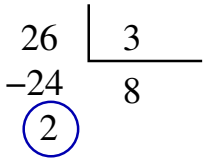
[1] A pesar de que este libro está escrito en castellano, para representar los números se utilizará la notación inglesa, es decir 4.0 y no 4,0 como debería de ser. Esto es para tener compatibilidad con el interprete de Python

```
>>> 16**0.5
4.0
>>> 16**1/2
8.0
>>> 2**3**2
512
>>> (2**3)**2
64
>>>
```

notemos que en la potenciación el resultado sólo es entero si tanto la base como el exponente son enteros y el resultado correcto también lo es. Para la división entera tenemos //

```
>>> 14//2
7
>>> 15//2
7
>>>
```

y finalmente % que no se usa para sacar porcentaje, se usa para hallar el resto de la división entera. ¿Se acuerdan de la escuela?



bueno entonces

```
>>> 26%3
2
>>>
```

este tipo de operación es muy útil y ya tendremos oportunidad de utilizarla más adelante. En definitiva el *orden de precedencia* en que se realizan las operaciones (si no se utilizan paréntesis) viene dado por la siguiente lista de prioridad siendo el primer operador el de mayor prioridad y el último el de menor prioridad:

- ** (exponenciación)
- *, /, //, % (multiplicación, división, división entera y resto)
- +, - (suma y resta)

cuando las operaciones tienen el mismo orden de precedencia se ejecutan de izquierda a derecha.

2.2 1 + 1 es igual a 10

Tal vez usted haya escuchado alguna vez decir que las computadoras solo entienden una clase especial de números llamados *binarios*. Pues es verdad, no le han dicho ninguna mentira. Los números binarios están formados por dos tipos de dígitos o símbolos “0” y “1”. Es fácil formar cualquier número entero con solo dos símbolos, por ejemplo:

```
0000 ---> 0
0001 ---> 1
0010 ---> 2
0011 ---> 3
0100 ---> 4
0101 ---> 5
0110 ---> 6
0111 ---> 7
1000 ---> 8
1001 ---> 9
```

... y así sucesivamente. ¿Por que esta sucesión en particular? se preguntarán algunos. Los que no se lo pregunten, pueden saltarse la siguiente tediosa explicación:

Para los números que usamos corrientemente en la actualidad, tenemos sólo diez símbolos distintos (del 0 al 9), si queremos representar un número mayor que 9 tendremos que repetir algunos símbolos. La solución es darle diferente valor a los símbolos dependiendo del lugar que ocupen. Por lo tanto para escribir $9+1$ colocamos (repetimos el dígito) 0 en el lugar de las unidades y colocamos (repetimos) un 1 en el lugar de las decenas, es decir: 10. De esta manera podemos crear nuestra lista de números "decimales" o en base 10 de la siguiente manera.

```
0000
0001
0002
0003
   .
   .
   .
0008
0009
0010
0011
0012
0013
   .
   .
   .
0019
0020
   .
   .
   .
0099
0100
0101
   .
```

.
.

Con los números binarios procedemos de igual forma, pero claro solo tenemos dos símbolos diferentes. Si a 0 le sumamos 1 no hay problema $0+1=1$, pero para lograr $1+1$ tenemos que repetir símbolos y haciendo uso de la posición, colocamos cero y nos "llevamos" 1, entonces $01+01=10$. Claro los humanos no suelen poner los ceros a la izquierda, pero las computadoras no son tan inteligentes, ellas guardan los números en un espacio en la memoria y cada bit cuenta.

Aún quedan por ahí maquinas con 32 bits (además de los teléfonos celulares y tablets) eso significa que en esas computadoras disponemos de una combinación de 32 "ceros" y "unos" para representar un número cualquiera. A esta combinación se le llama *palabra* (que muy frecuentemente se usa sin traducir, "word"). Por ejemplo el cinco es:

00000000000000000000000000000101

Adicionalmente debemos distinguir entre números positivos y negativos, así que el primer dígito lo reservamos al signo, si es 0 es positivo y si es 1 es negativo[2]. En el ejemplo anterior el número sería entonces el cinco positivo (+5). Ahora es fácil saber cuál es el entero positivo más grande que se puede representar con una palabra en una máquina de 32 bits, este es.

01111111111111111111111111111111

Python 3 viene con una interesante y nueva[3] característica para los números enteros, si queremos representar un número aún mayor al anterior, no hay problema, el interprete de Python utilizará dos palabras, luego tres, cuatro, y de esta manera se puede llegar hasta números enteros verdaderamente astronómicos. Claro que de ninguna manera la computadora puede manejar números arbitrariamente grandes, al final existe un límite dado por la memoria RAM de la máquina.

2.3 Manteniéndose a flote

La representación interna en la computadora de un número real por otra parte, sigue un procedimiento diferente [LPB08; PH04], y no está de mas entender como se realiza. Por ejemplo el número 321.456 tiene una parte entera (321) y otra decimal (456). Esto no es tan fácil de representar en la computadora, sobretodo porque queremos realizar operaciones sobre ellos. La manera en que casi todos los lenguajes incluido Python, representan a los números reales es inspirándose en algo llamado "**Notación Científica**" que para los científicos e ingenieros es simplemente una manera práctica de escribir números muy grandes o muy pequeños. Por ejemplo la masa del electrón es $m_e =$

[2]Esto no es tan sencillo, en realidad para representar un número negativo se requiere de un truquito adicional. La representación numérica en la máquina queda fuera de alcance de este libro, para los curiosos recomendamos [PH04]

[3]No está presente en la familia de Python 2.x, y mucho menos en Python 1.x

0.00000000000000000... bueno ya me cansé, todavía me quedan como diecisiete ceros... ¡Ha pero podemos utilizar la famosa Notación Científica! y escribir $m_e = 9.11 \times 10^{-31}\,kg$. Al escribir los números de esta manera podemos distinguir tres partes diferentes. La mantisa (9.11) la base (10) y el exponente (-31). Por supuesto que como estamos utilizando una computadora la base no puede ser diez, sino dos, ya que se utilizan números binarios. Pero como para todos los números es la misma base, no tiene sentido gastar espacio de memoria guardando esta información que de todas maneras ya se conoce. Por lo tanto solo es relevante guardar la mantisa y el exponente. Con la mantisa tenemos un problema, ella es en si misma un número real, que es lo que precisamente queremos representar. Para solventar este problema refinamos un poco y utilizamos notación científica *normalizada* donde siempre tenemos un dígito *diferente de cero* a la izquierda del punto (0.911×10^{-30} no está normalizado y 91.1×10^{-32} tampoco). En binarios esto quedaría de la siguiente manera:

$$1.xxxx \times 2^{xxxx}$$

donde las x representan tanto 0 como 1. Observemos que la mantisa siempre empieza por 1 entonces, esta información resulta redundante y sólo se toma la parte decimal. Como no podría ser de otra manera, esta forma tan ingeniosa de representar un número real en la computadora tiene nombre, se llama estándar IEEE 754[4].

En resumen si escribimos 9584.35 o 0.00489 o incluso .358 gracias al "." Python sabe que debe aplicar el estándar IEEE 754, como esto implica una notación normalizada se debe correr el punto para que a la izquierda de él, aparezca un dígito diferente de cero. De ahí el nombre de *punto flotante* que es como se conoce a la implementación de los reales en la computadora. Ahora realizar cualquier operación con números punto flotante es fácil pues el punto "flota" hasta el lugar correcto para que el número quede normalizado y el punto deje de ser un problema.

También podemos usar la notación científica directamente cuando queremos escribir números muy grandes o pequeños. Escribimos la mantisa y después el exponente aclarando cuál es el exponente con una letra "e" de exponente.

```
>>> 91.1e-32
9.11e-31
>>>
```

Cuando estemos trabajando se nos presentará comúnmente el caso en que debemos mezclar números enteros con punto flotante, en la mayoría de los lenguajes este es un aspecto al que el programador debe estar muy atento. Afortunadamente a partir de versión 3.0, Python tiene algo maravilloso que se llama conversión implícita, o en el idioma que los computistas suelen hablar, casting implícito. Esto es simplemente que el interprete de Python "se da cuenta" de la mezcla de formatos numéricos y los convierte automáticamente a la forma que mas convenga.

[4]Bueno, Python no cumple *exactamente* con esta normativa. Afortunadamente para el autor, una explicación detallada de las variaciones que utiliza Python, están totalmente fuera del alcance de este libro

```
>>> (5+52.3)*4
229.2
>>> 45/7
6.428571428571429
>>>
```

Esto implica que uno generalmente termina trabajando con punto flotante, ya que en el paso de un punto flotante a un entero se puede perder precisión, como se muestra en el siguiente ejemplo:

```
>>> 93.678954e3
93678.954
>>> int(93.678954e3)
93678
>>>
```

en el cual se ve como se pierden los decimales haciendo uso de una conversión explicita en int(93.678954e3)[5] y el resultado será 123.0 como cabría esperar.

Lamentablemente el punto flotante tiene una limitación que no la tenían los enteros, no podemos trabajar con números arbitrariamente grandes o arbitrariamente pequeños. Esta limitación no la da la memoria de la máquina, es una consecuencia de la implementación del estándar IEEE 754. Así el número mas grande que podemos representar es el:

```
>>> 1.7976931348623157e+308
1.7976931348623157e+308
>>>
```

Esto es sumamente importante, si producto de alguna operación se sobrepasa este número se tendrá una serie sin sentido de ceros y unos, es decir basura. A veces esto ocurre, y cuando ocurre se le da un nombre (es lindo nombrar las cosas) se le llama *error*. Si efectivamente metimos la pata, la máquina no se equivoca, es responsabilidad del programador no cometer este tipo de errores.

Como programando se pueden cometer muchas clases de errores diferentes, a cada uno de ellos se le han dado nombres diferentes. Si, están todos muy bien clasificados y estudiados [Ber11; LPB08] y todos, en última instancia son responsabilidad del programador. Por eso los programadores son gente tan responsable ¿O acaso se han encontrado con alguna aplicación o programa con errores o sistemas operativos que se "cuelgan"? El tipo de error del que estamos hablando aquí se llama técnicamente *error de desborde* (del ingles overflow) . Afortunadamente Python es muy inteligente, y si nos pasamos del máximo permitido nos da una respuesta muy interesante.

[5] Cuando hacemos este tipo de conversión, en realidad estamos aplicando una función llamada float(). Efectivamente la conversión explícita es muy fácil de hacer, acabamos de ver como pasar de un punto flotante a un entero, para pasar de un entero a punto flotante simplemente escribimos float(123). De las funciones nos encargaremos mas adelante.

```
>>> 1.8e308
inf
>>>
```

para los distraídos, la palabra *inf* es en realidad una abreviatura de la palabra infinity y que funciona notablemente bien en castellano. También ocurre algo parecido cuando el número es muy pequeño:

```
>>> 4.9406564584124654e-324
5e-324
>>>
```

y si seguimos bajando encontramos:

```
>>> 2e-324
0.0
>>>
```

Es decir que si tenemos un número menor o igual que 2×10^{-324} se toma como cero. Si inadvertidamente sobrepasamos este límite inferior cometemos un *underflow error* cuya traducción sería algo así como error de "desbordamiento por debajo".

2.4 A Python le sale todo redondo

Muchos números reales (de hecho incontables) como por ejemplo $1/3$, tienen tantas cifras decimales que no se acaban nunca. O para ponerlo de otra manera existen infinitos números reales con infinitas cifras decimales. Las computadoras, al igual que algunos humanos entre los cuales se encuentra el autor, no manejan bien el concepto de infinito. Es mas, en la computadora no se puede representar infinitas cifras decimales. Cada vez que se intenta algo así se produce un *error de redondeo*. Seguidamente se muestra un error de redondeo típico.

```
>>> 5/3
1.6666666666666667
>>>
```

Efectivamente, $5/3$ tiene infinitas cifras decimales y cuando se realiza la operación Python efectúa un redondeo en la última cifra decimal. Por lo tanto se debe tener mucho cuidado al calcular, por ejemplo:

```
>>> (5/3)-(5*(1/3))
2.220446049250313e-16
>>>
```

debería ser cero, pero aunque arroja un valor muy pequeño está lejos de ser cero (es mucho mas grande que el valor numérico de la masa del electrón). Un número racional puede tener infinitas cifras decimales dependiendo de su base. En base decimal un décimo es un número muy bonito y manejable, se escribe así: 0.1, Pero en binarios se vuelve horrendo con infinitas cifras decimales y este efecto es acumulativo.

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>>
```

Véase el error cometido por solo sumar tres veces 0.1. Parece poco pero si se suman 458 veces o mil, el error es apreciable y esta situación es bastante frecuente. Además hay números tan locos e irracionales como π, e, $\sqrt{2}$, etc, que siempre tendrán infinitas cifras decimales en cualquiera de las dos bases.

Por lo tanto es frecuente cuando se trabaja con números punto flotante que la máquina realice automática y silenciosamente algún tipo de redondeo. La manera que tiene Python de redondear puede parecer confusa en un principio. Nosotros estamos acostumbrados a la forma de redondeo que aprendimos en la escuela y que aparece en todo libro de física básica [HRW08]. Por ejemplo, el número 2.3 se redondea a 2.0, el 2.7 se redondea a 3.0 y el número 2.5 también se redondea "hacia arriba" a 3.0, como todos sabemos. Bueno esto último tiene un pequeño problema. La razón es que el "0" no se redondea, por lo tanto de los diez dígitos decimales solo tenemos que redondear 9. Esto conduce a una asimetría que se aprecia bien en el siguiente esquema:

$$\left.\begin{matrix}1\\2\\3\\4\end{matrix}\right\} \text{ redondeo hacia abajo}$$

$$5 \longrightarrow \text{¿?}$$

$$\left.\begin{matrix}6\\7\\8\\9\end{matrix}\right\} \text{ redondeo hacia arriba}$$

La manera clásica (la de la escuela) funciona muy bien cuando tenemos que redondear unos pocos números, pero si son cientos, miles, o millones de números redondeados, y se hacen operaciones entre ellos, se termina con un número mayor del que se debería. Otra forma mala de redondear es hacia abajo ($2.5 \rightarrow 2.0$) en este caso se termina con un numero menor. Adicionalmente debemos contar la influencia en el redondeo de los números negativos, lo cuál empeora la cosa. Por lo tanto es necesario escoger una forma de redondeo mas adecuada. El estándar IEEE 754 define al menos 5 tipos diferentes de redondeo. Python implementa el redondeo que el IEEE 754 trae por defecto. Este se llama *redondeo mitad al par* también conocido como el redondeo del banquero ya que es muy utilizado en el mundo financiero (claro allí cada céntimo cuenta). Se realiza

igual que antes, pero si el numero a redondear es 5, se redondea la cifra hacia arriba solo si el resultado final es par, sino se redondea hacia abajo. Un ejemplo puede aclararle la situación. Supongamos que queremos redondear 27.45 a un número con una sola cifra decimal, entonces el resultado es 27.4. Si en cambio el número fuese 27.75 el resultado sería 27.8. Otros dos ejemplos; 27.35 pasa a 27.4 y 27.25 pasa a 27.2.

El error de redondeo no es un error que cometa la máquina, es mas bien una característica derivada del hecho de no poder manejar infinitos con la computadora. Es responsabilidad del programador tener esta característica en cuenta. Si el programa se vuelve inestable o los resultados obtenidos no tienen sentido a causa del "error de redondeo", el error es del programador.

2.5 Aritmética del punto

Antes de abandonar el apasionante mundo de los números punto flotante, haremos unos últimos comentarios sobre las operaciones entre dichos números[Cou]. Si bien la propiedad conmutativa y la transitiva se cumplen sin problema, la asociativa no. Efectivamente $(a+b)+c$ no necesariamente es igual a $a+(b+c)$ esto se ilustra con un ejemplo:

```
>>> a = 2
>>> b = 0.001
>>> c = 5
>>> (a+b)+c
7.0009999999999994
>>> a+(b+c)
7.001
>>>
```

Es una diferencia pequeña, pero es una diferencia y si tenemos que hacer miles de estas operaciones al final podemos terminar con un resultado que no es el correcto. Lo mismo ocurre con la asociativa para la multiplicación (averigüelo usted mismo).

Como sabemos (yo me enteré hace poco) el conjunto de los números reales es denso, es decir entre dos números reales cualesquiera siempre podemos encontrar otro número real (de hecho podemos encontrar infinitos números). Con los punto flotante no ocurre lo mismo como debe quedar claro si ha leído las secciones anteriores. Esto nos lleva a otra de las propiedades que tampoco se cumplen a cabalidad con los números punto flotante, es la existencia del neutro en la adición. Si existe, lo que sucede es que no es único. En los reales existe solo un número x tal que $a+x=a$ ese número es el cero. Pero con punto flotante la cosa cambia un poco.

```
>>> a = 5.0
>>> b = 2.0e-16
>>> a+b
5.0
>>>
```

lo que significa que hay una enorme cantidad de números (en este ejemplo cualquier número menor que 2.0e-16) que cumplen con la condición $a+x=a$ y por lo tanto x no es único. Este aspecto es tan importante para la gente que estudia análisis numérico que le han puesto un nombre, según Wikipedia *...se llama épsilon de la máquina* $(\varepsilon - mach)$ *al menor valor de una determinada máquina que cumple lo siguiente:* $1.0+\varepsilon - mach > 1.0$

El épsilon es el número decimal más pequeño que, sumado a 1, la computadora nos arroja un valor diferente de 1, es decir, que no es redondeado. Para los curiosos el valor de este número en una máquina de 64 bits es 2.2204460492503131e-16.

Por lo anterior la siguiente propiedad matemática: *Sean cuatro números cualesquiera* a, b, c, *y* d. *Si* $a > b$ *y* $c > d$ *entonces* $a+c > b+d$
no siempre se cumple.

```
>>> a = 2
>>> a = 2 + 3e-20
>>> b = 2
>>> c = 1
>>> d = 1 - 3e-20
>>> a + c
3.0
>>> b + d
3.0
>>>
```

en este caso $a+c=b+d$ lo cual es incorrecto. Otra cosa interesante es que a veces multiplicar es mejor que sumar

```
>>> x = 1/7
>>> x+x+x+x+x+x+x
0.9999999999999998
>>> 7*x
1.0
>>>
```

Todo lo expuesto en estas secciones se deriva del hecho de no poder representar números arbitrariamente grandes o pequeños con la computadora y a que el conjunto de los números punto flotante no es denso. En resumen la implementación del estándar IEEE 754 es una excelente solución al problema de simular números reales con la computadora, pero tiene ciertas limitaciones que debemos conocer.

2.6 Python no es tan complejo

En un terminal, luego de iniciar la sesión de Python escriba 4+3j+2+2j, la respuesta que arroja el interprete de Python es (6+5j). Si usted escribe cualquier otra letra en lugar de j, por ejemplo “a”, obtendrá un mensaje de error.

```
>>> 4+3a
 File "<stdin>", line 1
   4+3a
```

```
      ^
 SyntaxError: invalid syntax
>>>
```

¿Que tiene de especial la j? Se preguntará el lector. Lo mismo se pregunta el autor, ya que la letra preferida para escribir números complejos es la "i". Tal vez sea porque la i se parece mucho a 1, y la intención fue evitar posibles confusiones. En todo caso Python entiende números complejos, solo que en vez de escribir (40 + i 23) debemos escribir (40 + 23j). Observe que la parte imaginaria se escribe 23j y no j23. Si queremos representar al número imaginario puro "i", simplemente escribimos 1j. Esto significa que en Python se puede hacer todo tipo de operaciones y por lo tanto usted puede resolver problemas (o crearlos) con números complejos. Observemos por ejemplo las siguientes operaciones:

```
>>> (1j)**2
(-1+0j)
>>>
>>> (2+3j)**(-1)
(0.15384615384615385-0.23076923076923078j)
>>>
>>> (1+2j)/(3-4j)
(-0.2+0.4j)
>>>
```

Esto es suficiente sobre complejos por ahora, más adelante tendremos mas oportunidades de complicarnos la vida.

2.7 Variables

Hasta ahora hemos visto algunas cositas sobre como manejar números con Python, pero para sacar provecho a lo aprendido en vez de números, debemos utilizar variables que representan a esos números. Afortunadamente no es necesario una declaración del tipo de número, ya que Python reconoce automáticamente si el número es entero, punto flotante, o complejo, a esto se le llama tipos dinámicos.

```
>>> a = 5
>>> b = 20
>>> a+b
25
>>> c = a-b
>>> c
-15
>>> a = -321
>>> a
-321
>>> a+b
-301
>>> z = 2-3j
>>> z
```

```
(2-3j)
>>>
```

Observese que $c = a - b$ no es una ecuación, y "=" no significa "es igual a". Lo que hace el signo "=" en $a = 5$ es reservar memoria en la computadora para una variable llamada "a" y "conectarla" con otro espacio de memoria en la que se encuentra guardado el valor[6] 5. Si mas adelante realizamos $a = -321$ estamos conectando la variable a con un nuevo espacio de memoria que tiene guardado el número -321. Por ejemplo el siguiente ejemplo es incorrecto desde el punto de vista matemático pero totalmente valido en programación y en particular en Python.

```
>>> n = 7
>>> n
7
>>> n = n+3
>>> n
10
>>>
```

lo que se ha echo es *asignar* lo que está a la derecha de "=" a la variable que está a la izquierda de "=". Internamente en la memoria de la computadora conectamos la variable de nombre n con el valor 7, y luego a la variable de nombre n la conectamos con su valor anterior mas 3. De hecho en computación $x = 20$ no es una igualdad sino una asignación, que significa que a la variable x se le asigna el valor 20.

A las variables también se les puede asignar números complejos, de hecho estas variables tienen algunas propiedades interesantes.

```
>>> a = 5-3j
>>> b = -6+10j
>>> c = a+b
>>> c
(-1+7j)
>>> c.real
-1.0
>>> c.imag
7.0
>>> b.conjugate()
(-6-10j)
>>>
```

Los nombres dados a las variables pueden ser casi cualquier combinación de letras y números que utilicen la codificación Unicode[7]. Bueno casi, ya que existen ciertas reglas.

- El primer carácter no puede ser un número. Así m2 es un nombre de variable válido, pero 2m no.

[6] Internamente es un poco mas complejo. Python no maneja variables si no mas bien referencia a objetos. En capítulos posteriores se hablará de objetos, pero de todas maneras este manejo interno es transparente al usuario.

[7] Esto se verá mas adelante en la sección 4.1

- Al utilizar codificación unicode se pueden usar letras propias del idioma español, lo que no ocurre con otros lenguajes como el C/C++, Fortran, Java, etc. Es perfectamente válido escribir : año = 2014, o papá = 32.
- No se pueden utilizar símbolos, a@g?3 no es un nombre de variable válido.
- Existen ciertas palabras que están reservadas para el uso del interprete Python y no se pueden usar como variables, aún no hemos visto ninguna, pero si quiere verlas todas diríjase al apéndiceA.

2.7.1 En Python no hay tipos malos

Ya hemos visto tres tipos de datos diferentes, tipo entero, tipo flotante y tipo complejo. Pero en Python hay muchos más, como el lector podrá apreciar si tiene la paciencia de seguir leyendo el libro. Los tipos de datos son algo a lo que el programador debe prestar mucha atención. Afortunadamente como Python tiene conversión implícita se puede operar con diferentes tipos sin que haya mayor problema.

```
>>> z = 5+2j
>>> a = 20
>>> b = -3.5
>>> z -a + b
(-18.5+2j)
>>>
```

En el ejemplo anterior se declararon variables de los tres tipos de datos que conocemos hasta hora, luego se "mezclaron" en una suma, pero gracias a la conversión implícita se calcula el resultado esperado. Por esto se dice que Python posee tipos dinámicos. Si se nos olvidó de que tipo es determinada variable, solo tenemos que preguntarlo.

```
>>> type(z)
<class 'complex'>
>>> type(a)
<class 'int'>
>>>
```

La función[8] *type()* (tipo) responde con la clase de dato al que pertenece la variable. Pero si lo que queremos es saber si la variable *b* es del tipo flotante, debemos preguntar con la función *isinstance(b, float)*. Si *b* es del tipo punto flotante, Python responderá verdadero (True), por supuesto si no es un "float" responde falso (False).

```
>>> isinstance(b, float)
True
>>> isinstance(a, float)
False
>>>
```

Por supuesto lo mismo se aplica para el tipo *int* o cualquier otro tipo.

[8] Ya se, aún no hemos visto que son las funciones en Python, pero falta poco.

2.8 Incompletándolo todo

Por razones de completitud debería hablar un poco sobre la representación de números octales (base 8), hexadecimales (base 16) e incluso binarios. En lugar de hacer eso, solo daré algunos ejemplos.

```
 >>> oct(123)
'0o173'
>>> hex(123)
'0x7b'
>>> bin(123)
'0b1111011'
>>>
```

Por último en vez de trabajar con enteros o punto flotante también lo podemos hacer directamente con decimales:

```
 >>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857')
>>>
```

muy interesante, pero le daremos poca utilidad en este libro. Si usted intenta reproducir el ejemplo anterior en el terminal obtendrá:

```
 >>> Decimal(1) / Decimal(7)
 Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
 NameError: name 'Decimal' is not defined
>>>
```

lo que evidentemente es un (inentendible) mensaje de error. Con un poco de suerte, en el próximo capítulo o en el que le sigue estará en condiciones de entender que ha ocurrido.

Finalmente una pequeña observación acerca del de uso del interprete de Python en forma interactiva. Una ayuda importante que nos brinda el terminal es que tiene "memoria", esto significa que se acuerda de los comandos que hemos tecleado. Si cometemos algún tipo de error no tenemos que repetir todo desde el principio, sino que con la ayuda de la tecla "flecha hacia arriba" o "flecha hacia abajo" podemos ir ejecutando todas las instrucciones ya escritas y corregir solo aquella donde se cometió el error. A esto se le llama *historia*. Es decir el terminal conoce la historia de todas las instrucciones tecleadas en el terminal.

2.9 Ejemplos y (un) ejercicio.

Como este pretende ser una especie de libro de texto, es necesario que al menos al final de capítulo aparezcan algunos ejemplos y ejercicios. Para seguir tan noble tradición presentaré aquí algunos ejemplos y un extraordinario ejercicio final.

■ **Ejemplo 2.1 — ¡Que pesado!.** Un cuerpo tiene una masa de 50 kg ¿Cuál es su peso?
Respuesta: Como sabemos $F = ma$, en este caso la aceleración es la de la gravedad $g = 9.8\,m/s^2$, y la masa $m = 50\,kg$. Por lo tanto el peso del cuerpo es:

```
>>> g = 9.8
>>> m = 50
>>> P = m*g
>>> P
490.00000000000006
>>>
```

En primer lugar se nota claramente que en el cálculo no se tomaron en cuenta las unidades. Es el programador que debe tomar en cuenta esto y diseñar el código con las unidades colocadas correctamente. Lo segundo es que la respuesta correcta debería ser 490 y no 490.00000000000006. Este es un típico error del que ya hemos hablado, como se ha hecho notar anteriormente, es responsabilidad del programador manejar este tipo de errores. ■

■ **Ejemplo 2.2 — ¿Está muy fría?.** Un termómetro mide una temperatura para un vaso de agua caliente de 123 °F ¿Cuál es su temperatura en grados Celsius?
Respuesta: Un grado Fahrenheit equivale a 0.556 Celsius por lo tanto.

```
>>> 123*0.556
68.388
>>>
```

y la respuesta correcta es que la taza está a 68.388°C. ■

■ **Ejemplo 2.3 — Móvil en SI.** Un automóvil viaja por la autopista a 108 km/h. ¿A que velocidad viaja si utilizamos el Sistema Internacional de Unidades?
Respuesta: Obviamente la respuesta es que viaja a la misma velocidad, ya que un cambio en el sistema de unidades no cambia la velocidad a la que se mueve el móvil; pero el valor numérico si cambia. Como sabemos $1km = 1000\,m$ y $1h = 3600\,s$, así que la operación de conversión que se debe hacer es $108\frac{km}{h}\,(\frac{1000m}{1km})\,(\frac{1h}{3600s})$

```
>>> 108 * (10/36)
30.0
>>>
```

lo que arroja un resultado de $30\,m/s$ ■

■ **Ejemplo 2.4 — Ejemplo complejo.** Comprobar que: *a)* $(\sqrt{2}-i)-i(1-\sqrt{2}\,i) = -2i$; *b)* $\overline{\frac{(2+i)^2}{3-4i}} = 1$

```
a)
>>> from math import sqrt
>>> a = sqrt(2)-1j
>>> b = 1-sqrt(2)*1j
```

```
>>> a - 1j*b
-2j
>>>

b)
>>> a = 2+1j
>>> b = 3-4j
>>> c = a^2
>>> c = a**2
>>> c.conjugate()/b
(1-0j)
>>>
```

■

En vista de que ya se han realizado una serie de interesantísimos y difíciles ejemplos, es hora de que el lector siga por su cuenta. Propongo entonces que realice al pié de la letra el siguiente ejercicio.

Ejercicio 2.1 Realice meticulosamente la siguiente secuencia de acciones:

- Tome un libro de física general de su preferencia.
- Escoja al azar tres o cuatro ecuaciones que le parezcan particularmente interesantes.
- Resuelvalas y encuentre el resultado utilizando Python.
- Compare esta forma de encontrar la respuesta, con la que obtendría usando una calculadora normal.
- Dese cuenta que no hay ninguna ventaja significativa y decepciónese.
- Recapacite y piense antes de borrar la copia digital que está leyendo, que apenas hemos aprendido las operaciones básicas y que tal vez la cosa se ponga mejor en capítulos posteriores.
- Siga leyendo (aunque no necesariamente ahora mismo) el resto del libro, con la esperanza de que tal vez consiga aprender algo útil después de todo (no doy garantías).

■

3 — Yéndose de bruces con Python

Luego de los tropiezos vienen las caídas, también deberemos darnos la cabeza contra la pared de vez en cuando. Eso es lo natural en casos como este donde uno se propone aprender cosas nuevas, o tal vez debería decir que eso es lo natural en este caso, donde usted insiste en aprender Python utilizando este libro. En cualquier caso, o en cualquiera de todos los casos, a continuación se profundizará un poco mas en el funcionamiento básico de Python. Al funcionamiento complicado el lector deberá dejarlo para mucho después después de haber llegado al final del libro.

3.1 ¿Cómo funcionan las funciones?

Paciencia, para contestar a esta pregunta hay que esperar unos capítulos, pero como necesitamos usar unas cuantas funciones lo antes posible, tenemos que hablar un poco de ellas. Obviamente este un concepto que ha sido tomado de las matemáticas, pero cuando tratamos con lenguajes de computación una función es algo con características propias. Para ser mas precisos una función es un trozo de código bien organizado, con un nombre asociado, que puede ser rehusado las veces que se desee, y que (si está bien diseñada) realiza una relativamente simple y (frecuentemente) única acción [Lan09]. Veremos detenidamente estos tres aspectos (o casi).

Lo de la organización interna es algo que dejaremos para mas adelante, cuando veamos como crear nuestras propias funciones. Por ahora sólo estamos interesados en las funciones predefinidas de Python, y lo único que nos interesa es saber que hacen y como invocarlas. La magia de la invocación de funciones se realiza convocándolas por su nombre. Un buen mago de la programación conoce un gran número de nombres de funciones que utiliza constantemente para sus propios fines. Estos nombres deben tener una estructura especial, la cuál es: ***Nombre_de_la_función****(parámetros)*. Por supuesto el ***Nombre_de_la_función*** no puede ser una palabra reservada (ver apéndice A). Por otra parte, los parámetros a veces son innecesarios y se pueden omitir, pero los paréntesis no.

Un buen ejemplo de función que ya hemos utilizado en la sección 2.7.1 es *float()*.

Esta función tiene un nombre (float), acepta como parámetro el número o variable al que se le quiere cambiar el tipo y regresa el número ahora con el tipo cambiado a float.

```
>>> a = 25
>>> b = float(a)
>>>
>>> b
25.0
>>>
```

otra función que ya hemos visto, que no necesita parámetros y que tampoco regresa ningún resultado es quit(). Otra función interesante es la que obtiene el valor absoluto de un número.

```
>>> a = -3.8
>>> abs(a)
3.8
>>>
```

Un ejemplo de función que acepta mas de un parámetro es la función redondeo

```
>>> round(-8.235816,3)
-8.236
>>>
```

aquí el primer parámetro es el número a ser redondeado, el segundo es el número de cifras decimales que se desean. Si no se quieren cifras decimales, no es necesario el segundo parámetro.

```
>>> round(3.5)
4
>>>
```

observemos que el tipo de redondeo que produce la función round(), es la tradicional (véase la sección 2.4).

En vez de realizar una lista interminable de funciones, se irán exponiendo a medida que se necesiten. El número de ellas es bastante grande y Python al igual que C/C++, Java y casi todos los lenguajes de programación, las agrupa en bibliotecas. Las funciones que hemos visto hasta ahora no están agrupadas en bibliotecas sino que se cargan por defecto cuando se invoca Python en la línea de comandos, pero este número de funciones es muy limitado y está reservado para tareas muy comunes. Las funciones se agrupan en bibliotecas según su tipo y en Python reciben el nombre de *módulos*.

3.2 Cada cosa en su lugar

En los módulos las funciones están organizadas según el tipo de tarea que realicen. Así las que interaccionan con el sistema operativo se encuentran en el módulo “os”, las funciones de hora y fecha se encuentran en el módulo “time” y así sucesivamente. Poco a poco se irán utilizando diferentes módulos y empezaremos con uno que tal vez resulte

de utilidad a alguna que otra persona, el módulo matemático. Supongamos por ejemplo que queremos hallar la componente x de un vector $\vec{a}$ conociendo su magnitud y su ángulo con el eje de las abscisas. Como alguno de los lectores ya sepa $a_x = |\vec{a}|\cos(\alpha)$ (desde ahora en adelante para simplificar la notación denotaremos la magnitud $|\vec{a}|$ simplemente como a). Para realizar el cálculo debemos dar los valores de la magnitud y del ángulo, estos son: $a = 44$ y $\alpha = 60$ Ahora podemos codificar el problema en Python.

```
>>> alfa = 60
>>> a = 44
>>> import math
>>> a * math.cos(alfa)
-41.90617113826688
>>>
```

Lo primero que observamos en el programa anterior es la palabra reservada *import* (al fin apareció una). La instrucción *import math* significa que estamos importando el módulo matemático, este se carga en memoria y así podemos utilizar cualquiera de las funciones que lo conforman. Si queremos saber cuales son, solo tenemos que emplear la función *dir()* así

```
>>>
>>> dir(math)
```

y seguidamente aparecerá en pantalla la lista completa de las funciones que pertenecen al modulo math.

En la siguiente instrucción (a * math.cos(alfa)) se realiza el cálculo que deseamos. Observemos aquí que para que Python reconozca a la función coseno necesitamos anteceder al nombre de la función, el nombre del módulo al cual pertenece seguido de un punto. Esta sintaxis se hereda de la manera en que muchos lenguajes (como el C++ por ejemplo) se refieren al los objetos. Esto es así porque efectivamente en Python todo son objetos, pero no nos preocupemos por eso ahora, simplemente seamos felices utilizando el módulo matemático. Claro, la felicidad sería completa si el resultado estuviese correcto. Efectivamente el coseno de sesenta grados es 1/2 y el resultado por lo tanto, debería ser 22 y no -41.90617113826688. Es muy doloroso admitir que Python se ha equivocado y por lo tanto no lo haremos. Lo que pasa es que en ningún momento se ha dicho que se estaba calculando el coseno de sesenta *grados*. En Python las funciones trigonométricas utilizan *radianes*. Para ejemplificar lo dicho se debería repetir el ejemplo con la conversión de sesenta grados a radianes, pero para que molestarse si tenemos la función *radians*

```
>>> math.radians(60)
1.0471975511965976
>>>
```

también podemos hacer lo contrario, o casi.

```
>>> math.degrees(1.0471975511965976)
59.99999999999999
>>>
```

Puede ser que el nombre del módulo no nos guste (math es bastante feo) o puede que sea muy largo (un módulo que utilizaremos más tarde se llama matplotlib), no hay problema podemos usar un alias.

```
>>> import math as m
>>> m.radians(60)
1.0471975511965976
>>>
```

En la sección de código anterior se ha importado el módulo matemático con el nombre "m" en vez de "math". Si lo que no nos gusta es la sintaxis estilo objetos, entonces podemos importar directamente la función, sin que luego tengamos que hacer referencia al módulo, utilizando la forma *from* módulo *import* función.

```
>>> from math import sin, pi
>>> sin(pi/6)
0.49999999999999994
>>>
```

En el ejemplo anterior se ha importado del módulo matemático la función seno "sin()", y la constante π "pi". En esta forma de importar funciones de los módulos, también se pueden utilizar los alias.

```
>>> from math import log as ln, sin as sen, pi
>>> ln(sen(pi/2))
0.0
>>>
```

Si por el contrario quisiéramos importar todas las funciones del módulo matemático simplemente haríamos *from math import* *. Donde * significa "todas" (las funciones y constantes).

```
>>> from math import *
>>>
```

y las podríamos usar sin hacer referencia al módulo.

Con el ánimo de confundir aún más al lector, en este libro se usarán indistintamente cualquiera de las formas de importar funciones vistas anteriormente. La predilección en un momento dado de una u otra forma, está dada únicamente por el capricho del autor. La excusa es que el lector se verá frecuentemente (considerando la calidad del presente libro) en la necesidad de complementar sus estudios leyendo código en otros libros y en Internet. Al hacer esto, debe estar preparado para encontrarse cualquiera de las formas de importar funciones antes mencionadas.

Llegado este punto, no está de más presentar algún que otro ejemplo.

■ **Ejemplo 3.1 — El ángulo.** ¿Cuál es el ángulo entre los vectores $\vec{a} = -5\hat{i} + 3\hat{j}$ y $\vec{b} = 4\hat{i} + 2\hat{j}$?

Respuesta: Como se encuentra explicado en cualquier libro de física general [HRW08; SJ10], el producto escalar de dos vectores está definido como

$$\vec{a} \cdot \vec{b} = ab\cos(\alpha)$$

Para despejar el ángulo α debemos calcular primeramente las magnitudes de los vectores así como su producto $\vec{a} \cdot \vec{b}$. La magnitud se obtiene gracias a Pitagoras: $a = \sqrt{a_x^2 + a_y^2}$ y el producto es simplemente: $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y$. El cálculo a realizar es desde luego:

$$\alpha = \cos^{-1}\left(\frac{\vec{a} \cdot \vec{b}}{ab}\right)$$

a continuación el código que resuelve este problema.

```
>>> from math import sqrt, acos
>>> a = sqrt( ((-5)**2) + (3**2) )
>>> b = sqrt( (4**2) + (2**2) )
>>> apuntob = ((-5)*4) + (3*2)
>>> alfa = acos( apuntob/(a*b) )
>>> alfa
2.137525544318403
>>>
```

Primeramente importamos del módulo matemático las funciones raíz cuadrada (*sqrt()*) y arcocoseno (*acos()*). Luego en la segunda y tercera línea calculamos la magnitud de $\vec{a}$ y $\vec{b}$ respectivamente. En la cuarta línea guardamos en la variable "apuntob" el producto escalar de los vectores, y a continuación en la quinta línea hallamos el ángulo. Finalmente le pedimos a Python que imprima el resultado, que desde luego viene expresado en radianes.

■

■ **Ejemplo 3.2 — Raíces de un polinomio de segundo grado.** Supongamos que pateamos una pelota desde el suelo de tal manera que la velocidad con que sale la pelota forma un ángulo con la horizontal de $\beta = 60°$ y su rapidez es $v = 6m/s$. ¿Cuánto tiempo debemos esperar para que llegue a una altura $h = 1.0\,m$?
Respuesta: Nuevamente guiándonos por nuestro texto de física general preferido, encontramos que la relación que necesitamos es:

$$y = y_o + v_{yo}t - \frac{1}{2}gt^2 \quad \text{que podemos reescribir como} \quad \frac{1}{2}gt^2 - v_{yo}t + y = 0 \quad (3.1)$$

Donde y_o es la altura inicial, que para este problema en particular es igual a cero, $y = h$ es la altura final, $v_{yo} = v_o sen(\beta)$ es la velocidad inicial en las coordenadas y, g es la aceleración de la gravedad y t el tiempo. Como la ecuación 3.1 es un polinomio de segundo grado en t tendrá dos soluciones de la forma

$$t_1 = \frac{v_{yo} + \sqrt{(-v_{yo})^2 - 4(\frac{1}{2}gh)}}{2(\frac{1}{2}g)} \quad \text{y la otra raíz es} \quad t_2 = \frac{v_{yo} - \sqrt{(-v_{yo})^2 - 4(\frac{1}{2}gh)}}{2(\frac{1}{2}g)}$$

(3.2)

El código en Python para resolver este problema es:

```
>>> import math as m
>>> h = 1.0
>>> vo = 6
>>> g = 9.8
>>> beta = m.radians(60)
>>> vyo = vo*m.sin(beta)
>>> t1 = ( vyo + m.sqrt(vyo**2 - 2*g*h) )/(g)
>>> t1
0.8078006657605836
>>> t2 = ( vyo - m.sqrt(vyo**2 - 2*g*h) )/(g)
>>> t2
0.25263860417954526
>>>
```

■

3.3 Python encadenado

En el capítulo anterior se explicó un poquito como funcionaban las variables numéricas, pero sobre variables hay mucho más que decir. Ahora se hará referencia a las variables que constan de cadenas de caracteres. Encadenando caracteres se forman palabras, con estas, oraciones y esto puede tener cierta utilidad siempre y cuando se tenga algo que decir. El tipo de variables que maneja cadenas o secuencias de caracteres en Python se llama *string*[1] (str) y para reconocerlas se deben poner entre comillas. A continuación algunos ejemplos:

```
>>> a = 'a'
>>> a
'a'
>>> b = 'Esto es una cadena de caracteres'
>>> b
'Esto es una cadena de caracteres'
>>> cad1 = ' y se pueden concatenar'
>>> cad1
' y se pueden concatenar'
>>>
```

Efectivamente como son variables se pueden concatenar, simplemente realizando una suma.

```
>>> b + cad1
'Esto es una cadena de caracteres y se pueden concatenar'
>>>
```

[1] La palabra "string" significa literalmente cuerda, pero como "cuerda de caracteres" no suena muy bien, utilizamos mejor cadena de caracteres.

Lamentablemente, no se “desconcatenan” restando. Y tampoco se pueden mezclar con números ya que estos son tipos incompatibles.

```
>>> c = 55
>>> cad1 + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

En cambio los números también se pueden ver como secuencias de caracteres.

```
>>> cad2 = '0123456789'
>>> cad2
'0123456789'
>>> cad3 = '01 esto es válido 23'
>>> cad3
'01 esto es válido 23'
>>>
```

Lo que también se puede hacer es la conversión entre tipos

```
>>> c = 55
>>> cad2 = '0123456789'
>>> str(c)
'55'
>>> int(cad2)
123456789
>>> float(cad2)
123456789.0
>>>
```

Aunque parezca mentira hay personas a las que le desagradan las comillas simples, para ellas existen las comillas dobles que son totalmente equivalentes. Solo hay que tener cuidado de no mezclar los dos estilos en una misma sentencia.

```
>>> cad4 = "otra cadena mas"
>>> cad4
'otra cadena mas'
>>>
```

Siempre es muy fastidioso estar repitiendo lo mismo una y otra vez, pero afortunadamente Python nos puede ayudar un poco.

```
>>> cad4 = ' ya te lo dije,' * 4
>>> cad4
' ya te lo dije, ya te lo dije, ya te lo dije, ya te lo dije,'
>>>
```

Tal vez este sea el momento oportuno para introducir una nueva función predefinida, tal vez no, en todo caso aquí va. La función es *print()*, maravíllense con su utilidad.

```
>>> a = 3
>>> print(a)
3
>>> b = " es un número primo"
>>> print( str(a) + b )
3 es un número primo
>>>
```

Bueno tal vez no parezca tan útil por ahora, pero a medida que avancemos iremos descubriendo su poderío y sus truquitos. El primero de ellos es el siguiente:

```
a = 'A veces uno tiene muchas cosas que decir y \nno hay \
... suficiente espacio, eso es \nverdaderamente desagradable'
>>> print(a)
A veces uno tiene muchas cosas que decir y
no hay suficiente espacio, eso es
verdaderamente desagradable
>>>
```

En el miniprograma anterior hay varias cosas que explicar, en primer lugar está "\n" y luego "\". Empecemos por este último. Cuando estamos programando y una sentencia o declaración es demasiado grande como para quepa en un renglón, simplemente usamos \ y podemos usar dos renglones para la misma línea de comando. Por otra parte print() entiende \n como un caracter especial que significa "salto de línea". El salto de línea forma parte de un grupo de caracteres especiales llamados *secuencias de escape*.

Secuencia de escape	Significado
\a	Campana
\b	Retroceso
\e	Escape (Esc)
\f	Salta una línea y continúa en la misma columna
\n	Nueva línea
\r	Salto de línea
\t	Tabulador (Tab)
\v	Tabulador vertical

Tabla 3.1: *Secuencias de escape*

En el cuadro 3.1 se encuentran todas las secuencias de escape[2] que maneja la función print(). El lector puede sentirse libre de experimentar con todas ellas, una y otra vez hasta que las domine, o se aburra, lo que ocurra primero. Ahora empezamos a entender

[2]Hace mucho, mucho, mucho tiempo, en la época de las máquinas de escribir, la diferencia entre \n y \r era importante, esta última se utilizaba para un nuevo párrafo, mientras que la anterior sólo para una nueva línea. Actualmente se usa casi exclusivamente \n.

la utilidad de print(), con esta función se le puede dar formato a una salida de texto para que sea mas amigable y entendible al usuario. Mas adelante volveremos sobre este tema.

Nos falta una última forma de crear cadenas

```
>>> '''Las cadenas también se pueden crear con tres comillas,
... la diferencia es que se pueden utilizar varios renglones
...
... e inclusive se pueden dejar algunos en blanco,
... y utilizar caracteres "especiales" # @ Se
... suelen utilizar en la documentación de programas'''
'Las cadenas también se pueden crear con tres comillas,\nla d
iferencia es que se pueden utilizar varios renglones \n\ne in
clusive se pueden dejar algunos en blanco, \ny utilizar carac
teres "especiales" # @ Se\nsuelen utilizar en la documentació
n de programas'
>>>
```

Las cadenas en Python son un ejemplo de un tipo especial de dato llamado *secuencia*. Las secuencias son una colección posicionalmente ordenada de objetos[3][Lut13]. Si la secuencia es una cadena, se tendría simplemente una colección ordenada de caracteres. Y como están ordenadas cada uno de sus caracteres se pueden ubicar con un índice.

```
>>>
>>> cadena = "Esta es una secuencia ordenada de caracterres"
>>> cadena[0]
'E'
>>> cadena[10]
'a'
>>>
```

Los *índices* se colocan entre paréntesis rectos y comienzan en cero. Con la función *len()* obtenemos su longitud, que en este caso es 44, como los índices empiezan en cero el último índice tiene que ser 43. Si nos pasamos de este valor obtenemos un error.

```
>>> len(cadena)
44
>>> cadena[43]
's'
>>> cadena[44]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

lo que si podemos hacer es utilizar índices negativos o cualquier expresión matemáticamente válida.

[3]Mas adelante en el libro se aclarará un poco más que son esas cosas llamadas objetos

```
>>> cadena[-1]     #Es equivalente a cadena[43]
's'
>>> cadena[-2]     #Es equivalente a cadena[42]
'e'
>>> cadena[len(cadena)-10]     #Comienzo de la palabra "caracteres"
'c'
>>>
```

también podemos tomar un pedazo o porción (*slice* en ingles). Observese que los espacios en blanco cuentan como caracteres.

```
>>> cadena[12:21]
'secuencia'
>>> cadena[9:18]
'na secuen'
>>>
```

si queremos una porción desde el principio, o una hasta el final, no tenemos que escribir los índices completos

```
>>> cadena[0:18]
'Esta es una secuen'
>>> cadena[:18]
'Esta es una secuen'
>>> cadena[18:len(cadena)]
'cia ordenada de caracteres'
>>> cadena[18:]
'cia ordenada de caracteres'
>>>
```

3.4 Decisiones ¡Ave María!

Efectivamente, en la vida al igual que en Python constantemente se tienen que tomar decisiones, por mas duras que estas sean. Decidir por ejemplo, cuando un número es mayor que otro puede ser un verdadero dolor de cabeza para los expertos. En esta sección se ayudará al lector (en la medida de lo posible) a pasar sano y salvo por esta dura prueba. Lo primero para tomar una decisión acertada es hacer la pregunta correcta. Debe tenerse en cuanta primeramente que Python solo contesta con "si" o con "no" a las preguntas que le hacemos, o mejor dicho con "verdadero" o "falso" o como dice Python "True" or "False". Si queremos saber si 3 es mayor que 1, preguntamos:

```
>>> 3>1
True
>>>
```

al ">" se le suele llamar *mayor que* y es uno de los varios operadores lógicos que existen en Python. Es mas el número de preguntas que se pueden hacer con los operadores

lógicos es bastante variado, estas son:

$>$ mayor que
$<$ menor que
$>=$ mayor igual que
$<=$ menor igual que
$==$ igual que
$!=$ diferente a

Estas son las únicas preguntas que necesitamos para tomar una buena decisión y pasar a la acción, es decir:

si una proposición es verdadera:
    se hace tal cosa
si no lo es:
    se hace otra cosa

En Python para hacer este tipo de pregunta se utiliza *if* y *else* y se escribe así

```
>>> if a>=b :
...     a+b
...     c = a*(b/5)
... else:
...     a-b
```

Lo anterior significa que si a es mayor que b entonces se sumará $a+b$ y luego a c se le asignará el resultado de $a*(b/5)$. Pero en caso de que la proposición sea falsa, es decir si a es menor que b, entonces se resta $a-b$. Observemos muy bien la forma en que se ha expresado la pregunta, en primer lugar la pregunta se finaliza con dos puntos":". Luego en el siguiente renglón se utiliza una sangría, de hecho todo lo que queremos hacer si la proposición es verdadera debe ir con la *misma* sangría, no es importante el ancho, pero es obligatorio usarla. Cuando ya hemos hecho todo lo que necesitábamos hacer si la proposición era verdadera, entonces dejamos de usar la sangría. Los tres puntos, Python los coloca automáticamente y remplazan a $>>>$ para recordarnos que se debe usar sangría. La palabra reservada *else* no es obligatoria, ya que no siempre estamos interesados en hacer algo específico si la proposición es falsa. Pero en caso que necesitemos usarla sigue las mismas reglas que el *if*. Se debe utilizar los dos puntos y luego lo que sigue debe expresarse con sangría. Como ejemplo véase el cálculo de $a*log(|b|)$.

```
>>> import math as ma
>>> a = 10
>>> b = -30
>>> if b<0:
...     b = -b
...     print("pasamos por aquí")
...
pasamos por aquí
```

```
>>> a * ma.log(b)
34.011973816621555
```

a propósito coloqué un valor de b negativo, de tal manera que la pregunta b<0 tiene como respuesta True y se realiza el cambio de signo (b = -b). De esta manera podemos calcular el logaritmo adecuadamente. Si por el contrario se hubiese colocado un valor de b positivo, por ejemplo b = 100, la respuesta hubiera sido False y no se ejecuta la instrucción b = -b. Compruébelo y verá que no pasamos por aquí.

■ **Ejemplo 3.3 — Raíces de un polinomio de segundo grado (parte 2).** En el ejemplo 3.2 calculamos las raíces de un polinomio de segundo grado para hallar la solución a un problema de cinemática. A continuación lo refinaremos un poco.

```
>>> import math as m
>>> h = 1.0
>>> vo = 6
>>> g = 9.8
>>> beta = m.radians(60)
>>> vyo = vo*m.sin(beta)
>>> det = vyo**2 - 2*g*h
>>> if( det>=0 ):
...     t1 = ( vyo + m.sqrt(det) )/g
...     t2 = ( vyo - m.sqrt(det) )/g
...     print("El tiempo t1 es: " , t1 , " y el t2 es: " , t2)
... else:
...     print("La raiz es negativa" )
...
El tiempo t1 es:  0.8078006657605836  y el t2 es:  0.25263860417954526
>>>
```

Lo que hemos hecho es preguntar si el radicando es mayor que cero, de no ser así las raíces serían imaginarias, lo que nos daría un tiempo imaginario, pero el tiempo tiene siempre que ser real (matemáticamente hablando). Pruebe hacer el ejercicio anterior con un h suficientemente grande o por el contrario con vyo muy pequeño para comprobar el funcionamiento del algoritmo.

La otra novedad es un nuevo uso de la función *print()*. Anteriormente ya la habíamos utilizado mezclando números con cadena de caracteres, la diferencia es que aquí no concatenamos las variables, sino que escribimos las variables separadamente ... separándolas con comas. Si el lector se fija bien fueron 4 variables, dos cadenas de caracteres y dos números. ■

¿Que pasa si queremos hacer muchas preguntas? ¿Que pasa si tenemos muchas dudas? ¿Que pasa si queremos saber esto y lo otro? ¿Que sucede si no resistimos el impulso de preguntar y preguntar todo el tiempo? Bueno en principio se pueden utilizar tantos *if* como se desee.

```
>>> a = 100
>>> if a>0:
...     print("pasé por aquí")
```

```
>>> if a>100:
...    print("no, ¡pasé por aquí!")
>>> if a<=0:
...    print("se muy bien por donde estoy pasando")
>>> if a!=0:
...   print(a)
```

en el ejemplo anterior se hace una pregunta tras otra. También se pueden tener *if* anidados.

```
>>> a = 100
>>> if a>0:
...    if a>50:
...        print(a)
...    else:
...       print("a no es menor que 50")
... else:
...    print("a no es mayor que 0")
...
100
>>>
```

pruebe lo que pasa para diferentes valores de a.

A veces no queremos hacer muchas preguntas, ya que esto es ineficiente desde el punto de vista computacional, solo queremos seguir preguntando en el caso de que la respuesta a la pregunta anterior sea negativa. Es decir:

si una proposición es verdadera:
    se hace una cosa
si no lo es, pero otra proposición es verdadera:
    se hace otra cosa
si no lo es, pero otra proposición diferente es verdadera:
    se hace otra cosa diferente
en caso contrario:
    terminamos haciendo cualquier otra cosa

lo anterior escrito de tal manera que Python lo entienda es:

if una proposición es verdadera:
    se hace una cosa
elif otra proposición es verdadera:
    se hace otra cosa
elif otra proposición diferente es verdadera:
    se hace otra cosa diferente
else:
    terminamos haciendo cualquier otra cosa

véase que *elif* es una contracción de *else* e *if*. En código esto sería algo así:

```
>>> a = 100
>>> if a>5000:
...    print("pasé por aquí")
>>> elif a>150:
...    print("no, ¡pasé por aquí!")
>>> elif a<=0:
...    print("se muy bien por donde estoy pasando")
>>> else:
...   print(a)
>>>
```

Ahora que ya sabe hacer preguntas, puede ir con toda seguridad a la sección siguiente.

3.5 ¿Me lo puede repetir por favor?

La ventaja fundamental que tienen las computadoras es que no se aburren. Pueden estar haciendo la misma tarea una y otra vez, miles o millones de veces. Es básicamente por esta razón que a los científicos les gustan tanto. En vez de hacer los tediosos y repetitivos cálculos por si mismos, los científicos se los dan a la computadora y ahora se aburren mientras esperan el resultado. Claro que comparadas con los humanos, las computadoras son muy rápidas haciendo tareas sencillas, que de hecho es lo único que saben hacer. Si parece que hacen cosas complicadas (como que funcione un editor de textos por ejemplo) es porque un ejercito de programadores se las ingenió para que así lo parezca.

Python facilita mucho el arte de repetir las cosas y lo hace con estilo, a tal punto que resulta divertido (bueno, para algunos).

```
>>> contador = 0
>>> while contador<10:
...    print("Repito ", contador, " veces lo mismo.")
...    contador += 1
...
Repito  0  veces lo mismo.
Repito  1  veces lo mismo.
Repito  2  veces lo mismo.
Repito  3  veces lo mismo.
Repito  4  veces lo mismo.
Repito  5  veces lo mismo.
Repito  6  veces lo mismo.
Repito  7  veces lo mismo.
Repito  8  veces lo mismo.
Repito  9  veces lo mismo.
>>>
```

En el ejemplo anterior, lo primero que hemos hecho es crear una variable llamada "contador" que inicializamos a cero, luego se creó una línea que lo que significa en

castellano es algo así como: *mientras contador sea menor que cero, repite lo que sigue*. Para que Python entienda claramente que es *lo que sigue* se necesitan los dos puntos(":") y utilizar sangría. Como se puede apreciar la sentencia *while* necesita evaluar una proposición, que en el ejemplo es *contador<0*, si esta proposición es verdadera, se ejecuta el ciclo hasta que la proposición sea falsa. Desde luego podemos colocar una proposición que nunca sea falsa, en este caso el ciclo se repetirá hasta el fin de los tiempos o hasta que la computadora se dañe, lo que ocurra antes. Si por el contrario la proposición nunca es verdadera, entonces nunca se realiza el ciclo.

La línea *print("Repito...")* no debería causar problemas al lector, imprime en pantalla en primer lugar una cadena de caracteres (*"Repito "*), luego el valor de la variable *contador* y por último otra cadena de caracteres (*" veces lo mismo."*). Observemos el espacio ente las comillas y las palabras para que estas no queden pegadas a los números.

La línea *contador += 1* es exactamente igual a *contador = contador + 1* solo que escribimos menos, y al escribir menos, nos equivocamos menos. Esto significa que la variable *contador* se incrementa en uno cada vez que se repite el ciclo, y el ciclo se repite diez veces ya que empieza en cero y la condición de parada es *contador<10*. Si se hubiese puesto *contador += 3* la variable *contador* habría aumentado de 3 en 3. También existen operadores similares para la resta $-=$, para la multiplicación $*=$ y para la división $/=$.

La sentencia *while* es muy útil y se usa muchísimo para las repeticiones, que formalmente se denominan *bucles* que es una palabra menos aburrida que repetición, al menos para los computistas.

■ **Ejemplo 3.4 — Contando Conejos.** Supongamos que se tienen dos conejos, un macho y una hembra. ¿Cuantos conejos se tendrán al cabo de un año? Responder esta pregunta le pareció de lo más interesante a un matemático llamado Leonardo de Pisa conocido también como Leonardo Bigollo allá por el año 1202. Claro como buen matemático idealizó un poco el problema. En primer lugar los conejos nunca morían. Se reproducían cada mes generando otros dos conejos (un macho y una hembra) que también se reproducían al mes y así sucesivamente. El período de gestación dura exactamente un mes al cabo del cual los conejos padres se vuelven a aparearse. Por lo tanto al principio tenemos cero conejos, al mes tenemos el primer par. Al segundo mes ocurre el apareamiento, pero seguimos con un solo par. Al tercer mes aparecen los primeros hijos para tener dos pares de conejos. Al cuarto mes los padres vuelven a parir otro par de conejos, pero los hijos nacidos el mes anterior aún no tienen sus propios hijos, por lo tanto el número de pares de conejos es 3. Al quinto mes aparecen los nietos (un nuevo par), adicionalmente los padres originales vuelven a tener otro par de hijos. Esta secuencia es como sigue:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 ...

Esta secuencia de números es demasiado idealizada para que sea de utilidad en la crianza de conejos (aunque estos si se reproducen muchísimo) pero si se encuentra representada mas o menos fielmente en varios contextos biológicos. Como por ejemplo, la secuencia en las ramas de los árboles, en la disposición de las hojas en el tallo, en el árbol genealógico de los machos de una colmena de abejas, en la flor de la alcachofa, además de numerosas aplicaciones en ciencias de la computación, matemáticas y teoría de juegos[Wikb].

Leonardo de Pisa fue un matemático muy importante en la edad media, que entre otros logros, fue quien introduzco el sistema de numeración arábigo en occidente. Adicionalmente parece que fue el hijo (filius) de un hombre muy bien intencionado o simple (Bonacci) de ahí su apodo póstumo Fibonacci. No dudo que ser hijo de un hombre bien intencionado tenga sus méritos, pero Leonardo de Pisa no lo necesita ya que brilla con méritos propios[Wika]. A pesar de eso la costumbre se ha impuesto y la secuencia de números anterior se conoce con el nombre de *sucesión de Fibonacci*.

Lógicamente lo que viene a continuación es hacer un código que calcule automáticamente la sucesión de Fibonacci para un número arbitrario de términos. Para ello observamos que un número cualquiera de la sucesión se forma con la suma de los dos anteriores. Por ejemplo 21 + 34 = 55 o 233 + 377 = 610. Claro que esta regla no funciona con el primer y segundo número, de hecho estas son las condiciones iniciales para toda la sucesión. Pero en general podemos comenzar en cualquier parte, solo debemos dar dos números consecutivos en la sucesión, calcular el tercer número consecutivo y luego repetir el proceso. He aquí el código.

```
>>>
>>> term1 = 0 #Primer término requerido para comenzar la sucesión
>>> term2 = 1 #Segundo término requerido para comenzar la sucesión
>>> cont =0 #Contador para la condición de parada en el bucle while
>>> while cont<10:
...     term = term1 +term2
...     print(term)
...     term1 = term2
...     term2 = term
...     cont += 1
...
1
2
3
5
8
13
21
34
55
89
>>>
```

■

En el ejemplo anterior se creo el primer término llamado *term1* el cual se inicializó a 0, Luego viene algo muy interesante que utilizaremos a menudo. Se llama comentarios. Al escribir código en Python, cualquier cosa que comience con el caracter "#" se ignorará. Por lo tanto después de "#" podemos escribir cualquier cosa que no afecta para nada al programa. En particular podemos mandarnos un mensaje a nosotros mismos y recordarnos cosas sobre el programa o aclarar ciertos aspectos. Por lo tanto ya no hace falta aclarar que hace la segunda línea ya que el comentario lo explica por si

mismo. La tercera línea también esta aclarada en su comentario. Luego se inicia el ciclo o bucle while, del cual solo saldremos cuando la variable *cont* sea igual o mayor que 10. Dentro del ciclo, creamos un nuevo término llamado *term* que es la suma de *term1* y *term2*. Seguidamente lo mandamos a imprimir, ya que justamente este es el resultado que deseamos ver. Lo que viene a continuación es muy importante para que el algoritmo funcione correctamente y es una técnica que se utilizará repetidamente. Lógicamente term1 no puede valer siempre 0 y term2 siempre 1 ya que en este caso obtendríamos siempre el mismo resultado. Para obtener un nuevo término en el siguiente ciclo, debemos utilizar los nuevos valores para term1 y term2. Estos nuevos valores son los siguientes en la sucesión, así el valor de term2 se lo pasamos a term1 y el valor de term se lo pasamos a term2. Después de hacer esto estamos en condiciones de seguir con el siguiente ciclo del bucle, solo queda incrementar el contador *cont* en uno para alcanzar luego de 10 ciclos, la condición de parada cuando la instrucción cont<10 sea falsa.

Simplemente por puro placer y para dejar claro de que las cosas se pueden hacer de mas de una manera, volveré a codificar la sucesión de Fibonacci de una forma levemente distinta

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
>>>
```

En la primera línea aparece una asignación múltiple, que significa que le asignamos valor a mas de una variable al mismo tiempo, en nuestro caso a *a* y a *b*. Esta vez no hemos usado un contador para salir del ciclo while sino que usamos la propia variable b, solo se saldrá del ciclo cuando la proposición *b <1000* sea falsa. Dentro del ciclo imprimimos b y luego en la siguiente línea otra vez una doble asignación donde actualizamos las variables.

El tipo de relación que existe entre los términos de la sucesión de Fibonacci se presenta muy seguido en programación. Para abreviar llamaremos a los términos simplemente t. En la sucesión de Fibonacci, el primer término vale 0, el segundo 1, el tercero 1, y así sucesivamente, esto lo podemos denotar de la siguiente manera $t_1 = 0$, $t_2 = 1$, $t_3 = 1$, $t_4 = 2$, $t_5 = 5$... A un término genérico lo denotamos con el subíndice "i" y su nombre se conoce como término i-esimo: t_i. Ahora podemos escribir matemáticamente la relación entre los términos de la sucesión.

$$t_{i-2} + t_{i-1} = t_i$$

En el contexto de sistemas dinámicos a este tipo de ecuación se le llama mapa. Finalmente la sucesión de Fibonacci queda totalmente determinada de la siguiente manera.

$$\begin{aligned} t_1 &= 0 \\ t_2 &= 1 \\ t_i &= t_{i-1} + t_{i-2} \quad \text{para} \quad i = 2,3,4,... \end{aligned}$$

3.6 ¿Cual es el objeto de Python?

En Python todo es un objeto, lo cual facilita mucho las cosas porque todo el mundo sabe lo que es un objeto. Desde luego en el ámbito del software los objetos no se pueden tocar, no son algo tangible, pero siguen teniendo todas las demás características. Se pueden manipular, tienen ciertas propiedades, ciertas características y se pueden hacer muchas cosas con ellos. O dicho de otra manera los objetos son piezas de software que simulan cosas. Formalmente hablando los objetos son una colección de datos mas su comportamiento asociado[Phi10].

Usemos un ejemplo, supongamos que queremos modelar a una pelota. Por la parte de los datos tenemos su tamaño, su forma, su color, la posición en el espacio, tal vez su eje de rotación si es que está girando, y su velocidad si se mueve. Todos estos son datos que se pueden codificar en variables, por ejemplo el radio, la posición del centro de masa en sus tres componentes cartesianas, la velocidad angular, etc., serían variables punto flotante; en cambio el color, una cadena como "rojo" o "azul".

Por el lado del comportamiento una pelota se mueve siguiendo las leyes de la cinemática, también habría que considerar la conservación del momento angular, etc. Todas estas cosas se pueden codificar y calcular. Esto se lleva a cabo mediante funciones pertenecientes al objeto, que en el argot de la programación orientada a objetos se les suele llamar *métodos*[4]. En este libro llamaremos indistintamente a las funciones pertenecientes al objeto como *métodos* o simplemente como *funciones*.

Entonces allí está nuestra pelota virtual, que se comportará mas o menos como una real. El nivel de similitud entre el comportamiento de la pelota virtual y la real va a depender directamente de la sofisticación de nuestro modelo.

[4]Esto también depende del lenguaje de programación que utilicemos, en C++ se le suelen llamar *funciones miembro*.

Claro que en nuestra simulación podríamos querer tener muchas pelotas diferentes, no hay problema, ya que una vez que conocemos todas las características que deben tener nuestras pelotas, podremos hacer varios objetos de la misma *clase*. Entonces el procedimiento sería definir una *clase* y luego crear tantos *objetos* pertenecientes a esa clase como queramos.

No vamos a adentrarnos por ahora en las profundidades del paradigma de programación llamado Programación Orientada a Objetos (POO), pero como ya se puntualizó al comienzo de esta sección " en Python todos son objetos" y por lo tanto debemos tener alguna idea acerca de ellos. Ya hemos hecho uso explícito de los objetos puesto que hemos utilizado sin decirlo varios métodos, por ejemplo en la en la sección 2.7 y en el ejercicio 2.4 se utilizó el método que encuentra el conjugado de una variable compleja, así es, como en Python todo es un objeto, las variables también lo son, sean complejas o no. Si a la variable compleja la llamamos *z*, entonces su conjugado es *z.conjugate()*. La sintaxis general sería *objeto.método*, de hecho cada vez que aparezca un punto en alguna parte, seguramente es porque se está llamando al método de un objeto, incluso hasta los módulos son objetos.

3.7 ¿Y ahora que sigue?

En la secuencia de conceptos ideada para este libro, lo que sigue son las secuencias. Las secuencias son un tipo de dato extremadamente poderoso, que consiste en un conjunto ordenado de objetos. Estos objetos están indexados para que podamos saber su orden relativo, y manipularlos individualmente según su indice. Cuando los objetos son simples caracteres entonces la secuencia es una cadena, sobre la cual se ha tratado en la sección 3.3, pero hay otras más. Las que veremos en esta sección son:

- cadenas
- listas
- tuplas

Todas tienen características similares, algunas de las cuales ya hemos visto en las cadenas. Por ejemplo si *sec* representa una secuencia cualquiera, *sec[i]* regresará el i-esimo componente de la secuencia, *sec[i:j]* un pedazo de secuencia entre *i* y *j*. La función *len(sec)* el largo de la secuencia, el operador "+" concatena secuencias, mientras que "*" se utiliza para repetir la secuencia varias veces.

Las secuencias mas que las variables, son el verdadero caballo de batalla de Python, por lo tanto mencionaremos brevemente algunas de sus características, desde luego es utilizándolas en ejemplos concretos donde se aprenderá su verdadero valor.

3.7.1 Cadenas

Las cadenas son iportantísimas para interaccionar con el usuario, que por lo general es un humano. Sobre ellas ya se expuso suficiente en la sección 3.3. Sin embargo no puedo resistir la tentación de agregar algunas líneas con el único propósito de hacer todo esto aún mas aburrido, si esto es posible. Observemos el siguiente trozo de código donde se ilustra el método *find()*.

```
cad1 = "Esta es una aburrida sucesión de caracteres que forma una cadena"
>>> cad1
```

```
'Esta es una aburrida sucesión de caracteres que forma una cadena'
>>> cad1.find("caracteres")
33
>>> cad1.find("aburrida")
12
>>>
```

find traducido al castellano significa "encontrar" y casualmente eso es lo que hace en el código anterior, se le pidió a Python que encontrara a la palabra "aburrida" y este devolvió el índice donde comienza dicha palabra.

Seguidamente ilustraremos una característica importante de las cadenas su *inmutabilidad*. Para ello utilizaremos el método *replace()* que curiosamente al igual que antes hace lo que dice en ingles, "remplaza" caracteres.

```
>>> cad1.replace("aburrida", "divertida")
'Esta es una divertida sucesión de caracteres que forma una cadena'
 >>> cad1
'Esta es una aburrida sucesión de caracteres que forma una cadena'
>>>
```

Observese que *cad1.replace(.ªburrida", "divertida")* remplaza la sucesión de caracteres *aburrida* por *divertida*, pero la cadena *cad1* no cambia, esta sigue con la misma sucesión de caracteres que antes, ya que las cadenas son inmutables. Si queremos obtener una cadena con la palabra *divertida* en vez de *aburrida* entonces debemos crear una cadena nueva.

```
>>>cad2 =  cad1.replace("aburrida", "divertida")
>>> cad2
'Esta es una divertida sucesión de caracteres que forma una cadena'
>>>
```

3.7.2 Python para las listas

Las listas son el objeto tipo secuencia mas general que posee Python, son una colección ordenada de objetos arbitrarios, muy parecido a lo que en otros lenguajes son los arreglos o vectores, pero más generales y flexibles. No poseen tamaño fijo, no poseen tipo único y a diferencia que las cadenas, las listas son *mutables*. La notación para las listas es que comienzan y terminan con paréntesis rectos y sus elementos deben estar separados por comas.

```
>>> [1, "cuervo", 4/7, "pelota plana", 6.87]
[1, 'cuervo', 0.5714285714285714, 'pelota plana', 6.87]
>>> l1 = ["puerta", 5.8, 24]
>>> l2 =[]
>>> l2
[]
>>> l3 = [54, ["lana", "oveja"], l1]
>>> l3
[54, ['lana', 'oveja'], ['puerta', 5.8, 24]]
```

```
>>> l1[1] = "perro"
>>> l1
['puerta', 'perro', 24]
>>>
```

En la primera línea del código anterior se formó una lista con 5 objetos, el primero es el número entero 1, el segundo la cadena de caracteres "cuervo", seguida de la fracción 4/7, luego otra cadena de caracteres y finalmente el número punto flotante 6.87. Mas adelante se crea una lista llamada "l1" formada por una cadena de caracteres, un número punto flotante y un número entero. La lista "l2" es muy interesante se declara una lista, pero no se le asigna ningún valor, es una lista vacía y curiosamente se llama *lista vacía*. Como se ve con l3 las listas también pueden estar formadas por otras listas. Finalmente se muestra como las listas pueden cambiar sus valores, esto ocurre ya que como se dijo anteriormente las listas son mutables.

Como las listas no tienen tamaño fijo, se pueden agrandar o achicar. En el código siguiente a la cadena *l1* se le agrega una cadena de caracteres al final de la lista con el método *append()*, con el método *pop(índice)* se le puede quitar un objeto a la lista indicándole el índice. Este objeto se le puede asignar a una variable.

```
>>> l1
['puerta', 'perro', 24]
>>> l1.append("cielo")
>>> l1
['puerta', 'perro', 24, 'cielo']
>>> fuera = l1.pop(1)
>>> l1
['puerta', 24, 'cielo']
>>> fuera
'perro'
>>>
```

Si lo que queremos es colocar un nuevo objeto en un lugar cualquiera de la lista usamos el método *insert(índice, objeto)*, donde obviamente debemos indicar el índice donde se a de insertar el objeto además del objeto en si mismo.

```
>>> l1.insert(2, "aquí")
>>> l1
['puerta', 24, 'aquí', 'cielo']
>>>
```

El último método que se estudiará en esta sección es *remove(objeto)* y como su nombre lo indica remueve de la lista el objeto especificado. Si el objeto no se encuentra en la lista lanza un mensaje de error.

```
>>> l1.remove(24)
>>> l1
['puerta', 'aquí', 'cielo']
>>> l1.remove(99)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

3.7.3 Tuplas

En matemáticas, una tupla es una secuencia ordenada de objetos. Y eso es justamente lo que en Python es una secuencia, así que vamos bien. La tupla es igual que una lista, la diferencia es que la tupla es *inmutable*. Esto significa que no se aplican ninguno de los maravillosos métodos que hemos visto para modificar las listas, justamente porque las tuplas no se pueden modificar, pero si funcionan todos los métodos generales que se apilan a las secuencias. Para crear una tupla se utilizan comas.

```
>>> t1 = 1, 2, 3, "todo al revés"
>>> t1
(1, 2, 3, 'todo al revés')
>>>
```

Por razones de claridad se suele escribir a las tuplas entre paréntesis, pero no es obligatorio. Especial cuidado hay que tener con la notación cuando se crean tuplas de un solo elemento.

```
>>> t2 = (5)
>>> type(t2)
<class 'int'>
>>> t3 = (5,)
>>> type(t3)
<class 'tuple'>
>>>
```

Cuando en el código anterior se le pregunta a Python de que tipo es *t2*, responde que es un objeto que pertenece a la clase entero (es decir, es un número entero). Pero cuando hacemos la misma pregunta con *t3* respuesta es que pertenece a la clase tupla (tuple en ingles). La diferencia es que *t3* se creó con una coma al final y las comas son lo que definen a las tuplas, no los paréntesis.

Una propiedad importante de las listas y tuplas es que se pueden *desempaquetar*, en el código que sigue la tupla t2 se desempaqueta en tres variables enteras y una cadena.

```
>>> t2 = 1, 2, 3, "todo al revés"
>>> x, y, z, cad = t2
>>> x
1
>>> y
2
>>> z
3
>>> cad
'todo al revés'
>>> type(z)
```

```
<class 'int'>
>>>
```

Todavía quedan más cosas por aprender acerca de las secuencias, pero lo dicho hasta aquí alcanza para nuestros fines, para investigar a mayor profundidad puede verse [Bea09; Pil09]

3.8 ¿I Python que hace?

El título de esta sección tiene un grave error ortográfico no se escribe I Python, se escribe ipython, o mas bien ipython3 ya que estamos utilizando Python3. Entonces, ¿ipython3 que hace? Bueno, hace todo lo anterior y algunas cosas mas. IPython es un muy completo sistema para computación interactiva, que utiliza desde luego a Python como lenguaje de programación[PG07; Ros13; Tea].

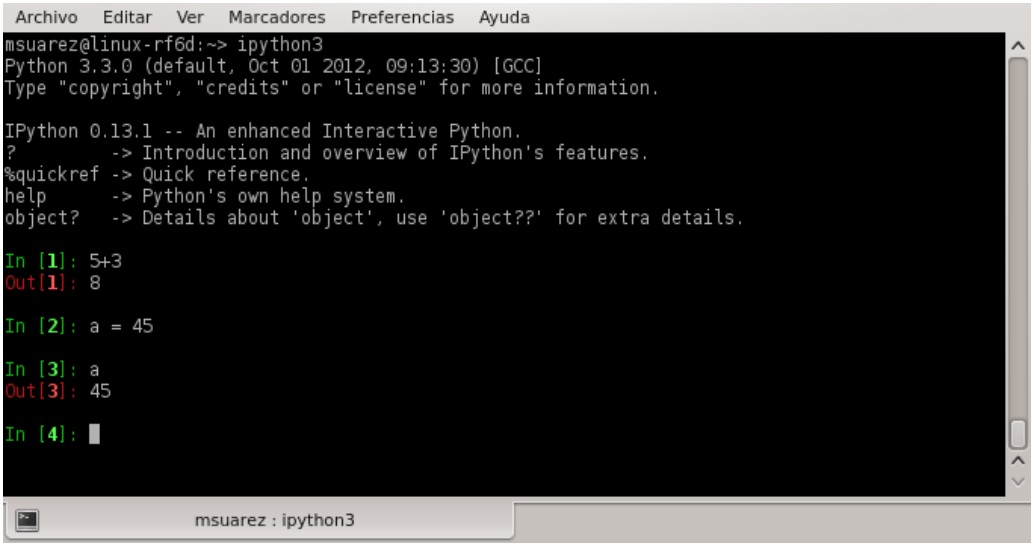


Figura 3.1: *ipython3 en un terminal. Se puede observar los números de lineas correspondientes a cada instrucción.*

Hasta ahora hemos estado trabajando en un terminal en el cual invocamos al interprete interactivo de Python tecleando python3.3, tal como se explicó en la sección 1.3. Si en vez de esto usamos el comando[5] ipython3, obtenemos adicionalmente una serie de ventajas. Sin llegar a ser exhaustivos vamos a mencionar algunas de ellas a continuación, pero antes echemos un vistazo a la figura 3.1. Luego de introducir el comando ipython3 en nuestro terminal habitual, aparece un mensaje donde aclara la versión de Python (en este caso 3.3.0), el compilador usado (GCC) y otras informaciones útiles como los comandos para la ayuda etc. Luego he realizado una serie de complicadísimas operaciones como por ejemplo sumar 5+3. Lo importante aquí es observar que la operación no está precedida por el habitual >>> sino por colorido In [1]: esto significa que se ha introducido un comando de "entrada", el resultado, Python procesa este comando y

[5]Por supuesto para que esto funcione ipython3 tiene que estar instalado en su computadora. Para ello véase *www.ipython.org*

como "salida" nos da el resultado en Out [1]: Ambas salida y entrada (In, Out) están numeradas para poder hacer referencia a ellas mas tarde.

Como es lógico IPython hereda todas las características del terminal "normal" entre ellas la complementación de sintaxis con la tecla tabulador. Supongamos que queremos escribir la declaración *while* empezamos por escribir la *w* y luego apretamos la tecla tabulador. El resultado es una lista de sentencias que todas comienzan con w[6].

```
In [1]: w
%who     %who_ls  %whos    while    with

In [1]: w
```

Si ahora le agregamos la "h" a while el número de posibilidades se reduce un poco, finalmente al colocar la "i" de while y apretar la tab o tabulador, se escribirá automáticamente la palabra completa. Esto es genial, sobretodo cuando no nos acordamos de de como se escribe la palabra o simplemente queremos teclear menos.

También heredamos otra característica útil como la historia de los sentencias introducidas en el programa. Simplemente utilizando la tecla "flecha hacia arriba" y "flecha hacia abajo podemos navegar por la historia de todas las instrucciones tecleadas. si hemos empezado a escribir antes de apretar las teclas con las flechas, solo se seleccionarán de toda la historia aquellas instrucciones que comiencen con lo que ya hemos escrito. ¿Cómodo no? Pero hay más, con la tecla "piso" (_) seguida de un número podemos recuperar la salida asociada a ese número, y con _i recuperamos la entrada.

```
In [1]: a = 45

In [2]: b = 126

In [3]: b/a
Out[3]: 2.8

In [4]: c=(b-a)/10

In [5]: a+b+c
Out[5]: 179.1

In [6]: _3
Out[6]: 2.8

In [7]: _i4
Out[7]: 'c=(b-a)/10'

In [8]:
```

[6]Como el lector puede ver por si mismo algunas de estas palabras empiezan por % y no por w, este símbolo indica que son palabras *mágicas* y no pertenecen a Python sino a IPython. Como este no es un tratado sobre IPython no se tocarán todas las palabras mágicas aquí, pero si la curiosidad lo carcome consulte la bibliografía.

Puede que no estemos interesados en que se muestre una salida en particular, ya sea porque es muy larga, muy fea, o simplemente porque no se nos da la gana, entonces colocamos ";" al final de la línea.

```
In [8]: c*2-b
Out[8]: -109.8

In [9]: c*2-b;

In [10]:
```

Adicionalmente IPython pose palabras mágicas, las palabras mágicas comienzan siempre con el caracter "*%*" y realizan cosas por nosotros como por arte de magia. Por ejemplo si queremos tener una lista de todos los comandos utilizados hasta ahora, es decir si queremos saber nuestra historia de comandos simplemente tipeamos *%hist*

```
In [11]: %hist
a = 45
b = 126
b/a
c=(b-a)/10
a+b+c
_3
_i4
c*2-b
c*2-b;
%hist

In [12]:
```

Suficiente, hacer una lista de de todas la capacidades que ofrece IPython no solo es aburrido sino que tendríamos que mencionar cosas que aún no hemos estudiado. Utilizaremos IPython desde ahora en adelante e iremos aprendiendo sobre la marcha.

3.9 ¿Me lo puede volver a repetir por favor?

Como ya se ha mencionado, repetir cosas es lo mejor que hace una computadora, por eso todos los lenguajes tienen varias formas para repetir instrucciones y Python no es una excepción. Ahora que sabemos que son las secuencias debemos agregar que son iterables, es decir podemos accesar a cada uno de los miembros que conforman una secuencia incrementando su índice de uno en uno. Python tiene una forma muy cómoda y eficiente de hacer esto con ayuda de la sentencia *for*.

```
In [1]: cad1 = "El contrasentido de la vida"

In [2]: for c in cad1:
   ...:     print(c)
   ...:
```

```
E
l

c
o
n
t
r
a
s
e
n
t
i
d
o

d
e

l
a

v
i
d
a

In [3]:
```

En la instrucción 1 creamos una cadena, en la instrucción 2 se encuentra el ciclo *for*, su sintaxis es:

for variable *in* objeto_iterable:
instrucciones

Lo que hizo el ciclo *for* fue tomar una variable *c* y asignarle consecutivamente los caracteres de la cadena *cad1[0], cad1[1], cad1[2], ...* Para cada iteración le asigna un nuevo caracter y lo imprime, el incremento en el índice de la cadena se realiza automáticamente. Si no nos gusta que los caracteres se impriman verticalmente, podemos intentar la siguiente modificación para que *print()* no imprima al final el comando para una nueva línea (\n), sino mas bien que no imprima nada al final (“”)

```
In [3]: for c in cad1:
   ...:     print(c, end="")
   ...:
El contrasentido de la vida
In [4]:
```

También en vez de que no escribiera nada al final le pudimos haber pedido que escribiera Cualquier cosa.

```
In [1]: cad2 = "Otra cadena"

In [2]: for c in cad2:
   ...:     print(c, end="Cualquier cosa ")
   ...:
OCualquier cosa tCualquier cosa rCualquier cosa aCualquier cosa  C
ualquier cosa cCualquier cosa aCualquier cosa dCualquier cosa eCua
lquier cosa nCualquier cosa aCualquier cosa
In [3]:
```

Ahora un ejemplo con listas en vez de cadenas. Aquí la variable es *animal* y la secuencia una lista llamada *L1*.

```
In [5]: L1 = ["gato", "perro", "mariposa", "elefante"]

In [6]: for anim in L1:
  ...:    print("El animal ", anim, "tiene ",len(anim)," letras.")
  ...:
El animal  gato tiene  4  letras.
El animal  perro tiene  5  letras.
El animal  mariposa tiene  8  letras.
El animal  elefante tiene  8  letras.

In [7]:
```

Observemos como en la función *print()* se han mezclado las variables como por ejemplo *animal* con cadenas literales como *"tiene "* separándolas por comas para que se puedan imprimir en pantalla correctamente.

Iterar directamente sobre "objetos iterables" como las secuencias es muy práctico, pero no siempre queremos hacer esto, por ejemplo cuando queremos modificar los valores de la propia lista sobre la que estamos iterando. Para estos casos tenemos la función *range()* . Esta función genera una secuencia progresiva de enteros y su sintaxis es la que sigue ***range(*** *inicio, fin, incremento)*, es decir la secuencia comienza en *inicio*, termina en *fin* y los enteros se incrementan con *incremento*. ¿Fácil no? Un ejemplo: *range(15, 23, 3)* genera la siguiente secuencia de números 15 18 21. Si omitimos especificar el *incremento*, se toma 1 como incremento por defecto, es decir *range(15, 23)* nos da 15 16 17 18 19 20 21 22. Si en cambio omitimos *inicio* se comienza en 0, por ejemplo *range(4)* nos da la secuencia 0 1 2 3. Notemos que el 4 no aparece, así como tampoco lo hace el 23 en la secuencia anterior.

Hasta Python 2.7 inclusive, la función *range()* devolvía como secuencia una lista, a partir de Python 3.0 genera un *iterable* o *iterador*, sobre este tipo de objetos se tratará mas adelante.

Volviendo a nuestro querido *for*, observemos el siguiente trozo de código

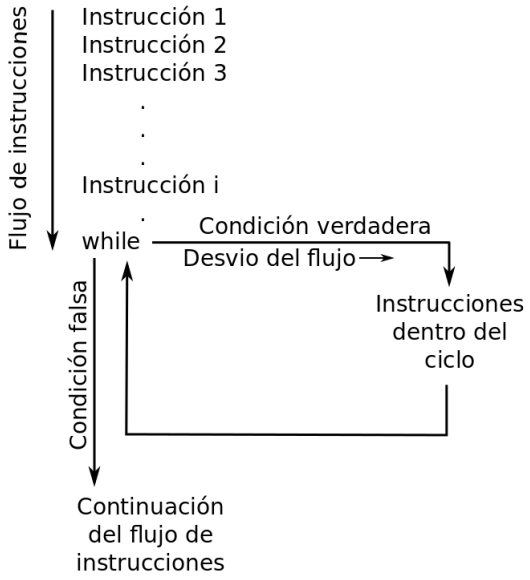
```
In [1]: L1 = [0, 1, 2, 3, 4]
```

```
In [2]: for i in range(5):
   ...:     L1[i] = L1[i]*L1[i]
   ...:

In [3]: L1
Out[3]: [0, 1, 4, 9, 16]

In [4]:
```

En el ejemplo anterior se muestra la utilización de *range* en la sentencia *for*.

3.10 Todo bajo control

Supongamos que creamos un programa escribiendo una sentencia en cada línea, línea tras línea, y el interprete de Python las va ejecutando una tras otra en el orden en que van apareciendo. Se forma así una especie de *flujo de ejecuciones* secuencial. Esto es así hasta que en el código aparezca un *while* o un *if*, cuando esto ocurre el flujo deja de ser "secuencial", se desvía para ejecutar otras acciones, se salta unas dependiendo de si una condición es cierta o falsa, o se retuerce sobre si mismo creando un bucle. Todo esto bajo el control del programador, bueno, cuando el programador sabe lo que hace.



A este tipo de acción se le denomina frecuentemente *control de dujo*, y a las estructuras como *while*, *if* y *for...in* se las denomina *estructuras de control* . A veces es útil

representar los flujos en de un programa o de cualquier proceso gráficamente, a esto se le llama *diagrama de flujo* .

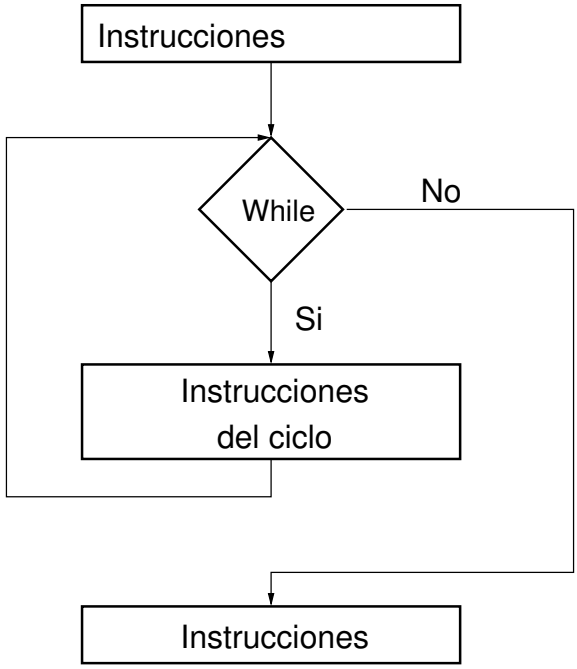


Figura 3.2: *Diagrama de Flujo mostrando un ciclo while*

En esta sección se mostrarán unas cuantas estructuras de control adicionales. La primera que consideraremos es la declaración *break* , que se usa para romper bucles. Ilustraremos su uso con un ejemplo, supongamos que queremos comprobar si los números que se encuentran en un lista son todos impares. Para resolver esto hacemos un ciclo sobre todos los números de la lista y los revisamos uno por uno, pero basta que uno solo sea par para darse por satisfecho y parar la búsqueda. A continuación se presenta el código que ilustra esto.

```
In [1]: lista = [3, 5, 23, 4, 11, 55, 33, 71]

In [2]: for n in lista:
   ...:     if (n % 2) == 0:
   ...:         print("Se encontró un número par en la lista")
   ...:         break
   ...:     else:
   ...:         print("Vamos bien, el número es impar")
   ...: print("El programa ha terminado")
   ...:
Vamos bien, el número es impar
Vamos bien, el número es impar
Vamos bien, el número es impar
Se encontró un número par en la lista
El programa ha terminado
```

```
In [3]:
```

Si el número es par, el resto de la división entera es cero, esto se codifica con la instrucción *if (n% 2) == 0:*, cuando esta condición es cierta se ejecuta la sentencia *print("Se encontró un número par en la lista")* y luego se ejecuta *break*. La instrucción *break* nos saca del bucle y sigue con la ejecución del programa. La sentencia *break* es útil pero no hay que abusar de ella, al romper la continuidad del flujo tan violentamente el código puede resultar mas difícil de entender. Antes de usar una instrucción *break*, tómese un tiempito para pensar si no existe una solución mejor. Como uno anda siempre con muy poco tiempo los *break* aparecen por todas partes. Como en el ejemplo que sigue.

■ **Ejemplo 3.5** Indique cual de los primeros diez números enteros es primo y cual no.

```
 In [1]: for n in range(2, 10):
   ...:     for i in range(2, n):
   ...:         if (n % i) == 0:
   ...:             print(n, " es igual a ", i, "*", n/i)
   ...:             break
   ...:     else:
   ...:         print(n, " es un número primo")
   ...:
2  es un número primo
3  es un número primo
4  es igual a  2 * 2.0
5  es un número primo
6  es igual a  2 * 3.0
7  es un número primo
8  es igual a  2 * 4.0
9  es igual a  3 * 3.0

In [2]:
```

En este ejemplo al principio nos encontramos con dos ciclos *for...* anidados (uno dentro del otro). En el primero de ellos se utiliza un iterador entre 2 y 10 , por lo tanto la variable n va tomando sucesivamente los valores 2, 3, 4, ... hasta legar a 9. El segundo ciclo comienza igualmente en 2, pero el iterador termina en el valor que en ese momento tenga la variable n, por ejemplo supongamos que tenemos un valor de $n = 5$, entonces i tomará sucesivamente los valores 2, 3, 4. Luego en la sentencia *if* se evalúa si el resto de la división entera entre n e i es igual a cero o no. Si es igual a cero significa que n es divisible entre i y por lo tanto n no puede ser primo. La sentencia *break* nos saca del ciclo interno (*for i in range(2, n):*) pero no del externo. Véase la nueva forma de utilizar *else*, aquí la sentencia *else* no está asociada al *if* sino como alternativa al ciclo *for...in* interno. Si el ciclo termina correctamente, la sentencia *else* se ejecuta, en cambio si el ciclo se interrumpe por alguna causa (en este caso debido al *break*) la sentencia *else* no se ejecutará. Véase el siguiente código para asegurase de que se entendió bien el uso de *else*.

```
 In [3]: for n in range(2, 10):
    ...:     for i in range(2, n):
    ...:         if (n % i) == 0:
    ...:             print(n, " es igual a ", i, "*", n/i)
    ...:             break
    ...:     print(n, " es un número primo")
    ...:
2  es un número primo
3  es un número primo
4  es igual a  2 * 2.0
4  es un número primo
5  es un número primo
6  es igual a  2 * 3.0
6  es un número primo
7  es un número primo
8  es igual a  2 * 4.0
8  es un número primo
9  es igual a  3 * 3.0
9  es un número primo

In [4]:
```

■

Del ejemplo anterior se deduce que la forma genérica del ciclo *for...in* es
for *variable* **in** *iterador*:
instrucciones dentro del bucle
else:
instrucciones que solo se ejecutan
si el ciclo for...in *termina abruptamente*

lo mismo ocurre para el ciclo *while*

while *condición*:
instrucciones dentro del bucle
else:
instrucciones que solo se ejecutan
si el ciclo while *termina abruptamente*

Otra nueva estructura de control es *continue* . Esta declaración interrumpe el ciclo pero no se sale de el, sinó que continúa con la siguiente iteración.

```
In [1]: cont = 0

In [2]: while cont < 10:
    ...:     cont += 1
    ...:     if cont == 5:
    ...:         continue
```

```
   ...:     print(cont, end=" ")
   ...:
1 2 3 4 6 7 8 9 10
In [3]:
```

es decir si la variable vale 5, se ejecuta la declaración *if* que emplea a *continue* para que el ciclo continúe sin ejecutar la función *print()*, por eso la sucesión de números salta de 4 a 6.

Finalmente mencionaremos la sentencia *pass*, su utilidad consiste en no hacer absolutamente nada, simplemente existe. Muchos lectores ingenuos pensaran que esto no es útil, pero aveces no hacer nada es lo mejor que se puede hacer, el autor de este libro lo hace todo el tiempo.

3.11 Ejemplos y ejercicios

▪ **Ejemplo 3.6 — Factorial.** Uno de los cálculos más fáciles que se pueden hacer con ciclos es el cálculo del factorial de un número. A continuación se muestra una de las múltiples posibles soluciones al cálculo del factorial del número N.

```
In [1]: N = 12

In [2]: result = 1

In [3]: while N>0:
   ...:     result = N*result
   ...:     N -= 1
   ...:

In [4]: result
Out[4]: 479001600

In [5]:
```

▪

▪ **Ejemplo 3.7 — Preguntas trascendentales.** Supongamos que tememos que resolver el problema clásico (mecánica clásica) de un cuerpo que cae libremente bajo la acción de la gravedad en presencia de la fuerza de resistencia que ejerce el aire. Por ejemplo una persona que cae de un avión, bueno pongámosle a dicha persona un paracaídas. Teniendo en cuenta las leyes de Newton y la resistencia del aire la ecuación que nos indica a que velocidad el paracaidista se aproxima al suelo[Gre06] es:

$$v = \frac{g}{k}(1 - e^{-kt})$$

donde v, es la velocidad, g es la aceleración de la gravedad, t es le tiempo y k es una constante positiva que representa la resistencia del aire. En esta constante influyen factores muy difíciles de precisar como la forma del objeto que cae, la densidad del aire, etc. Si lo que queremos es encontrar a que velocidad el paracaidista se estrella

contra el suelo, la ecuación anterior no representa ningún problema. Pero si estamos interesados en diseñar un paracaídas que frene al paracaidista a una velocidad razonable en determinado lapso de tiempo, entonces estamos interesados mas bien en hallar el valor de k para poder determinar que han grande debe ser el paracaídas, etc. El problema es que esto no se puede hacer, no hay manera de despejar k de la ecuación anterior.

Ecuaciones de este tipo aparecen con relativa frecuencia, no se puede resolver la ecuación analíticamente por lo tanto necesitamos recurrir al cálculo numérico. Como ejemplo resolveremos un problema un poco más estilizado desde el punto de vista matemático, pero que contiene todas las características de las ecuaciones que no tienen solución analítica. Supongamos que queremos encontrar el punto donde una recta se encuentra con la función coseno. Es decir, queremos encontrar el punto x, $x > 0$ tal que $\cos(x) = x$, los matemáticos hace tiempo han estado dándole vueltas a este clase de problemas y decidieron llamarlas *ecuaciones trascendentes* tal vez porque están mas allá de una solución analítica. En primer lugar empezaremos con un truquito clave a la hora de resolver ecuaciones numéricamente que consiste en convertir el problema original en un problema de obtención de raíces. Efectivamente, observemos que hallar la solución a $\cos(x) = x$ es equivalente a hallar la raíz de la ecuación $\cos(x) - x = 0$. Existe una gran variedad de estrategias para encontrar numéricamente las raíces de una ecuación, como se puede constatar en cualquier libro de cálculo numérico, a continuación utilizaremos uno de los métodos mas simples (aunque no el más eficiente), se trata del *método de bisección*. En este método comenzamos con un intervalo $[a,b]$ dentro del cual estamos *seguros* que se encuentra una sola raíz. ¡Un momento! Si justamente no sabemos donde está la raíz, ¿Cómo vamos a saber que intervalo escoger? ¡¿He...?! ¡Dígame, dígame...! ¡Cómo sabemos? Muy fácil, no sabemos. Pero en cambio podemos encender la computadora, utilizar el graficador de su preferencia [7] y obtener una gráfica como la de la figura 3.3 en ella podemos ver que el valor de x buscado está alrededor de 7.5, lo que indica que usar un intervalo igual a $[0.1\ ,\ 1.1]$ es mas que seguro.

Luego de escogido el intervalo inicial apropiado, lo dividimos a la mitad (claro es el método de "bisección), digamos que este nuevo punto se llama c; $c = (a+b)/2$. La pregunta ahora es, ¿La raíz se encuentra en el intervalo $[a,c]$ o en el $[c,b]$?. No, no volvemos a graficar, así también funcionaría, pero para la mayoría de las situaciones de interés práctico encontrar el resultado con la precisión requerida significaría repetir todo el procedimiento una enorme cantidad de veces y esto es muy tedioso. En cambio para saber en que intervalo se encuentra la raíz recordemos que la función cambia de signo cuando pasa por la raíz. Supongamos que la raíz cae en el intervalo $[c,b]$, esto significa que $f(c)$ y $f(b)$ deben tener signos opuestos, así que $f(c) * f(b) < 0$, por el contrario $f(a)$ y $f(c)$ deben tener el mismo signo, entonces $f(a) * f(c) > 0$, de esta forma encontramos en cual intervalo se esconde la raíz, la figura 3.4 puede sernos de ayuda para entender el procedimiento.

Seguidamente volvemos a dividir el intervalo de interés en dos y llamamos al nuevo punto d; $d = (c+b)/2$ repetimos todo el proceso para determinar si la raíz queda en $[c,d]$ o en $[d,b]$ y seguimos así hasta que se nos acaben las letras de abecedario. Aunque tal vez sea mejor usar algún criterio de parada, como la exactitud del resultado. Si estamos interesados en obtener la raíz x con una exactitud de 4 cifras decimales,

[7] Si aún no tiene un graficador de su preferencia como por ejemplo *gnuplot* que es excelente, no importa porque mas adelante aprenderemos a graficar utilizando Python

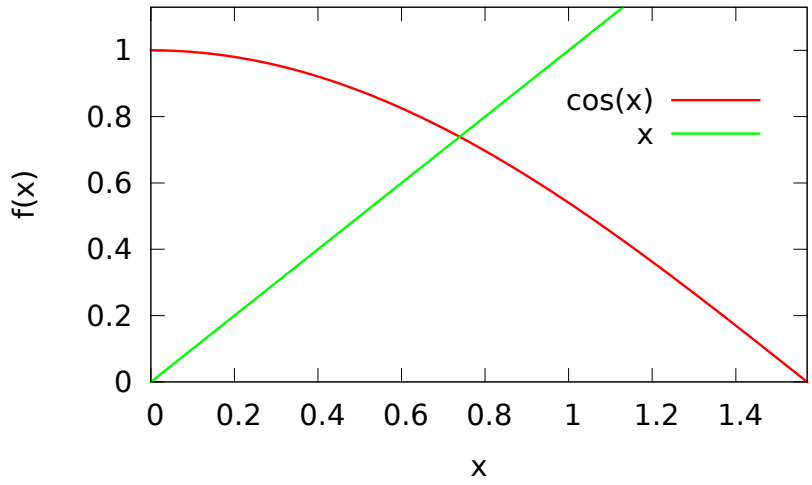


Figura 3.3: *En la gráfica se aprecia la intersección del la función $cos(x)$ con la función x en el intervalo $[0, \pi/2]$.*

nuestra condición de parada podría ser reportar el resultado cuando la diferencia entre los extremos del intervalo sea menor que 0.00001. Es decir si $(a-b) < 0.00001$ y x está entre a y b entonces seguramente cumple con la condición exigida.

```
In [1]: import math as ma

In [2]: a = 0.1

In [3]: b = 1.1

In [4]: ancho = b-a

In [5]: epsilon = 0.00001

In [6]: while ancho>epsilon:
   ...:     x = (a+b)/2
   ...:     fx = ma.cos(x)-x
   ...:     fa = ma.cos(a)-a
   ...:     fb = ma.cos(b)-b
   ...:     if (fx*fa) < 0:
   ...:         b = x
   ...:     else:
   ...:         a = x
   ...:     ancho = b-a
   ...:     print(x)
   ...:
0.6000000000000001
0.8500000000000001
0.7250000000000001
```

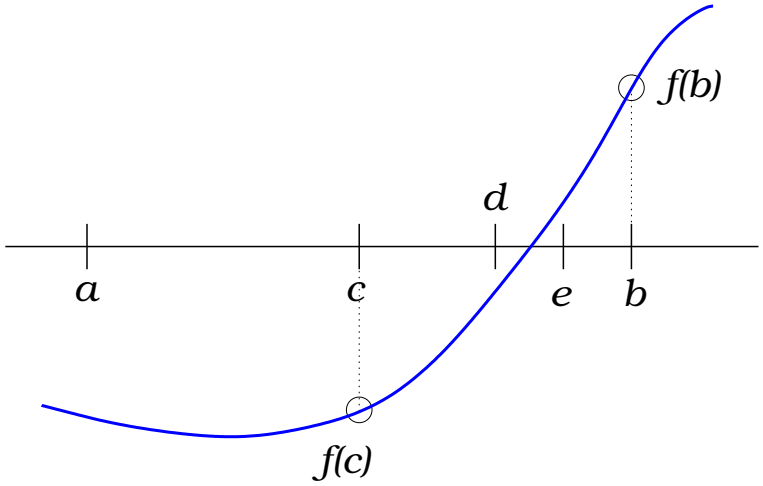


Figura 3.4: *En la figura se ilustra el método de la bisección. Se comienza con el intervalo* $[a,b]$ *que se divide a la mitad para obtener el punto c, véase que* $f(c)$ *y* $f(b)$ *tienen signos diferentes ya que la raíz se encuentra dentro del intervalo* $[c,b]$*. Este intervalo a su vez se divide a la mitad para encontrar el punto d. Ahora el intervalo que contiene la raíz es el* $[d,b]$*, que nuevamente dividimos a la mitad y encontramos el punto e, este procedimiento se repite hasta que se cumpla la condición de parada.*

```
0.7875000000000001
0.7562500000000001
0.7406250000000001
0.7328125000000001
0.7367187500000001
0.7386718750000001
0.7396484375000001
0.7391601562500001
0.7389160156250001
0.7390380859375001
0.7390991210937501
0.7390686035156251
0.7390838623046876
0.7390914916992188

In [7]:
```

Expliquemos el código anterior que implementa el método de bisección. En la primera línea se importa el módulo matemático, en la segunda y tercera línea se asignan valores a las variables que fijan los extremos del intervalo. Luego hallamos el ancho el intervalo $(a-b)$ y la condición de parada $epsilon = 0.00001$ que utilizaremos en el ciclo while. Luego viene el ciclo while propiamente dicho, mientras que la condición $ancho > epsilon$ sea verdadera se ejecuta el ciclo. Dentro del ciclo la primera sentencia consiste en hallar el punto medio del intervalo, este punto será nuestra solución tentativa al problema, a medida que el ciclo se ejecuta x se irá aproximando cada vez mas a su valor verdadero. Luego calculamos el valor de la función para los puntos a, b, y x, el signo de

estos valores nos dice en cuál sub-intervalo se encuentra la raíz. Esto lo descubrimos con la sentencia *if*. Si $(fx * fa) < 0$ es verdadero entonces la raíz esta en el intervalo $[a,x]$ en este caso debemos pasar el valor de x a la variable b para que en el nuevo ciclo del bucle while podamos calcular x correctamente. En caso contrario (else) la raíz se debe encontrar en el intervalo $[x,b]$, por lo tanto el valor de la variable x se pasa a a. En este punto del programa tenemos un nuevo intervalo $[a,b]$ que tiene la mitad de la longitud que el original. A continuación debemos calcular el nuevo valor para el ancho del intervalo, como $b > a$ basta con hacer $ancho = b - a$ luego imprimimos el valor de x. La respuesta final al problema es $x = 0.73909 \pm 0.00001$ ∎

Los ciclos son ideales para encontrar el resultado de sumatorias y series, a continuación una serie de ejercicios que se pueden resolver con las herramientas aprendidas hasta ahora.

Ejercicio 3.1 — π. Una forma de aproximar el valor de π es

$$\sum_{n=1}^{\infty} \frac{1}{n^4} = \frac{\pi^4}{90}$$

Encuentre el valor de π para varios términos de la sumatoria y compárelo con el valor verdadero. ∎

Ejercicio 3.2 — e. El valor de e se puede hallar con la siguiente serie.

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \ldots$$

Encuentre el valor de e para varios términos de la sumatoria y compárelo con el valor verdadero. ∎

Ejercicio 3.3 — ln. El logaritmo natural de cualquier valor x se puede hallar mediante la serie.

$$\ln(x) = \sum_{n=0}^{\infty} \frac{(-1)^n (x-1)^{n+1}}{(n+1)}$$

Encuentre el logaritmo natural para diferentes valores de x. ∎

Ejercicio 3.4 — Anímese. Se acuerda de la serie de Taylor, bueno algunos de los anteriores ejercicios son expansiones en series de Taylor de funciones específicas, atrévase a hacerlo por si mimo. Tome una o varias funciones de su agrado, realice la expansión en series de Taylor y luego calcule el resultado para algún valor de x en especial. ∎

4 — El colorido mundo de Python

En este capítulo seguiremos profundizando en las herramientas que el mundo de Python nos pone a nuestra disposición para hacernos la vida más fácil, pero no se preocupe, una vez terminado este capítulo usted estará en capacidad de resolver problemas aún más complejos y de esta manera mantener el nivel de estrés en su medida habitual.

Abandonaremos por ahora el ámbito de la programación iterativa, no sólo debemos abordar otras áreas que Python nos ofrece, sino que además es más fácil al escribir un libro, representar el código como si estuviese escrito en un editor y no en un terminal. Y al autor a diferencia del resto del mundo, le gustan las cosas fáciles, pero nos estamos adelantando, en la siguiente sección se hablará de Python y los editores.

4.1 Guiones y Programas

Lo siento mucho pero a pesar de lo que se diga por ahí, a mi entender Python no es un lenguaje de guiones (en ingles script) como si lo puede ser el JavaScript, o el lenguaje de bash[1] por ejemplo. Es cierto que Python es interpretado, es cierto que Python puede interactuar (notablemente bien) con el sistema operativo, etc; pero Python es mucho mas que eso. Claro, Python *también* es un lenguaje de guiones. Frecuentemente los guiones son programas simples que generalmente se ejecutan desde un terminal, pero esto se puede aplicar igualmente a los programas "normales" y vice versa los programas complejos también pueden ser guiones (como algunos script hechos en bash que he tenido el desagrado de ver). Desde ahora en adelante no se harán ningún tipo de distinción entre guiones y programas, todos serán programas escritos con ayuda de un editor y ejecutados en un terminal.

Para programar se debe usar un *editor*, no un *procesador de textos*. Los procesadores de textos o palabras al estilo *Word* o *LibreOffice Writer* se diferencian de los editores en que estos últimos escriben sólo texto, sin formato y sin imágenes, es decir no tienen

[1] Desde el principio hemos utilizado terminales o consolas, si se utiliza Unix, Linux o Mac OS X, se está usando un tipo de terminal llamado Bash. Bash también es el lenguaje de programación que utiliza este tipo de terminales para hacer guiones y poder programarlos[Ram].

diagramación, mientras que los procesadores de texto si la tienen. A pesar de que los editores guardan lo escrito como *texto plano* (es decir sin florituras ni diagramación) no necesariamente lo escrito con un editor es entendido correctamente por un editor distinto, sobretodo si los textos fueron escritos en sistemas operativos diferentes. El problema radica en la codificación, es decir la manera en como los caracteres que vemos en la pantalla son traducidos por el editor a ceros y unos, que es lo que finalmente entiende la computadora. En los actuales momentos existe una gran variedad de codificadores debido fundamentalmente a que los distintos idiomas utilizan diferente conjunto de caracteres, esto puede ser un gran problema para desarrolladores de páginas web, donde los usuarios pueden ser chinos como rusos o ingleses. En cuanto a programación se refiere, el más usado conjunto de caracteres son los del código ASCII[2], este tipo de código solo entiende los caracteres del diccionario ingles, los siento nada de eñes ni acentos. No se podía escribir palabras como *año* ni siquiera los nombres de físicos famosos como Schrödinger. Desde luego esto originó una enorme cantidad de codificaciones diferentes para poder incluir todos estos caracteres extraños, afortunadamente en la actualidad existe algo llamado UTF-8[3] que cada día gana mas y mas aceptación. La codificación UTF-8 acepta todos los caracteres de todos los idiomas del planeta, además de símbolos como !$% , etc. Hasta la versión Python 2.7 si uno quería usar la codificación UTF-8 tenía que especificarlo explicitamente ya que Python como todos los demás lenguajes empezó entendiendo solo caracteres ASCII, pero a partir de Python 3.0 utiliza UTF-8 por defecto. Por lo tanto lo mas razonable para mantener compatibilidad entre diferentes editores e incluso diferentes sistemas operativos es asegurarse de que nuestro editor está usando caracteres codificados en UTF-8.

A continuación debemos encontrar un editor de nuestra preferencia el autor ha probado Emacs, porque es un fanático; vim, porque no es tan fanático; kate, porque es un editor hecho pensando en los programadores para trabajar con el escritorio KDE que me agrada enormemente (lo siento por la gente de Gnome); gedit, porque es un editor hecho pensando en los programadores para trabajar con el escritorio Gnome que me agrada enormemente (lo siento por la gente de KDE); Anjuta, un proyecto interesantísimo que provee un entorno de desarrollo integrado muy poderoso (lo siento por la gente de Qt); Kdevelop, un proyecto interesantísimo que provee un entorno de desarrollo integrado muy poderoso (lo siento por la gente de GTK); Geany, me parece genial, al igual que Bear, solo que este último es mas incipiente, de Spyder estoy deseoso de volver a usarlo ahora que también tiene soporte para Python 3; lo mismo me pasa con eric5; Eclipse, esta en la lista de editores para probar, al igual que tantos otros, pero no voy a aburrir a mi ya de por si escuálida audiencia con una lista exhaustiva. El lector podrá por si mismo tomarse los próximos seis meses probando editores al azar hasta encontrar el que le guste. Una vez hecho esto fanatícese y diga abiertamente que todos los demás editores son una porquería, si no lo hace así, corre el riesgo de pasarse no seis meses sinó el resto de su vida probando editores, bueno en realidad no es tan malo, a mi me parece divertido.

Bien, supongamos que ya pasaron los seis meses ya encontró el editor de su agrado, jugó con el un poco, lo tiene configurado para trabajar en UTF-8 y ya está listo para su primer programa en Python. Como es su "primera vez" daremos las instrucciones paso a

[2] American Standard Code for Information Interchange

[3] UCS Transformation Format—8-bit

paso. Primero abra un archivo nuevo, luego guárdelo con el nombre de su preferencia, por ejemplo primerporgrama.py. Note que el nombre debe contener una sola palabra y debe terminar con la extensión *.py*, para la gente que usa el sistema operativo Windows ya están acostumbrados a esto de las extensiones, pero en linux la extensiones no existen, forman parte del nombre como cualquier otro conjunto de caracteres. Para linux el nombre pudiera haber sido primer.programa y de todas maneras hubiese funcionado. En cambio para los humanos es de gran ayuda ya que que nos dice de que se trata el texto en cuestión, si tiene la extensión *.py* no tenemos que abrirlo para ver que es lo que tiene dentro, ya sabemos que se trata de un programa escrito en lenguaje de programación Python. A los editores les pasa algo parecido, necesitan saber la extensión para poder usar todo el colorido poder de la *coloración de sintaxis*. Si la extensión es *.c*, entonces se trata de un programa escrito en el lenguaje *C* y el editor utiliza la coloración adecuada para ese lenguaje.

En los lenguajes compilados al programa escrito con el editor se le llama archivo *fuente*, este es el archivo que tiene que ser compilado para producir un *ejecutable*. En Python que es un lenguaje interpretado el ejecutable y el archivo fuente son la misma cosa, pero debemos decirle al sistema operativo, en nuestro caso linux [4], que tiene que tratar al archivo de texto como un ejecutable. Esto se hace de dos maneras, la primera es indicando explicitamente que el programa será ejecutado con el interprete de Python

```
msuarez@linux-rf6d:~> python3.3 NombrePrograma.py
```

al escribir esto en el terminal, el sistema operativo llama al interprete de Python para que ejecute al programa llamado NombrePrograma.py. Procedamos de inmediato, abra un editor de su agrado, escribamos al mundo nuestro primer mensaje.

```
print('Adiós, mundo cruel')
```

guardemos el programa con el nombre Test1.py, cuando esto sucede aparece la magia de la *coloración de sintaxis* y su programa lucirá algo así como

```
print('Adiós, mundo cruel')
```

lindo ¿no? En su editor no necesariamente aparecerán los mismos colores, ya que distintos editores utilizan coloración de sintaxis diferentes, pero que se pueden modificar según los gustos del usuario. La coloración de sintaxis no solamente le da colorido al programa, es útil ya que nos ayuda con una forma horrible de cometer errores de programación, los errores de sintaxis, supongamos que escribimos mal la palabra print, entonces se perderá la coloración

```
pront('Adiós, mundo cruel')
```

de esta forma sabemos que hemos escrito algo mal. Procedamos ahora a ejecutar el programa escribiendo en el terminal[5] el comando *python3.3 Test1.py*. En el terminal se debe ver lo siguiente:

```
msuarez@linux-rf6d:~> python3.3 Test1.py
Adiós, mundo cruel
msuarez@linux-rf6d:~>
```

[4] El procedimiento en otros sistemas operativos se encuentra fácilmente en Internet

[5] Asegúrese de que en el terminal, usted se encuentra en la misma carpeta que el archivo Test1.py.

Así de simple, ese ha sido su primer programa o guión.

La segunda manera de decirle a linux que el archivo de texto es un programa ejecutable, es convirtiéndolo en ejecutable. Para lograrlo debemos colocar en la primera línea la siguiente instrucción *#!/usr/bin/env python3.3*, este es un mensaje enviado al terminal. Cuando el terminal bash se encuentra con esta instrucción sobreentiende que se trata de un guión (¿quién dijo que Python no era un lenguaje de guiones?) y que debe ejecutar el comando *env* que se encuentra en la dirección /usr/bin/. El comando *env* localiza la dirección del interprete Python3.3 para que este a su vez pueda ejecutar el programa. Una vez hecho esto debemos cambiar la permisología al archivo para volverlo ejecutable con el comando *chmod*, así.

```
msuarez@linux-rf6d:~> chmod 755 Test2.py
```

Ahora ya se puede ejecutar el programa simplemente haciendo "clic sobre su icono, hágalo y compruebe que no pasa absolutamente nada. Bueno en realidad si pasa, lo que sucede es que el programa está diseñado para que escriba en un terminal con la función *print()*, si lo ejecuta con un clic (o con dos, dependiendo del manejador de ventanas que utilice) no hay terminal, y la orden de imprimir se pierde. Para evitar esto deberíamos realizar el programa de tal manera que genere el terminal donde va a escribir el mensaje, o programar una IGU (Interfaz Gráfica de Usuario) conocida también como GUI (del inglés Graphical User Interface) De estas dos últimas opciones nos encargaremos en otra oportunidad, por ahora ejecutaremos el programa desde el terminal, así no hay problema con la función *print()*. Suponiendo que nuestro terminal esta en la misma dirección que donde se encuentra el programa ejecutable escribimos en el terminal *./nombre_del_programa.py* que en este caso será simplemente *./ Test2.py*. Los caracteres *./* son los encargados de decirle al sistema operativo (Linux) que el ejecutable se encuentra en la dirección actual en la que estamos trabajando.

```
msuarez@linux-rf6d:~> ./Test2.py
Adiós, mundo cruel
msuarez@linux-rf6d:~>
```

De no ser así deberíamos escribir en su lugar la dirección completa, por ejemplo:
/home/mi_carpeta_personal/programas/Python/Test2.py

4.2 Usando programas

Volvamos al nuestro viejo programa de cinemática (ejemplo 3.3) Transcribiremos este programa íntegramente con ayuda de un editor de texto a un archivo al que le pondremos un nombre que nos recuerde que es lo que hace, por ejemplo *cinemática.py*. De esta manera cada vez que tenga que resolver un problema similar no se tiene que volver a escribir todo desde el principio. Claro es poco probable que tengamos que resolver siempre el mismo problema, las condiciones iniciales podrían ser diferentes, por ejemplo un ángulo de 45° en vez de 60°, en este caso solo tendríamos que modificar levemente el programa y luego correrlo. Pero podría ocurrir que queremos utilizar el programa después de varios meses y ya no nos acordamos exactamente donde hay que modificarlo para que haga lo que queremos. En este caso tendríamos que estudiar el

programa línea por línea hasta encontrar el punto exacto donde tenemos que hacer la modificación, este problema se vuelve particularmente importante cuando los programas son largos, de varias decenas o cientos de líneas de código. Aquí es donde se pone de manifiesto la utilidad de los comentarios y de una programación clara y ordenada. Por ejemplo, el inicio del programa que aparece en el ejemplo 3.3 se podría programar así.

```
import math as m #se importa el módulo matemático

g = 9.8 #aceleración de la gravedad

#Condiciones iniciales
vo = 6 #velocidad inicial
beta = m.radians(60) #ángulo en radianes

h = 1.0 #altura final

vyo = vo*m.sin(beta) #velocidad en el eje x

#cálculo del determinante de la ecuación de segundo grado
det = vyo**2 - 2*g*h
 .
 .
 .
```

De todas maneras tener que modificar el programa cada vez que se use no es práctico, además el que usa el programa (el *usuario*) no necesariamente es la misma persona que lo escribió, y posiblemente no sepa nada de programación y no tiene ningún interés en estar modificando programas. Lo mejor entonces es preguntar al usuario cuales son las condiciones iniciales. Eso se hace mediante la función *input()*

```
# A continuación se espera que el usuario introduzca un número
cad = input()
#input regresa una cadena que debemos convertir a número
num = float(cad)
# El dato introducido por el usuario se usará en una operación
a = num*10
print(a)
```

Al programa anterior le hemos dado el nombre de input1.py y lo probaremos con el número cinco, al correrlo la salida en pantalla es:

```
msuarez@linux-rf6d:~> python3.3 input1.py
5
50.0
msuarez@linux-rf6d:~>
```

Si se molestó en ejecutar el programa anterior habrá notado lo inadecuado que es,

una vez que se escribe el comando *python3.3 input1.py* en el terminal, no pasa nada, el programa espera que el usuario introduzca un número, nosotros sabemos que eso es lo que hay que hacer y lo hacemos (en el ejemplo se introduzco el número 5) pero un usuario "genérico" no sabrá que es lo que tiene que hacer, lo mejor es decírselo.

```
# A continuación se espera que el usuario introduzca un número
cad = input('introduzca un número')
#input regresa una cadena que debemos convertir a número
num = float(cad)
# El dato introducido por el usuario se usará en una operación
a = num*10
print(a)
```

En el terminal veremos lo siguiente:

```
msuarez@linux-rf6d:~> python3.3 input1.py
Intoduzca un número: 5
50.0
msuarez@linux-rf6d:~>
```

Ahora podremos reescribir el programa del ejemplo 3.3 pero indicándole al usuario que introduzca las condiciones iniciales. De esta manera no hay que modificar el programa cada vez que se ejecute (ver ejercicio 3.2). Pero podemos ir un poquito mas lejos, en el ejemplo de cinemática lo que en realidad se hace es hallar las raíces de un polinomio de segundo grado, este problema se presenta con tanta frecuencia no solo en cinemática sino en otras áreas de la ciencia y tecnología que vale la pena realizar un programa que lo resuelva. Veamos si puede entender lo hace el siguiente programa solo con los comentarios.

```
#!/usr/bin/env python3.3
#La línea anterior es para que podamos hacer un ejecutable.
'''Este programa halla las raíces de un polinomio de segundo
grado de la forma ax**2 + bx +c, el usuario debe introducir
los coeficientes a b y c'''

#importo el módulo matemático
import math as ma

#interacción con el usuario
print('Este programa halla las raíces de un polinomio de segundo grado')
aa = input('Introduzca el coeficiente a: ')
bb = input('Introduzca el coeficiente b: ')
cc = input('Introduzca el coeficiente c: ')

#realizo la conversión a punto flotante
a = float(aa)
```

```
b = float(bb)
c = float(cc)

#verifico que a sea diferente de cero
if( a==0.0 ):
        print("El valor de a no puede ser cero")
else:
        #calculo del determinante de la ecuación
        det = b*b - 4*a*c

        #compruebo que determinante no sea menor que cero
        if( det<0.0 ):
                print("La ecuación no tiene raíces reales")
        else:
                #calculo la primera raíz
                x1 = (-b + ma.sqrt(det))/(2*a)
                #calculo la segunda raíz
                x2 = (-b - ma.sqrt(det))/(2*a)
                #imprimo la respuesta
                print("Las raíces son:")
                print("x1 = ", x1, "  y x2 = ", x2)
```

No sea tímido ejecute el programa anterior y vea como funciona.

4.3 Una imagen vale mas que mil palabras

Una imagen vale mas que mil palabras, debe ser por eso que los cuadros son mas caros que los libros, pero además todo buen artículo científico suele tener alguna que otra gráfica [6] Por lo tanto ha llegado la hora de echarle un vistazo al poder gráfico de Python. Con Python se pueden realizar gráficas con calidad de publicación, es decir con una calidad lo suficientemente alta como para que las gráficas sirvan para aparecer en un artículo o libro. Empecemos por lo más básico, grafiquemos una serie de puntos, lo primero que debemos hacer es utilizar las librerías adecuadas importando el módulo *matplotlib*, luego generamos las respectivas coordenadas de los puntos a graficar, y luego utilizando las funciones de *matplotlib* las graficamos. A continuación un pequeño programa que muestra las capacidades gráficas básicas, que se ha guardado con el nombre de *grafica1.py*.

```
#!/usr/bin/env python3.3
'''
Nuestra primera gráfica
'''
```

[6] Existen notables excepciones. Laplace se preciaba de que la mecánica analítica por el desarrollada, era tan poderosa que para su libro no hizo falta ninguna figura. Afortunadamente los libros modernos de mecánica analítica si las tienen.

```
# se importa el módulo gráfico
import matplotlib.pyplot as plt

# se genera la gráfica
plt.plot([11, 20, 30, 35],[4, 9, 7, -2])
# se le pone un título
plt.title("Me parece que he visto una linda figura")
# se muestra la gráfica
plt.show()
```

Antes de correr este programa analizaremos su estructura. Las primeras líneas ya son conocidas, la primera es el comando para que el programa pueda ser utilizado como ejecutable si es eso lo que se desea, luego viene una cadena con tres comillas que se utiliza para documentación del programa y que en este caso tan trivial que resulta totalmente prescindible, sin embargo la colocamos igualmente por disciplina. Lo que sigue es la importación del modulo que trae las funciones para graficar, es decir importamos a *matplotlib*. Lo nuevo comienza a partir de este punto, *plot()* es la función encargada de generar la gráfica, recibe como argumentos las coordenadas de los puntos que se desea graficar. En nuestro caso se le pasaron dos listas la primera $[11, 20, 30, 35]$, contiene las coordenadas en el eje de las abscisas, mientras que la segunda $[4, 9, 7, -2]$ las coordenadas correspondientes al eje de las ordenadas. La siguiente línea se explica por si sola, es donde se especifica el título de la gráfica con la función *title()* . finalmente la función *show()* es la encargada de mostrar la gráfica en la pantalla. Como siempre este programa se ejecuta escribiendo en el terminal [7]

```
msuarez@linux-rf6d:~> python3.3 grafica1.py
msuarez@linux-rf6d:~>
```

y como por arte de magia parece en pantalla una gráfica como la que se muestra en la figura 4.1.

Lo primero que se nota es que en la figura no aparecen puntos, sino rectas azules que unen los lugares donde deberían aparecer los puntos. Este es el funcionamiento por defecto que utiliza matplotlib, pero lógicamente se puede cambiar, mas adelante veremos como. Debido a que no se le ha puesto un nombre especifico a la gráfica, matplotlib le pone uno por defecto, este es *figure 1* que se puede ver en la parte superior de la figura. En la parte inferior izquierda se aprecian una serie de siete iconos los cuales describiremos brevemente. Empezando de la derecha a izquierda nos encontramos en primer lugar con el ícono que nos permite guardar la figura. El formato por defecto es *png*, pero también se pueden guardar en muchos otros formatos como el postscript encapsulado *eps*, o también el formato *pdf*. El segundo ícono de derecha a izquierda configura la forma de la figura, haciendo clic en el, se crea una segunda ventana donde aparecen una serie de parámetros que se pueden variar a gusto del usuario, el botón *Reset* regresa dichos parámetros a sus valores originales, la mejor forma de entender que es lo que dichos parámetros hacen es aprobándolo, ¡pruébelo ahora!. Seguidamente, el

[7] Desde ahora en adelante no especificaremos como se ejecuta el programa en el terminal.

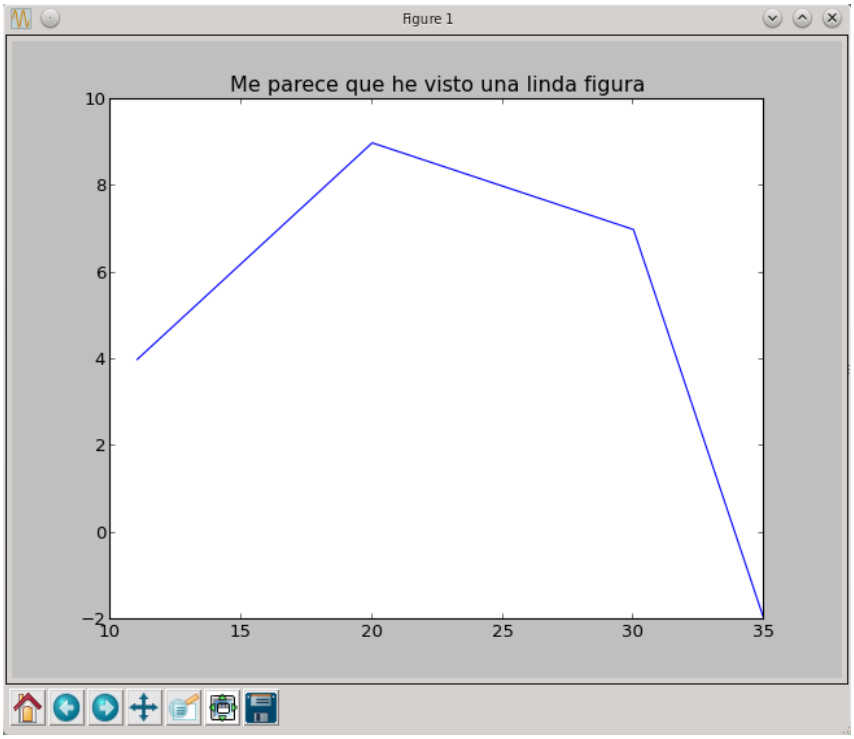


Figura 4.1: *Primera gráfica con matplotlib.*

tercer ícono de derecha a izquierda, nos permite ampliar un trozo de la figura a nuestro antojo, tantas veces como queramos. El Cuarto ícono realiza cambios sobre los ejes de coordenadas, si utilizamos el botón derecho del ratón se varía la escala, si en cambio utilizamos el botón izquierdo, variamos el rango. ¡compruébelo! Los dos siguientes iconos (que contienen flechas) nos permite "navegar" entre los cambios realizados a la figura. El último ícono de derecha a izquierda, o el primero de izquierda a derecha (con una figura de casita) deshace todos los cambios realizados a la figura y nos regresa a su apariencia original.

Modificaremos ahora el programa anterior haciendo que matplotlib dibuje solo puntos en vez de líneas rectas.

```
#!/usr/bin/env python3.3
'''
Nuestra segunda gráfica
'''

# se importa el módulo gráfico
import matplotlib.pyplot as plt

# se utiliza etiquetas para los ejes
plt.xlabel("Este es el eje de las abscisas")
plt.ylabel("Este es el eje de las ordenadas")
# se genera la gráfica
plt.plot([11, 20, 30, 35],[4, 9, 7, -2], 'ro')
```

```
# se muestra la gráfica
plt.show()
```

En el programa se han puesto etiquetas a los ejes con las punciones *xlabel()* y *ylabel()* y a la función *plot()* se le ha agregado la cadena *'ro'*, "r" significa *rojo* y la "o" significa que vamos a usar puntos. El resultado se muestra en la figura 4.2, desde ahora en adelante no se mostrará la ventana que genera matplotlib, sino solo la gráfica que es la que contiene la información relevante que deseamos mostrar.

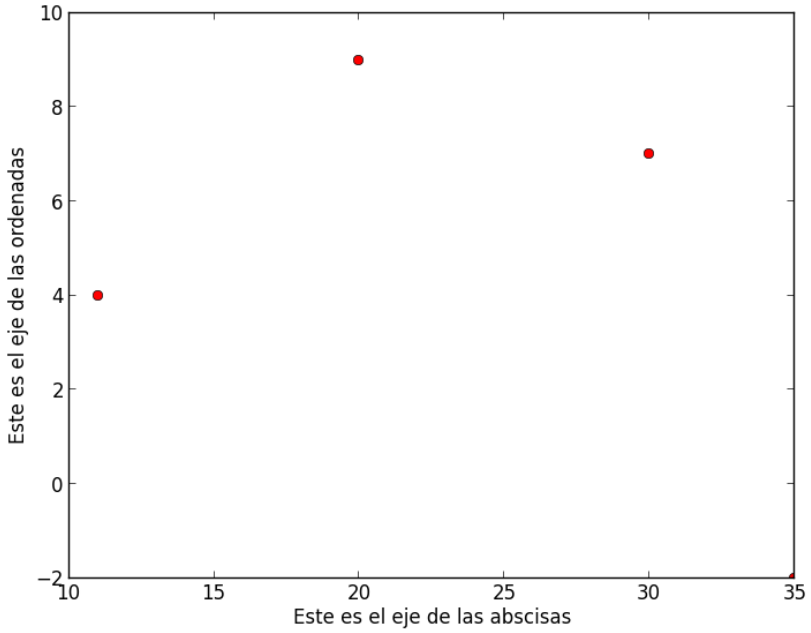


Figura 4.2: *Segunda gráfica con matplotlib. Aquí se muestra solo la gráfica en si misma, sin las decoraciones de la ventana que genera matplotlib.*

Observese que el último punto cuyas coordenadas son $x = 35; y = -2$ apenas es visible en la esquina inferior derecha de la gráfica. Esto lo podemos corregir inmediatamente utilizando el ícono que cambia el rango en el eje de las abscisas y/o ordenadas, pero también podemos especificar el rango directamente en el programa con las funciones *xlim()* y *ylim()*. Si quisieramos hacer esto en nuestro ejemplo se tendrían que agregar las siguientes líneas, en cualquier punto antes del *plt.show()*

```
plt.xlim(10,36)
plt.ylim(-3,10)
.
.
.
plt.show()
```

Estas funciones aceptan como argumentos dos números punto flotante que especifi-

can los límites del intervalo en las coordenadas respectivas.

¿Y si en vez de puntos rojos queremos cruces amarillas? También se puede, de hecho la combinación de colores y tipos es muy, muy grande, para una revisión exhaustiva se recomienda algún buen libro sobre matplotlib [Dal+09; Dev14; Tos09]. Aquí en cambio se mostrarán las principales características. En la tabla 4.1 se muestran algunos colores[8] en su forma abreviada colocando una sola letra, pero también se puede colocar la palabra completa en ingles.

Letra	Color
b	azul
c	cían
g	verde
k	negro
m	magenta
r	rojo
w	blanco
y	amarillo

Tabla 4.1: *Algunos colores aceptados por matplotlib en su forma abreviada*

Para las líneas tenemos cuatro posibilidades:

Abreviación	Tipo de línea
-	línea sólida
- -	línea cortada
-.	línea y punto
:	línea punteada

Tabla 4.2: *Diferentes tipos de línea en las gráficas*

y finalmente algunos estilos de puntos

Por supuesto que podemos colocar más de una curva en la misma gráfica, por ejemplo con el siguiente código:

```
#!/usr/bin/env python3.3
'''
Nuestra tercer gráfica
'''
# se importan los módulos necesarios
import matplotlib.pyplot as plt
import math as ma

# genero el eje x y las funciones a graficar
x = []
f = []
```

[8] Pero se pueden obtener muchos mas si se utiliza la codificación RGB

Abreviación	Estilo de punto
o	círculo
∧	triángulo hacia arriba
v	triángulo hacia abajo
s	cuadrado
D	diamante
+	cruz
x	x
*	estrella
.	punto
,	pixel

Tabla 4.3: *Estilos de puntos*

```
g = []
h = []
for i in range (0,180):
        z = (i/30)
        x.append(z)
        f.append(ma.exp(-z))
        g.append(ma.exp(-z)*ma.sin(2*ma.pi*z))
        h.append(-ma.exp(-z))
# etiquetas para los ejes
plt.xlabel("x")
# se genera la gráfica
plt.plot(x, f, label=r"$e^{-x}$")
plt.plot(x, g, label=r"$\sin(2\pi x)e^{-x}$")
plt.plot(x, h, label=r"$-e^{-x}$")
# genera la leyenda
plt.legend()
# se muestra la gráfica
plt.show()
```

Analizaremos solo las sentencias nuevas que aparecen en el programa anterior. Luego de la importación de los módulos se crean unas listas vacías con la variable *x* y las tres funciones que vamos a graficar *f*, *g* y *h*, en un ciclo *for...in* es donde se rellenarán la listas con los valores correspondientes usando la función *append()*. Como el iterador del ciclo *for...in* varía entre 0 y 180, entonces las curvas tendrán 180 puntos, es necesario una gran cantidad de puntos para que la curva se vea "suave" en la gráfica y no una colección de líneas rectas (véase el ejemplo 4.2). Si bien el iterador nos da los 180 puntos, es con la variable z (punto flotante) que calculamos los valores específicos de la coordenada. Con la sentencia *x.append(z)* construimos la lista que contiene la variable *x*, en las sentencias siguientes construimos de la misma manera las tres funciones. Observemos que en los argumentos de la función *append()* calculamos directamente las funciones que vamos

a graficar usando el módulo matemático *math*. Una vez construidas las listas con las variable y las funciones procedemos a graficar mediante *plt.plot()*, aquí aparecen varias cosas nuevas. En primer lugar en vez de usar las listas directamente, hemos utilizado el nombre asignado a esas listas, por ejemplo *plt.plot(x, f)* graficará la lista f en función de la lista x. Adicionalmente hemos añadido una leyenda con el comando *label="..."*, lo que se ponga dentro de las comillas será tomado como una cadena y se colocará textualmente en la leyenda, esto funciona muy bien la mayoría de las veces, pero no es muy útil si queremos representar fórmulas. Para representar fórmulas matemáticas como e^x, símbolos matemáticos como $\sum$, o constantes como π debemos realizar un truquito. El truco consiste en llamar al compilador de LaTeX[9] mediante r"..." colocando dentro de las comillas la formula escrita en LaTeX. Matplotlib tiene todo lo necesario para entender estas fórmulas sin problema, pero en cambio el programador debe conocer como escribir con LaTeX[MG04]. Ya se han añadido las leyendas con la sentencia *plt.plot()* pero para "crear" la leyenda y colocarla en la gráfica hace falta escribir la orden explicitamente con *plt.legend()*. El resultado se muestra en la figura 4.3. Observese que matplotlib gráfica las 3 curvas en colores diferentes de una manera automática sin necesidad de que lo especifiquemos explicitamente.

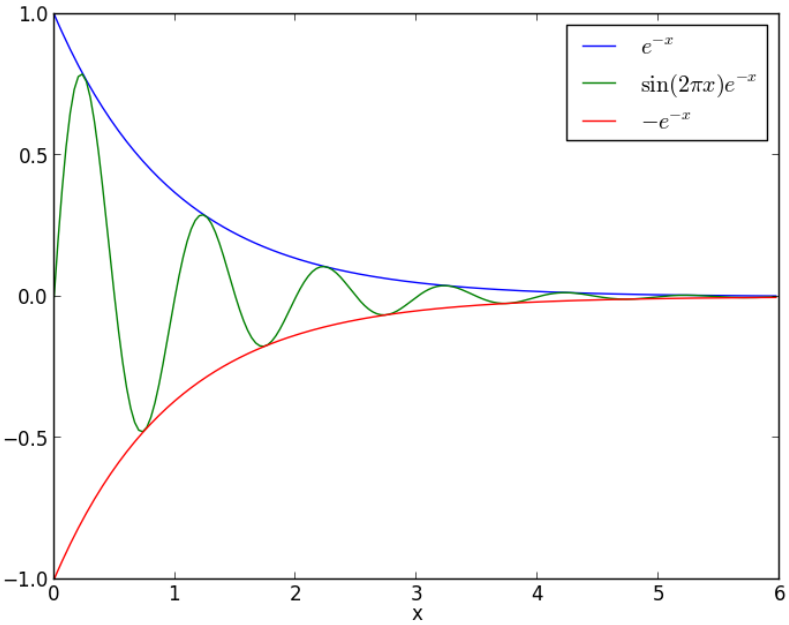


Figura 4.3

[9] LaTeXes un sistema de escritura de documentos de alta calidad tipográfica. Originalmente diseñado para documentos técnicos y científicos tiene un soporte maravilloso para escribir fórmulas, pero a evolucionado tanto que se puede escribir poesía tanto como un artículo científico. Este libro se ha escrito completamente usando LaTeX

4.4 Para entender Python es necesario saber leer y escribir

Frecuentemente técnicos, científicos e ingenieros, toman datos de campo o en el laboratorio, hacen experimentos, toman mediciones, etc. Todos esos datos son guardados minuciosamente en archivos digitales frecuentemente con varias copias para no correr riesgos. Luego por supuesto, los datos deben ser procesados para extraer información útil, Python es muy bueno para eso, pero antes hay que aprender a leer la información directamente de los archivos, y ya que estamos aprendiendo a leer, también vamos a aprender a escribir. Supongamos que queremos leer un archivo de texto que contiene datos, haciendo gala de una imaginación desbordante lo llamaremos *texto.dat*. Supongamos además que contiene una información sumamente importante como por ejemplo:

```
Este es un importantísimo
archivo de texto, que contiene
tanto palabras como números.
Aquí van los números:
10 20 30 40 50 60 70 80 90 100
Esta oración no indica el fin del archivo.
```

Lo primero que necesitamos para poder leer un archivo, es saber donde se encuentra[10], por el momento asumiremos que el archivo que se desea leer y el programa que crearemos para leerlo están en la misma carpeta. Las sentencias involucradas son simples, primero abrimos el archivo con la función *open()* .

```
open('texto.dat', encoding='utf-8')
```

Como se ve, la función *open()* acepta varios argumentos, el primero es el nombre del archivo que queremos abrir, el segundo es la codificación que se usó para guardarlo (ver 4.1). En Linux, Mac OS X y Unix los archivos se crean usando la codificación utf-8 (como debe ser) a no ser que se especifique lo contrario, pero en Microsoft Windows pudiese ser cp1252 (aunque puede cambiar según las versiones). Por lo tanto la codificación correcta es importante para hacer el programa portable de tal manera que funcione en varios sistemas operativos. Si la codificación no es la correcta se obtendrá un mensaje de error o los caracteres serán ilegibles. Hacer programas totalmente portables queda fuera del alcance de este libro, desde ahora en adelante supondremos que el archivo a ser leído está generado con la misma codificación que se utiliza para crear el programa, en este caso no es necesario aclarar la codificación usada. Si el archivo no existe se obtendrá un mensaje de error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'texto.dat'
```

Frecuentemente ocurre que el archivo que se desea leer si existe solo que no está en el mismo directorio que el programa que lo quiere leer, en este caso debemos proporcionar la dirección completa del archivo a ser leído junto con el nombre. Por ejemplo si el archivo se llama "texto.dat" y se encuentra en la dirección */home/msuarez/Python/Datos*, para abrirlo se debería escribir:

[10] A lo largo de todo el libro y en particular en esta sección se supondrá que el lector está familiarizado con la estructura de archivos, direcciones, etc, es decir sabe donde está parado.

```
arch = open("/home/msuarez/Python/Datos/texto.dat")
```

Por defecto la función *open()* abre el archivo en modo de lectura, pero también podríamos estar interesados en escribir, o los datos podrían ser binarios y no caracteres. El programador debe conocer estas alternativas de antemano y abrir el archivo en el modo correspondiente. Para abrir correctamente una archivo en modo lectura se debe escribir:

```
archivo = open("nombre del archivo", "r")
```

la "r" viene de "read" que naturalmente significa escribir, en el siguiente cuadro se enumeran los modos utilizados por la función *open()*.

Modo	Significado
r	Lectura
w	Crear un archivo y escribir en el
a	Adicionar datos al final de un archivo ya existente
r+	Lectura y escritura en un archivo ya existente
w+	Crea un archivo para la lectura y la escritura
a+	Lo mismo que 'w+' pero los datos son adicionados al final del archivo
rb	Lectura en archivo binario
wb	Crear un archivo binario y escribir en el
ab	Adicionar datos binarios al final de un archivo ya existente

Tabla 4.4: *Modos de apertura de archivos*

por ejemplo para escribir en un archivo usamos:

```
archivo = open("nombre del archivo", "w")
```

Notemos además que el *nombre del archivo* es una cadena de caracteres, lo cual nos permite usar una variable que la represente, por lo tanto otra manera de abrir el archivo es:

```
nom_archivo = "Texto.dat"
arch = open( nom_archivo, "r" )
```

es mas, no siempre se sabe con antelación el nombre del archivo que se va a leer, o tal vez estamos interesados en leer mas de un archivo, en estos casos es conveniente que sea el propio usuario del programa quien decida el nombre del archivo mientras el programa se ejecuta.

```
#Se pide al usuario que introduzca el nombre del archivo
nom_archivo =input("Introduzca el nombre del archivo a ser leído: ")
arch = open(nom_archivo,"r") # se abre el archivo en modo lectura
```

Ya hemos aprendido a abrir un archivo ahora vamos a leerlo, para eso utilizamos el hecho de que la función *open()* regresa un objeto tipo cadena

```
archivo = open("texto.dat", "r")
```

así se ha creado el objeto *archivo* puede ser usado para leer. Volvamos por un momento a la forma iterativa usando el terminal, para comprender mejor como funciona internamente el proceso de lectura.

```
>>> archivo = open("texto.dat", "r")
>>> archivo.read()
'Este es un importantísimo\narchivo de texto, que contiene\ntan
to palabras como números.\nAquí van los números:\n10 20 30 40 5
0 60 70 80 90 100\nEsta oración no indica el fin del archivo.'
>>>
```

Para leer el contenido del archivo se utilizó la función (o más precisamente el método asociado al objeto "archivo") *read()* , no parece de mucha ayuda, pero al menos sabemos que contiene el archivo. Observemos que *read()* en realidad arroja una sola cadena donde el salto de línea esta representado por el caracter "\n". Si por el contrario en vez de usar el Python en forma iterativa usamos un programa, para observar la salida producida por *read()*, necesitamos emplear la función *print()*, de lo contrario no se muestra salida alguna.

```
archivo = open("texto.dat", "r")
print(archivo.read())
```

cuya salida en pantalla gracias a la función *print()* será:

```
Este es un importantísimo
archivo de texto, que contiene
tanto palabras como números.
Aquí van los números:
10 20 30 40 50 60 70 80 90 100
Esta oración no indica el fin del archivo.
```

Notemos que ahora la salida se corresponde perfectamente con lo que está escrito en el archivo *texto.dat*, no aparecen por ningún lado los "\n" ya que la función *print()* se encarga de interpretar correctamente los saltos de línea al igual que todas las demás secuencias de escape. Claro que sería mucho mas útil si pudiésemos leer el archivo línea a línea y efectivamente si podemos.

```
>>>
>>> archivo = open("texto.dat", "r")
>>> archivo.readline()
'Este es un importantísimo\n'
>>>
```

Sin argumentos, la función *readline()* lee la primera línea incluido los caracteres que denotan el salto de línea. Si se quiere leer parte del renglón, se tiene que dar como argumento el tamaño en bytes, cada carater es un byte, incluyendo espacios en blanco

```
>>>
>>> archivo = open("texto.dat", "r")
>>> archivo.readline(6)
'Este e'
>>>
```

Por lo tanto si queremos leer completamente el archivo texto.dat renglón a renglón, debemos de escribir *readline()* seis veces, claro hay una forma mejor y es utilizando un ciclo. Pero antes de hacerlo con un ciclo *for...in* o *while* leamos el archivo utilizando *readline()* no seis sino nueve veces.

```
>>>
>>> archivo = open("texto.dat", "r")
>>> archivo.readline()
'Este es un importantísimo\n'
>>> archivo.readline()
'archivo de texto, que contiene\n'
>>> archivo.readline()
'tanto palabras como números.\n'
>>> archivo.readline()
'Aquí van los números:\n'
>>> archivo.readline()
'10 20 30 40 50 60 70 80 90 100\n'
>>> archivo.readline()
'Esta oracion no indica el fin del archivo.'
>>> archivo.readline()
''
>>> archivo.readline()
''
>>> archivo.readline()
''
>>>
```

Como era de esperar, cada vez que se ejecuta la sentencia *readline()* se lee un renglón completo. En el último renglón no aparece el salto de línea ya que el archivo original (*texto.dat*) tampoco lo tiene, pero lo interesante viene después de la última línea. Con cada nueva aplicación de la sentencia *readline()* hallamos siempre el mismo resultado, una cadena vacía (¡Cuidado!, no es un espacio en blanco). Esto es lo que verdaderamente indica que se ha llegado al final del archivo. A continuación en vez de usar la forma iterativa, vamos a crear un programa equivalente donde estamos obligados a usar la función *print()*, y de paso ilustraremos una característica que a veces causa un poco de confusión.

```
archivo = open("texto.dat", "r")
print(archivo.readline())
print(archivo.readline())
print(archivo.readline())
print(archivo.readline())
print(archivo.readline())
print(archivo.readline())
print(archivo.readline())
print(archivo.readline())
print(archivo.readline())
```

el resultado arrojado es:

```
Este es un importantísimo

archivo de texto, que contiene

tanto palabras como números.

Aquí van los números:

10 20 30 40 50 60 70 80 90 100

Esta oracion no indica el fin del archivo.



msuarez@linux-rf6d:~/Python/programas>
```

La función *print()* toma las cadenas vacías cuando se llega al final del archivo y por supuesto no imprime nada, de ahí los últimos renglones en blanco. Pero, ¿por qué aparece un renglón vacío intercalado entre las oraciones? La respuesta es debido a que *print()* automáticamente coloca al final un salto de línea, pero la línea leída por *readline()* ya tiene incorporado un salto de línea, por lo tanto ahora tenemos dos saltos de línea, y entonces se imprime un renglón en blanco. Por supuesto que hay una forma de evitar este comportamiento, esto ya se explicó en la sección 3.9, ahora veremos como leer línea a línea usando un ciclo. Si bien el ciclo *while* se puede usar para esta tarea (ver ejercicio) la forma más eficiente de leer un archivo es con el ciclo *for...in*. Los archivos abiertos con *open()* se convierten en objetos iterables, y de esa manera podemos iterar sobre las lineas para leer todo el archivo renglón a renglón.

```
archivo = open("texto.dat", "r")
for linea in archivo:
        print(linea)
```

observemos que no es necesario usar una función de lectura, simplemente imprimimos la linea iterada y eso es todo. La salida por pantalla de este programa se muestra a continuación.

```
 Este es un importantísimo

 archivo de texto, que contiene

 tanto palabras como números.

 Aquí van los números:

 10 20 30 40 50 60 70 80 90 100
```

```
Esta oracion no indica el fin del archivo.
msuarez@linux-rf6d:~/Documentos/Python/programas>
```

La iteración termina en el momento en que el ciclo llega al final del archivo cuando se obtiene una cadena vacía. Pero todavía tenemos el problema del renglón intercalado. Para evitarlo se le tiene que ordenar a *print()* que no incluya al salto de línea al final de la cadena de caracteres, como ya se ha visto esto se logra utilizando

```
archivo = open("texto.dat", "r")
for linea in archivo:
        print(linea, end=")
```

y la salida en pantalla será:

```
Este es un importantísimo
archivo de texto, que contiene
tanto palabras como números.
Aquí van los números:
10 20 30 40 50 60 70 80 90 100
Esta oración no indica el fin del archivo.msuarez@linux-rf6d:~>
```

Observemos el último renglón ¿Que ha pasado? La respuesta es que la última oración del archivo *texto.dat* no se terminó con un salto de línea y al usar la opción *end=''* se elimina el salto de línea que introduce la función *print()*, por lo tanto lo que vaya a ser escrito en el terminal después de la última oración (que en este caso es *msuarez@linux-rf6d:* >) se escribirá en el mismo renglón.

Ya que hemos aprendido a leer ahora aprendamos a escribir, para ello utilizaremos la función *open()* en modo de escritura, por ejemplo *open("nombre_arch", "w")*, esta sentencia crea un archivo llamado *nombre_arch*, pero si ya existe un archivo con ese nombre será borrado para colocar el nuevo contenido. Veamos el siguiente ejemplo.

```
archi_sal = open("Salida.dat", "w")
archi_sal.write("Un pequeño mensaje \nde dos renglones")
```

Si utilizamos un editor de texto para ver el contenido del archivo "Salida.dat", observamos lo siguiente

```
Un pequeño mensaje
de dos renglones
```

como vemos la función *write()* se comporta muy parecido a *print()* solo que *write()* se utiliza con archivos. Siempre hay que tener en cuenta el salto de línea, veamos que

```
archi_sal = open("Salida.dat", "w")
archi_sal.write("Un pequeño mensaje")
archi_sal.write("de dos renglones")
```

escribe en el archivo de salida

```
Un pequeño mensajede dos renglones
```

Muchas veces lo que necesitamos escribir son datos, estos pueden ser números enteros o punto flotante o tal vez estamos interesados en escribirlos en notación científica. Para ello las funciones *print()* y *write()* soportan salida formateada. La escritura con formato para un número entero es {:*a*d}.*format(m)* donde "*a*" el ancho en caracteres, "d" indica que el número ha de ser un entero y "*m*" es el número, por ejemplo:

```
>>> print("Este es un número entero{:1d}".format(5))
Este es un número entero5
>>>
```

El entero5 se ve francamente mal, es fácil corregirlo poniendo espacios en blanco entre la palabra y el número *entero {:1d}*, pero veamos lo siguiente

```
>>> print("Este es un número entero{:10d}".format(5))
Este es un número entero         5
>>>
```

si escribimos *{:10d}* se reservan 10 caracteres, pero el número cinco solo tiene un dígito, por lo tanto los otros nueve caracteres se rellenan con espacios en blanco. Si en vez de espacios en blanco queremos ceros simplemente hacemos.

```
>>> print("Este es un número entero {:010d}".format(5))
Este es un número entero 0000000005
>>>
```

Una forma mas elegante de reservar espacio es con el tabulador *t* que es ideal para hacer tablas.

```
>>> print("Dos números enteros\t{:2d}\t{:2d}".format(5,10))
Dos números enteros         5      10
>>>
```

también se pueden usar variables y expresiones matemáticas

```
>>> a = 2
>>> b = 3
>>> print(Dos números enteros\t{:1d}\t{:1d}".format(a+b,10*b))
Dos números enteros         5      30
```

observemos que a los números se les dio espacio para un solo caracter, pero 30 tiene dos dígitos y por lo tanto se requieren dos caracteres, a pesar de eso el resultado es el correcto.

Para un número punto flotante la forma general es {:*a.n*f}.*format(m)*, como antes "*a*" es el ancho en el cual se incluye el punto decimal, "*n*" es la cantidad de dígitos decimales y "*m*" el el número.

```
>>> print("Este es un número punto flotante {:4.2f}".format(1.23))
Este es un número punto flotante 1.23
>>>
```

Esta manera de escribir la salida formateada está disponible desde Python 2.7 en adelante, antes se usaba una forma muy parecida a la función *printf()* de C/C++. Python 3.x todavía acepta esta vieja forma de salida formateada (al estilo C) que aún es muy popular, el ejemplo anterior se vería de la siguiente manera.

```
>>> print("Este es un número punto flotante %4.2f" %(1.23))
Este es un número punto flotante 1.23
>>>
```

finalmente si queremos usar la notación científica:

```
>>> print("número en notación científica {:4.2e}".format(0.0058))
número en notación científica 5.80e-03
>>>
```

La función *print()* y la *write()* actúan de forma idéntica, solo que la primera imprime en pantalla, mientras la segunda escribe en un archivo. Por último para escribir al final de un archivo ya existente se usa:

```
archi_sal = open("texto.dat", "a")
archi_sal.write("Hola que tal.")
archi_sal.close()
```

Lo nuevo en el código anterior es la función *close()* como su nombre lo indica es la encargada de cerrar el archivo abierto con *open()*. No es necesario hacerlo ya que Python hace esta tarea por nosotros de forma automática una vez que el programa finaliza, pero es muy buena práctica de programación que una vez que el archivo abierto no se utilice mas, se cierre. Esto evita errores en programas largos y complejos y los hace más claros. Veamos como quedó el archivo *texto.dat* después de correr el código anterior.

```
Este es un importantísimo
archivo de texto, que contiene
tanto palabras como números.
Aquí van los números:
10 20 30 40 50 60 70 80 90 100
Esta oracion no indica el fin del archivo.Hola que tal.
```

efectivamente la oración “Hola que tal” se adicionó al final del archivo.

▪ Ejemplo 4.1 — Lectura y escritura. En el archivo *texto.dat* se encuentra un renglón que contiene números. Extraiga esos números realice un promedio y guarde el resultado en un archivo llamado *Salida.dat*

Respuesta: Para llevar a cabo esta tarea aplicaremos un algoritmo sencillo con los conocimientos previos aprendidos y una función nueva. En el archivo *texto.dat* se tienen cuatro renglones antes de que aparezcan los números. Toda esa información nos es inútil para nuestro propósito, así que debe ser leída y descartada. Esto lo lograremos con la aplicación cuatro veces de la función *redline()*. Cuando lleguemos al renglón en donde se encuentran los números, en vez de desechar la información la guardamos en una cadena de caracteres que llamaremos *cad*. Como ya hemos dicho en reiteradas oportunidades,

en Python todos son objetos, y las cadenas de caracteres no son la excepción, uno de sus métodos es la función *split()* . Esta función toma a la cadena y la separa por los espacios en blanco para crear una lista *lista = cad.split()*. En este caso la lista creada será *['10', '20', '30', '40', '50', '60', '70', '80', '90', '100']*. Como las listas son objetos iterables utilizaremos un ciclo *for...in* para realizar el promedio.

```
suma = 0.0
for num in lista:
        suma += float(num)
prom = suma/10
```

La función *split()* no convierte automáticamente los caracteres a números, por lo tanto esto lo debemos hacer explicitamente con *float(num)*. El siguente paso es guardar el resultado en el archivo *Salida.dat*, el programa completo es siguiente:.

```
entr = open("texto.dat", "r")
entr.readline()
entr.readline()
entr.readline()
entr.readline()
cad = entr.readline()
lista = cad.split()
entr.close()
suma = 0.0
for num in lista:
        suma += float(num)
prom = suma/10
sal = open("Salida.dat", "w")
sal.write("El promedio es: " + str(prom))
sal.close()
```

■

4.5 Hagamos un experimento

Tomemos un caso simple, se recogieron datos de un experimento y queremos graficarlos para ver si podemos encontrar alguna ley física que los explique. Como suele ocurrir las mediciones se han repetido varias veces para luego tomar el promedio y así disminuir el error. Supondremos tambien que el error de medicion para las variables involucradas es conocido y constante $\Delta x = 0.01$ y $\Delta y = 0.05$. Los datos fueron recogidos en forma de tabla y guardados en un archivo llamado *Experimento.dat* de la forma siguiente:

```
 x         y1       y2       y3       y4       y5
 0.05      3.4       1.3      1.8      0.2      4.6
 0.10      7.4      11.7     11.5      6.9      3.3
 0.15      4.9       8.6     11.5      4.5      5.8
 0.20      8.2      17.5     10.7     12.8     13.2
 0.25     13.8      17.3     15.3     17.8     16.6
```

0.30	13.3	16.4	21.1	21.5	20.2
0.35	23.4	24.7	25.7	25.7	21.6
0.40	28.8	25.9	26.5	22.7	22.9

En la taba anterior *x* representa la variable indepandiente, mientras *y1, y2, y3, y4, y5* son los resultados obtenidos depues de realizar cinco veces el mismo experimento. Para llevar a cabo nuestro proposito correremos el siguiente programa:

```
#!/usr/bin/env python3.3
'''
Aquí debe ir una documentación larga donde
explicitamente se diga lo que hace el programa.
'''
# se importan los módulos necesarios
import matplotlib.pyplot as plt

#Se pide al usuario que introduzca el nombre del archivo
#nom_archivo = input("Introduzca el nombre del archivo a ser leido: ")
nom_archivo = "Experimento.dat"
arch = open(nom_archivo,'r') # se habre el archivo en modo lectura


x = []
y = []
arch.readline()
for linea in arch:
        cad = linea.split()
        x.append( float(cad[0]) ) #eje x
        suma = 0.0
        i = 1
        #ciclo while con el resto de los datos
        while i<6:
                suma += float(cad[i])
                i += 1
        prom = suma/5.0 #cálculo del promedio
        y.append( prom ) #construyo los datos de y

#graficamos  y vs. x
# etiquetas para los ejes
plt.xlabel("X (unidades de x)")
plt.ylabel("Y (unidades de y)")
#rango en los ejes
plt.xlim(0, 0.45)
plt.ylim(0, 30)
#grafica domde se muestran las barras de error
plt.errorbar(x, y, xerr=0.01, yerr=0.5, fmt='bx')
plt.show()
```

Lo único nuevo en el programa anterior el la función graficadora *plt.errorbar(x, y, xerr=0.01, yerr=0.5, fmt='bx')*. Se encarga escencialmente de colocar barras de error de un ancho que esta dado por *xerr* y *yerr*, la opción *fmt='bx'* dibujará además sobre los puntos una *x* de color azul. La grafica generada se ve a continuación

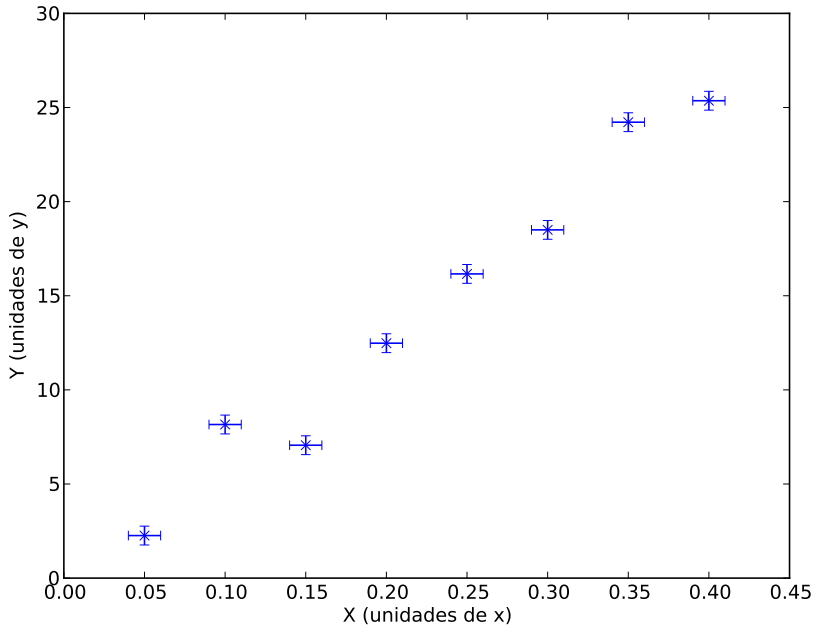


Figura 4.4

4.6 Ejercicios

Ejercicio 4.1 Reescriba el programa del ejercicio 3.3 agregándole comentarios y pidiéndole al usuario los datos necesarios. ■

Ejercicio 4.2 Grafique una función cualquiera en el intervalo [0,50] de tal manera que las curvas contengan 10 puntos, 50 puntos, 100 puntos, 200 puntos y 300 puntos. Observe la diferencias en la gráfica. ■

Ejercicio 4.3 Escoja una función cualquiera de uso frecuente (seno, coseno, exponencial, etc). Desarrolle la función en series de Taylor, encuentre la forma matemática del término genérico. Realice un programa que grafique la función original y luego haciendo uso del término genérico en un ciclo, grafique la serie de Taylor. ■

5 — El IGU no es una fruta

El IGU es una sigla que significa Interfaz Gráfica de Usuario, muchas veces se utiliza el término en ingles que es GUI, pero en castellano es mas guay. Con Python tenemos muchas formas de hacer IGU, una de las más fáciles es utilizando el módulo Tkinter. Nuestro propósito no es realizar una completa y hermosísima IGU, sino mas bien dar una introducción básica y utilizar el "lienzo" para dibujar y visualizar procesos físicos.

5.1 ¿Dónde está la ventanita?

Procedamos a continuación a crear nuestra primera ventana, como se verá esto se puede hacer con un número muy pequeño de líneas de código.

```
'''
Este programa crea una pequeña ventana con un mensaje
'''

#Importa el módulo tkinter
from tkinter import *

#Creo la ventana principal
raiz = Tk()

#Creo una etiqueta que contiene el texto "¿Que hubo?"
etiqueta = Label(raiz, text="¿Que hubo?")

#Ajusta automáticamente el tamaño de la ventana al tamaño de
#la etiqueta y la hace visible
etiqueta.pack()

#Finalmente el bucle principal hace que todo funcione
raiz.mainloop()
```

Analizaremos solo un poco el programa anterior ya que los comentarios son lo suficiente explicativos. Como es de esperar la primera linea que importa es la que importa al módulo Tkinter. Luego creamos la ventana a la que llamamos raiz, desde luego, de la misma manera podemos crear mas de una ventana. Como una ventana que no hace nada es un poco aburrido, se ha incluido una etiqueta que es algo parecido al comando *print* pero para ventanas. Finalmente *.mainloop()* es un ciclo que deja a la ventana en modo de espera, si el usuario no hace nada la ventana se queda esperando toda la vida pobrecita, pero eso ocurre pocas veces, en la mayoría de los casos el usuario mueve la ventana de lugar, la minimiza, etc., a esto se le llama eventos. Por lo tanto este ciclo deja a la ventana en espera de que el usuario realice algún evento. Ahora por favor ejecute el programa anterior en su computadora (tiene computadora ¿verdad?) y luego observe la ventanita producida, observe bien, ya que la ventanita es muy pequeña, por eso se llama ventanita.

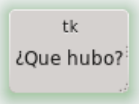


Figura 5.1: *Una ventanita*

En la figura 5.1 se muestra el resultado. Es una ventana totalmente funcional que se puede mover, redimensionar, minimizar, maximizar y matar. Estos últimos eventos se realizan simplemente, como en todas las ventanas, haciendo "click" con el ratón en los botones correspondientes. Lo malo es que la ventana que hemos creado, no se ven dichos botones, esto se debe al uso de *etiqueta.pack()* cuya función es ajustar la ventana a la etiqueta y hacer a esta última visible. Para que aparezcan los botones se debe agrandar la ventana utilizando el ratón tal como se hace con una ventana común y corriente. Una vez que halla jugado con su nueva ventana hasta el aburrimiento, el último evento será matar a la pobre ventanita haciendo "click" en el botón con la cruz, que no le de lástima, ese es el destino final de todas las ventanas.

Lo que sigue, como de costumbre, es complicar el programa un poquito mas, vamos a agregar mas botones. En el siguiente programa a la ventana le agregaremos dos botones, uno imprimirá en el terminal un importantísimo mensaje y el otro cerrará la ventana.

```
#!/usr/bin/env python3.3

'''
Crea una pequeña ventana con botones
'''

#importa el módulo tkinter
from tkinter import *

#Creo la ventana principal
```

```
mamaVentana = Tk()

#la siguiente función imprime un importantísimo
#mensaje en el terminal
def mensaje_importante():
        print("¡Ñángara gángara!")


#creamos un botón para salir de la ventana del lado izquierdo
botoncitoI = Button( text="Adiós", fg="red", command=quit)
botoncitoI.pack(side=LEFT)

#creamos un botón que escribe un mensaje en el terminal del lado derecho
botoncitoD = Button(text="Mensaje", fg="blue", command=mensaje_importante)
botoncitoD.pack(side=RIGHT)

mamaVentana.mainloop()
```

La mayoría de las sentencias del programa anterior ya se conocen, las nuevas son las que crean los botones, para ello se utiliza la función *Button*. Los argumentos utilizados para esta función son *text* cuya función es especificar el texto que aparecerá en el botón, *fg* que especifica el tipo de color del texto, y finalmente *command* que es lo que Tkinter hace cuando hacemos click el botón. En este caso el botoncito izquierdo *botoncitoI* destruye la ventana y el derecho *botoncitoD* ejecuta una función que consiste en imprimir un mensaje en el terminal. Por último la función *.pack()* es la encargada de posicionar el botón en la ventana. En los ejemplos anteriores la ventana tiene un tamaño mínimo ya aparece posicionada en la pantalla en un lugar donde Tkinter decide que es apropiado, generalmente un lugar donde no halla una ventana ubicada previamente. Este comportamiento se puede cambiar con los comandos:

```
#Da el tamaño y ubica la ventana en la pantalla
mamaVentana.geometry("300x100+0+0")
```

Este comando hace que la ventana tenga 300 pixels de ancho por 100 de alto (300x100) y la ubica en la posición (0,0), es decir en el origen de coordenadas con respecto a la pantalla. Por razones históricas y totalmente arbitrarias el origen de coordenadas de la pantalla se encuentra en el margen superior izquierdo. Si además queremos darle un color específico a la ventana, por ejemplo negro, utilizamos el comando:

```
# Le da color al fondo
mamaVentana.config(bg="black")
```

5.2 Sierpinski no es un triángulo

Desde luego una revisión a profundidad del módulo Tkinter nos llevaría mucho tiempo, afortunadamente como se ha visto, trabajar con el es muy sencillo y exiten

exelentes tutoriales y libros[blo; Cha13; GRA00; wika; wikb]. Lo que realizaremos a continuación es crear un lienzo azul en el cual posteriormente dibujaremos figuras geométricas.

```
'''
Ventana con un lienzo azul.
'''

#importa el módulo tkinter
from tkinter import *

#Creo la ventana principal
Ventana = Tk()

#Creamos el lienzo
lienzo = Canvas(bg="blue", height=250, width=300)
lienzo.pack()

#creamos un boton para salir de la ventana del lado izquierdo
botoncitoI = Button(text="SALIR", command=Ventana.quit)
botoncitoI.pack()

Ventana.mainloop()
```

El proximo paso es colocarle a la ventana dos botones adicionales, uno para dibujar un triángulo y el otro para el cuadrado. La instrucción para dibujar en el lienzo un rectángulo es:

```
lienzo.create_rectangle(100,  80, 250, 130, fill="orange", width=5)
```

Los tres números se refieren a la posición en pixels de los vértices medidos tomando como origen la esquina superior izquierda del lienzo. Más específicamente 100, 80 significa que el vértice superior izquierdo del rectángulo se encuentra en una posición $x = 100$ *pixels* y $y = 80$ *pixels*, el otro vértice relevante para posicionar el rectángulo es el inferior derecho que en el ejemplo se encuentra a $x = 250$ *pixels* y $y = 130$ *pixels*, de esta manera se posiciona y se da tamaño al rectángulo. La opción *fill="orange"* simplemente indica el color de relleno del rectángulo, mientras que *width=5* indica que el borde del rectángulo tendrá un ancho de 5 pixels. Para dibujar un triángulo el procediemiento es similar.

```
lienzo.create_polygon( (50,30), (110,50), (20,100), fill="green")
```

Observemos que el comando utilizado es *polygon*, es decir para dibujar un triángulo utilizamos un polígono de tres vértices. También podríamos dibujar un rectángulo utilizando *create_polygon*, pero sería un poco mas complicado que la forma mostrada anteriormente con *create_rectangle*, ya que tendríamos que dar los cuatro vertices, calcular que los ángulos fueran rectos, etc. Por supuesto con *polygon* podemos dibujar

polígonos con un número arbitrario de vértices. En el ejemplo se han puesto los vértices entre paréntesis para que resulte mas claro, pero esto no es obligatorio. El posible resultado final es:

```
'''
Ventana con un lienzo azul y figuras geomérticas.
'''

#importa el módulo tkinter
from tkinter import *

#Creo la ventana principal
Ventana = Tk()

#Creamos al lienzo
lienzo = Canvas(bg="blue", height=250, width=300)
lienzo.pack()

#Esta función crea un rectángulo
def rectangulo():
    lienzo.create_rectangle(100,  80, 250, 130, fill="orange", width=5)

#Y esta función crea un triángulo
def triangulo():
    lienzo.create_polygon( (50,30), (110,50), (20,100), fill="green")

#creamos un boton para salir de la ventana del lado izquierdo
botonI = Button( text="SALIR", command=Ventana.quit)
botonI.pack(side=LEFT)
#creamos un boton para dibujar un rectángulo
botonII = Button( text="Rectángulo", command=rectangulo)
botonII.pack(side=RIGHT)
#creamos un boton para dibujar un triángulo
botonIII = Button(text="Triángulo", command=triangulo)
botonIII.pack()

Ventana.mainloop()
```

En este programa solo debemos hacer “click” en el boton triángulo o rectángulo para que las figuras se dibujen, y esto es independiente del orden en que se haga. El resultado se aprecia en la figura 5.2

Ya que sabemos como dibujar triángulos ahora aprenderemos a *no* dibujar un triángulo. Esto requiere una explicación concienzuda y detallada que yo no pienso ofrecer, solo diré que los triángulos al igual que cualquier figura plana tienen solo dos dimensiones. Matemáticamente esto se puede expresar de la siguiente manera, supongamos que la letra L representa la dimensión espacial, es decir la longitud, entonces el área de una figura plana como el triángulo es proporcional al cuadrado de la longitud, $A \propto L^2$. Hasta

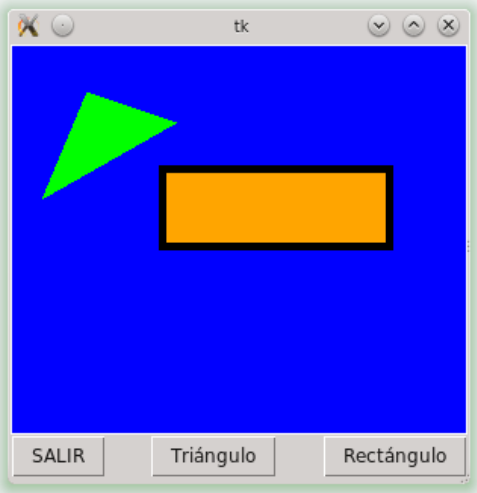


Figura 5.2: *Una ventana con lienzo y figuras*

hace poco[1], solo conocíamos objetos geométricos con un número entero de dimensiones, las curvas tienen una dimensión (L), las figuras planas tienen dos dimensiones (L^2), las pirámides, cubos, y demás poliedros son cuerpos geométricos con tres dimensiones (L^3), a algunos matemáticos les encanta hablar de cosas como el hipercubo que seria una especie de cubo en cuatro dimensiones (L^4), etc. Con la ayuda de las matemáticas podemos utilizar objetos geométricos con un número arbitrario de dimensiones, aunque seamos totalmente incapaces de representar en nuestra cabeza ese tipo de artefactos, como por ejemplo un cubo de diez dimensiones.

No fue sinó hasta que un señor llamado Mandelbrot descubrió allá por la década de los 70 del siglo pasado (suena muy lejano, pero para el mundo de la geometría esto fue apenas ayer) que había objetos geométricos que tienen una dimensión no entera. A estos objetos les llamó fractales. Para entender lo que es un fractal, lo mejor es empezar por ver uno. En la figura 5.3 se aprecia como se construye uno de los fractales mas famosos, el triángulo de Sierpinski (parece pero no es un triángulo) llamado así en honor al genial matemático Polaco que lo descubrió a principios del siglo XX. En dicha figura se observa una de las maneras en las cuales se puede construir. Se comienza por un triángulo equilátero de color negro, luego a este triángulo se le quita un triángulo invertido justo del centro, tal como se ve en la figura. Ahora pareciera que tenemos tres triángulos negros mas pequeños. La tercera etapa consiste en repetir el proceso con cada uno de estos tres triángulos y así se obtiene el tercer dibujo de la figura 5.3, en la cual se dibujan igualmente la cuarta y quinta etapa. El triángulo de Sierpinski queda terminado después de repetir el proceso anterior infinitas veces.

Este tipo de figuras planas no llena el plano en el cual está dibujado, de hecho tiene tantos "huecos" que está casi "vacio", y por ende no tiene dimensión igula a 2. Encontrar la dimensión a un objeto fractal se escapa del propósito de este libro, pero basta decir que hay varias definiciones de dimensión fractal. La dimensión fractal de

[1] Si tomamos en cuenta la larga historia de las matemáticas

Figura 5.3: *Una manera de construir el triángulo de Sierpinski*

Hausdorff-Besicovitch para el triángulo de Sierpinski es $D_{HB} = \frac{ln3}{ln2}$. en general se puede decir que un obgeto es un fractal si cumple con las siguientes caracteristicas.

- Autosimilitud. Esto significa que si hacemos un cambio de escala como una ampliación o una reducción, el objeto se sigue viendo igual.
- Dimensión fractal. Es decir su dimensión no es un número entero.
- Se puede definir recursivamente

Los fractales no se restringen al ámbito de las matemáticas, Tambien existen una cantidad enorme de objetos reales que se pueden considerar fractales, claro la propiedad de autosimilitud tiene un límite ya que la naturaleza no se lleva bien con los infinitos. Ejemplo de fractales en la naturaleza son, las nubes, los bronquios, los árboles, los helechos, el romanescu[2], las cuencas hidrográficas, los copos se nieve, las descargas electricas, y un gran etc.

Lógicamente lo que sigue a continuación es implementar el triángulo de Sierpinski con la ayuda de Python. La forma en que lo haremos es diferente de la mostrada en la figura 5.3, utilizaremos un proceso iterativo. Es muy curioso como este tipo de reglas muy sencillas ejecutadas de forma iterativa producen fractales, el procedimiento es el siguiente.

1. Se fijan los tres vértices del triangulo.
2. Escojamos un punto cualquiera del plano, haciendo uso de nuestra prodigiosa imaginación lo llamaremos P_0.
3. Se escoge aleatoriamente alguno de los tres vértices.
4. Se halla el punto medio de la recta que une el vértice escogido con el punto P_0. A este punto lo llamaremos $P1$ y lo dibujamos en el lienzo.
5. Actualizamos $P_0 = P_1$ y repetimos el proceso iterativamente a partir de item 2.

Nótese que si nuestro punto inicial está muy lejos del interior del triángulo, los primeros puntos no no son validos. Solo sirven los puntos que siguen el procedimiento anterior que son internos al triángulo. Adicionalmente aunque esté dentro del triángulo, el primer punto como es aleatorio, nada nos garantiza que sea un punto válido, podría por ejemplo, caer cerca del centro del triángulo donde no hay ningún punto. Afortunadamente el proceso converge rápidamente y en pocos pasos los puntos generados son válidos. Sin embargo para que no se vea feo, los primeros puntos deben ser desechados, por lo tanto debemos hacer dos ciclos. En el primero de ellos generamos puntos que no dibujamos (unos diez son suficientes) y el segundo ciclo será el que utilizamos para dibujar. El programa es el siguiente:

[2]El romanescu es un híbrido de brécol y coliflor de la familia de las brasicáceas. El brécol romanescu fue documentado inicialmente en Italia (como Broccolo romanesco) en el siglo XVI.A Gracias a Dios, (y a los señores Wales y Sanger) existe wikipedia

```
"""
Dibuja en la pantalla el triángulo de Sierpiński
"""
import numpy as np
#Llamo al módulo aleatorio
import random as rnd
#Llamo al módulo gráfico
import tkinter as Tki

#creo una ventana
window = Tki.Tk()

#creo el lienzo
w = 600 #ancho de la ventana
hh = 500 #altura de la ventana
lzo = Tki.Canvas(window, width=w, height=hh, background="white")
lzo.pack()

# vértices del triángulo de sierpinski
A = np.array([50, 450])
B = np.array([550, 450])
C = np.array([300,(450-np.sqrt(3)*250)] )

#primer punto
P0 = (50,450)

#número de vértices
sec =(1,2,3)

#Fase transitoria
for i in range (10):
        #genera un número aleatorio de la secuencia sec
        n = rnd.choice(sec)
        if n == 1:
                D = A
        elif n == 2:
                D = B
        else:
                D = C
        P1 = (P0 + D) * 0.5
        P0 = P1

#Dibuja el triángulo de Sierpiński
for i in range (90000):
        #genera un número aleatorio de la secuencia sec
        n = rnd. choice(sec)
```

```
        if n == 1:
                D = A
                color = "green"
        elif n == 2:
                D = B
                color = "blue"
        else:
                D = C
                color = "red"
        P1 = (P0 + D) * 0.5
        x,y = P1
        lzo.create_line(x, y, x+1, y+1, fill = color)
        lzo.update()
        P0 = P1

print("fin")
#necesario para que aparezca la ventana
window.mainloop()
```

Analizaremos brevemente el programa anterior. Lo primero fue importar los módulos, obsérvese que a diferencia de los ejemplos anteriores, el módulo tkinter se importó utilizando un alias (se importa como Tki) que es una práctica muy utilizada. Además se importó un módulo nuevo llamado *random* que se utiliza para generar números aleatorios. Luego se crea la ventana y el lienzo. Posteriormente se definen los tres vértices del triángulo. En este programa hemos escogido un valor fijo para el punto inicial P0, pero bien pudo haber sido aleatorio, eso no es importante. A continuación implementamos una corta fase transitoria (apenas diez puntos) que consiste en primer lugar en obtener aleatoriamente un número de la secuencia *sec* que consta de tres números (1, 2, 3), esto es equivalente a lanzar un dado de tres lados [3]. Luego haciendo uso adecuado del *if*, del *elif*, y del *else* seleccionamos el vértice correspondiente para mas tarde hallar el punto medio con *P1 = (P0 + D) * 0.5*. Observemos que P0, P1 y D son listas con dos elementos cada uno que representan las coordenadas x y y de cada punto. Finalmente se actualiza el valor de P0 para seguir con el proceso iterativo. A continuación sigue el código donde se dibuja el triángulo propiamente dicho con unos 90.000 puntos. Es muy similar al anterior solo que hemos agregado color dependiendo del vértice seleccionado. En *x,y = P1* se recuperan las coordenadas del punto para poder dibujarlo con la instrucción *lzo.create_line(x, y, x+1, y+1, fill = color)*, donde para dibujar un punto simplemente hemos creado una linea de un pixel de largo, es decir hemos dibujado un pixel y lo hemos rellenado del color anteriormente seleccionado. La instrucción *lzo.update()* actualiza el dibujo cada vez que se dibuja un pixel, esto hace el proceso muchísimo mas lento, pero así tenemos el placer de ver aparecer el triángulo de Sierpinski delante de nuestros ojos. Como el programa se tarda un poquito en dibujar los 90.000 puntos, finalmente imprimimos el la pantalla la palabra *fin* para estar seguros de que el programa al fin a terminado. El resultado final se ve en la figura 5.4

[3] Ciertamente los dados de tres lados no existen en el universo conocido, pero si existieran la forma de simularlo con Python sería precisamente esa.

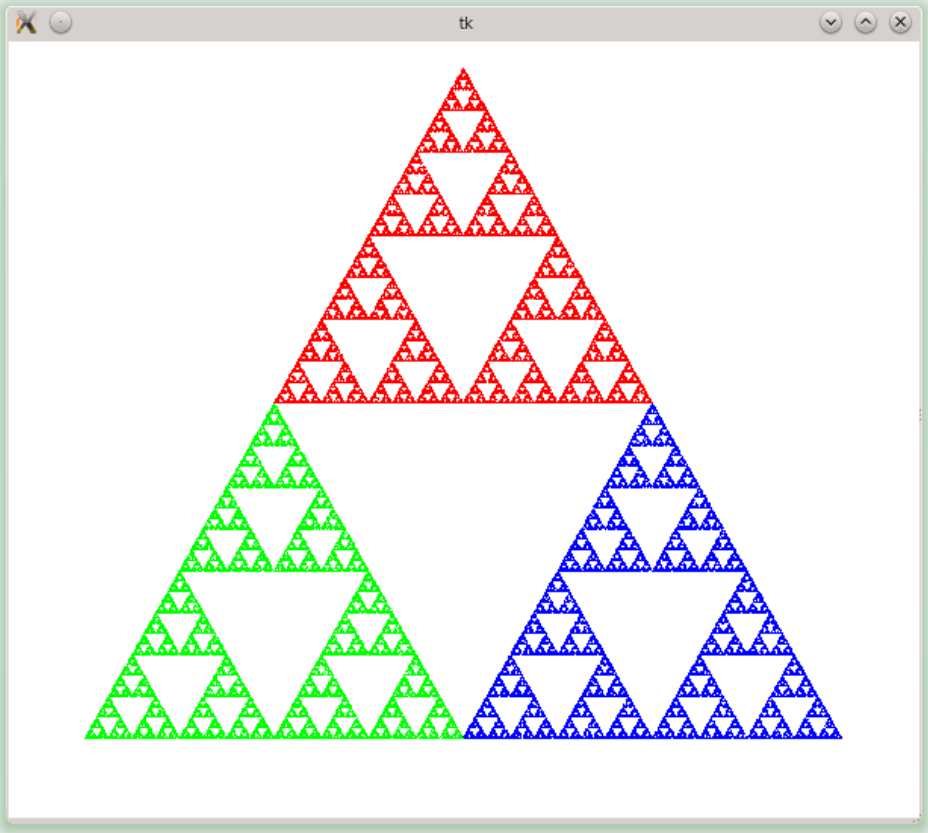


Figura 5.4: *Triángulo de Sierpinski*

A — Palabras reservadas

La siguiente es una lista de las palabras reservadas que utiliza Python.

and	**False**	**nonlocal**
as	**finally**	**not**
assert	**for**	**or**
break	**from**	**pass**
class	**global**	**raise**
continue	**if**	**return**
def	**import**	**True**
del	**in**	**try**
elif	**is**	**while**
else	**lambda**	**with**
except	**None**	**yield**

Bibliografia

Artículos

[PG07] Fernando Pérez y Brian E. Granger. «IPython: a System for Interactive Scientific Computing». En: *Computing in Science and Engineering* 9.3 (mayo de 2007), páginas 21-29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: http://ipython.org (véase página 51).

Libros

[Bea09] David M Beazley. *Python Essential Reference*. Pearson Education, 2009 (véase página 51).

[Ber11] Herman J. C. Berendsen. *A Student's Guide to Data and Error Analysis*. Editado por Cambridge University Press. 2011 (véase página 17).

[Cha13] Bhaskar Chaudhary. *Tkinter GUI Application Development*. Packt Publishing Ltd, 2013 (véase página 92).

[Dal+09] Darren Dale y col. *Matplotlib*. 2009 (véase página 75).

[Dev14] Alexandre Devert. *matplotlib Plotting Cookbook*. 4 Packt Publishing, 2014 (véase página 75).

[GRA00] JOHN E. GRAYSON. *Python and Tkinter Programming*. Manning Publications Co., 2000 (véase página 92).

[Gre06] Douglas Gregory. *Classical Mechanics*. Cambridge University Press, 2006 (véase página 60).

[HRW08] Holliday, Resnick y Walker. *Fundamentals of Physics*. John Wiley y Sons, Inc., 2008 (véase páginas 19, 33).

[LPB08] Rubin H. Landau, Manuel José Paez y Cristian C. Bordeianu. *A Survey of Computational Physics*. PRINCETON UNIVERSITY PRESS, 2008 (véase páginas 15, 17).

[Lan09] Hans Petter Langtangen. *A Primer on Scientific Programming with Python*. Volumen 9. Texts in Computational Science and Engineering. Springer, 2009 (véase página 29).

[Lut13] Mark Lutz. *Learning Python, Fifth Edition*. O'Reilly, 2013 (véase página 37).

[MG04] Frank Mittelbach y Michael Goossens. *The LaTeXCompanion*. Second Edition. Addison-Wesley, 2004 (véase página 77).

[PH04] David A. Patterson y John N. Hennessy. *Computer Organization Desing*. Editado por Elsevier. 3.ª edición. 2004 (véase página 15).

[Phi10] Dusty Phillips. *Python 3 Object Oriented Programming*. Packt Publishing Ltd., 2010 (véase página 46).

[Pil09] Mark Pilgrim. *Dive Into Python 3*. Springer-Verlag, 2009 (véase página 51).

[Ros13] Cyrille Rossant. *Learning Ipython for Interactive Computing and Data Visualization*. Packt Publishing, 2013 (véase página 51).

[SJ10] Raymond A. Serway y John W. Jewett. *Physics for Scientists and Engineers*. Brooks/Cole, 2010 (véase página 33).

[Tos09] Sandro Tosi. *Matplotlib for Python Developers*. Packt Publishing, 2009 (véase página 75).

Páginas Web

[blo] blogspot. *Tutorial TKInter Python (GUI)*. URL: http://gmendezm.blogspot.com/2012/12/tutorial-tkinter-python-gui.html (véase página 92).

[Cou] Cambridge Course. *Numerical Programming in Python*. URL: http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/NumericalPython/ (véase página 20).

[FSF] FSF. *Free Software Foundation*. Free Software Foundation. URL: http://www.fsf.org/ (véase página 7).

[GNU] GNU. *Various Licenses and Comments about Them*. GNU. URL: http://www.gnu.org/licenses/license-list.html (véase página 7).

[GPL] GPL. *GNU General Public License*. URL: http://www.gnu.org/licenses/gpl.html (véase página 7).

[OP] OP. *The Open Source Initiative*. URL: http://www.opensource.org/ (véase página 7).

[Pyt] Python. *Python Programming Language Official Website*. URL: http://www.python.org/ (véase página 8).

[Ram] Chet Ramey. *Bash*. URL: http://tiswww.case.edu/php/chet/bash/bashtop.html (véase página 65).

[Tea] The IPython Development Team. *IPython Documentation*. URL: http://ipython.org/documentation.html (véase página 51).

[Wika] Wikipedia. *Fibonacci*. URL: http://en.wikipedia.org/wiki/Fibonacci (véase página 44).

[Wikb] Wikipedia. *Fibonacci number*. URL: http://en.wikipedia.org/wiki/Fibonacci%20_number (véase página 43).

[wika] wiki.python. URL: https://wiki.python.org/moin/TkInter (véase página 92).

[wikb] wiki.Tkinter. URL: http://tkinter.unpythonic.net/wiki/ (véase página 92).

Índice alfabético