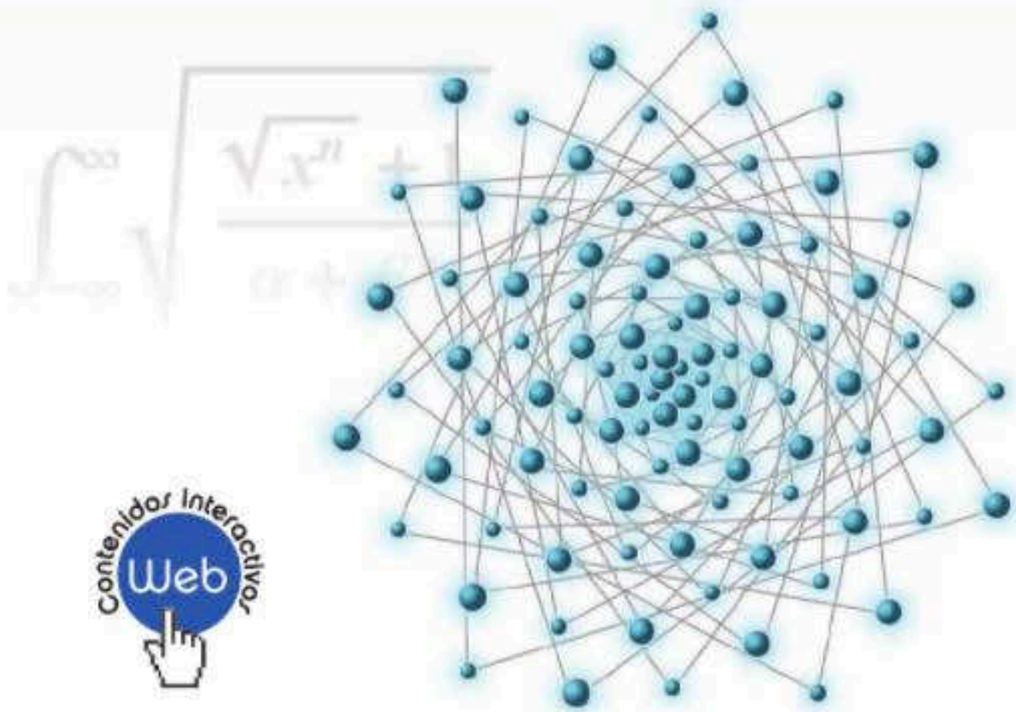
OFELIA D. CERVANTES VILLAGÓMEZ · DAVID BÁEZ LÓPEZ
ANTONIO ARÍZAGA SILVA · ESTEBAN CASTILLO JUÁREZ

python™

CON APLICACIONES A LAS MATEMÁTICAS, INGENIERÍA Y FINANZAS



Alfaomega

Director Editorial:
Marcelo Grillo Giannetto
mgrillo@alfaomega.com.mx

Jefe de Edición:
Francisco Javier Rodríguez Cruz
jrodriguez@alfaomega.com.mx

Datos catalográficos
Cervantes, Ofelia; Báez, David; Arízaga, Antonio; Castillo, Esteban
Python con aplicaciones a las matemáticas, ingeniería y finanzas
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-622-673-5
Formato: 17 x 23 cm Páginas: 452

Python con aplicaciones a las matemáticas, ingeniería y finanzas
Ofelia Delfina Cervantes Villa Gómez, José Miguel David Báez López, Antonio Arízaga Silva, Esteban Castillo Juárez.

Derechos reservados © Alfaomega Grupo Editor, S.A. de C.V., México

Primera edición: Alfaomega Grupo Editor, México, agosto 2017

© 2017 Alfaomega Grupo Editor, S.A. de C.V.
Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720. Ciudad de México

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: http://www.alfaomega.com.mx
E-mail: atencionalcliente@alfaomega.com.mx

ISBN: 978-607-622-673-5

Derechos reservados:
Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en todo el mundo.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, C.P. 06720, Del. Cuauhtémoc, Ciudad de México – Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 – E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215, piso 10, CP: 1055, Buenos Aires, Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

Acerca de los autores

Ofelia Cervantes Villagómez Ingeniera en Sistemas Computacionales (UDLAP), DEA en Informática (ENSIMAG, Francia) y Doctorado en Computación (INPG, Francia). Es profesora del Departamento de Computación, Electrónica y Mecatrónica en la UDLAP desde 1988 donde fue Jefa del Departamento de 1990 a 1993. Sus áreas de especialidad son la inteligencia artificial, bases de datos y sistemas distribuidos.

Fue Decana de la Escuela de Ingeniería y posteriormente Decana de Asuntos Internacionales de 1994 a 2005. De 2006 a 2008 creó y dirigió la Escuela de Verano de la UDLAP. De 2011 a 2012 fue fundadora y primera Directora del Programa de Liderazgo para Jóvenes Indígenas. Ha colaborado durante décadas en la vinculación de la comunidad académica especializada en TICs, con sus homólogos en EUA, Francia y Alemania. Ha publicado en congresos y revistas nacionales e internacionales. Es coautora de libros y capítulos de libros en su área de especialidad.

Desde 2003 es Cónsul Honoraria de Francia en Puebla. En Abril 2014 fue condecorada por el Presidente de la República Francesa como Caballero de la Orden Nacional del Mérito como reconocimiento a los servicios brindados a Francia en la protección consular y promoviendo la cooperación académica con México. Es miembro de la Academia Mexicana de Computación.

David Báez López. Licenciado en Física (UAP), Maestro en Ciencias (Universidad de Arizona), Doctor en Ingeniería Eléctrica (Universidad de Arizona). De 1979 a 1985 fue investigador del Instituto Nacional de Astrofísica, Óptica y Electrónica en Tonantzintla, Puebla, siendo Coordinador de Electrónica y Director Técnico Interino. De 1985 a 2015 fue profesor del Departamento de Computación, Electrónica y Mecatrónica de la Universidad de las Américas Puebla (UDLAP) en Cholula, Puebla, México. En la UDLAP fungió como jefe del Departamento de Ingeniería Electrónica y como Presidente del Comité de Evaluación de la Facultad. Así mismo, es fundador del Congreso Internacional de Electrónica, Comunicaciones y Computadoras que en 2017 está en su 27ª edición y que se lleva a cabo en la UDLAP.

Ha publicado más de cien papers en revistas y congresos nacionales e internacionales, respectivamente, es autor y coautor de capítulos de libros y de trece libros acerca de simulación de circuitos, MATLAB, y procesado analógico de señales. Como investigador fue miembro del Sistema Nacional de Investigadores de 1984 hasta 2014.

Juan Antonio Arízaga Silva recibió el grado de Maestro en Ciencias de la Universidad de las Américas Puebla en 2006 y el título de licenciado en electrónica en 2003 por la Benemérita Universidad Autónoma de Puebla. Actualmente es profesor investigador de tiempo completo en la Universidad Politécnica de Puebla en el área de Sistemas Automotrices. Ha escrito más de 20 artículos técnicos en diversas revistas nacionales e internacionales. Las áreas de interés de sus investigaciones incluyen sistemas embebidos, sistemas de telemetría automotriz y sistemas electrónicos de potencia.

Esteban Castillo Juárez es Licenciado en ciencias de la computación (2009) y maestro en ciencias de la computación (2012) por parte de la Benemérita Universidad Autónoma de Puebla. Actualmente es estudiante de doctorado en ciencias de la computación en la Universidad de las Américas Puebla. Sus áreas de interés incluyen el procesamiento de lenguaje natural, minería de datos, aprendizaje automático, análisis de redes sociales y teoría de grafos.

Ha publicado mas de 15 artículos internacionales relacionados con el tratamiento automático del texto utilizando representaciones basadas en grafos con el lenguaje de programación Python, con el cual ha trabajado activamente por más de 5 años.

Contenido

Prólogo **XVII**

Plataforma de contenidos interactivos **XIX**

1. Introducción 1

1.1. Introducción . . . 2

1.2. Evolución de la computadora . . . 2

1.3. Arquitectura de una computadora . . . 6

1.3.1. Instrucciones y datos de una computadora . . . 7

1.3.2. Sistemas numéricos decimal y binario . . . 7

1.3.3. Otros sistemas numéricos . . . 11

1.3.4. Estructura de la memoria . . . 12

1.3.5. Lenguajes de una computadora . . . 13

1.4. El lenguaje Python . . . 14

1.4.1. Descarga de Python . . . 15

1.4.2. La interfase de Python . . . 15

1.5. Organización del libro . . . 16

2. Fundamentos de algoritmos y de Python **17**

2.1. Introducción . . . 18

2.2. ¿Qué es un algoritmo? . . . 18

2.3. Pseudocódigo . . . 21

2.4. Variables . . . 22

2.4.1. Tipos de variables . . . 22

2.4.2. Asignación de valores a variables . . . 23

2.4.3. Inicialización de variables . . . 23

2.4.4. Operaciones básicas . . . 24

2.4.5. Operaciones con variables enteras . . . 25

2.4.6. Operaciones con enteros y reales . . . 25

2.5. Partes de un algoritmo 26
2.6. Algoritmos en pseudocódigo 27
2.7. Lenguaje Python 29
2.7.1. El ambiente de Python 30
2.8. Estructura de un algoritmo en Python 32
2.9. Variables 32
2.10. Bibliotecas y encabezados 34
2.11. Operadores 35
2.11.1. Operadores aritméticos 35
2.11.2. Operadores relacionales 36
2.11.3. Operadores lógicos 36
2.11.4. Otras operaciones booleanas 38
2.11.5. Operadores de asignación 39
2.12. Comentarios 39
2.13. Entrada y salida de datos en Python 40
2.13.1. Despliegue de datos 40
2.13.2. Entrada de datos 42
2.13.3. Tabulador 43
2.14. Algoritmos sencillos 43
2.15. Variables alfanuméricas en Python 48
2.15.1. Operaciones con cadenas 49
2.15.2. Concatenación o suma de cadenas 50
2.15.3. Multiplicación de cadenas 50
2.16. Listas 52
2.16.1. Diccionarios 53
2.17. Instrucciones de Python del Capítulo 2 54
2.18. Conclusiones 54
2.19. Ejercicios 55

3. Condiciones **59**
3.1. Introducción 60
3.2. Condiciones 61
3.2.1. La condición en Python 63
3.2.2. Ejercicios de condiciones simples 68
3.3. La condición **Si - Si_no** 73
3.3.1. Condición `if - else` 73

3.3.2. Ejemplos adicionales de condiciones if-else 75
3.4. Condiciones anidadas 77
3.4.1. if anidado en Python 78
3.4.2. Ejemplos adicionales con condiciones anidadas 80
3.5. Casos 83
3.6. Instrucciones de Python del Capítulo 3 89
3.7. Conclusiones 89
3.8. Ejercicios 89

4. Ciclos en Python **95**
4.1. Introducción 96
4.2. Ciclos Mientras 96
4.2.1. El ciclo Mientras en Python 100
4.3. Ciclos Para 106
4.3.1. La función **range** 112
4.4. Ciclos anidados 113
4.5. La instrucción continue 123
4.6. La instrucción `break` 124
4.7. Ejemplos adicionales 126
4.8. Instrucciones de Python del Capítulo 4 132
4.9. Conclusiones 133
4.10. Ejercicios 133

5. Cadenas, Listas, Diccionarios y Tuplas **137**
5.1. Introducción 138
5.2. Cadenas 138
5.2.1. Longitud de una cadena 141
5.2.2. Separación de una cadena 141
5.2.3. Operaciones con cadenas 143
5.2.4. Concatenación o suma de cadenas 143
5.2.5. Multiplicación de cadenas 144
5.2.6. Inmutabilidad de las cadenas 144
5.2.7. Otras operaciones con cadenas 145
5.2.8. Condiciones 148
5.2.9. Ciclos 149
5.3. Listas 151

5.4. Definición de listas 152
5.4.1. Operaciones con listas 152
5.4.2. Mutabilidad 154
5.4.3. Conversión de lista de cadenas a cadena 155
5.4.4. Otras operaciones con listas 156
5.5. Tuplas 159
5.5.1. Intercambio de valores 159
5.5.2. Operaciones con tuplas 160
5.6. Diccionarios 161
5.6.1. Otras operaciones para diccionarios 162
5.7. Instrucciones de Python del Capítulo 5 164
5.8. Conclusiones 164
5.9. Ejercicios 164

6. Arreglos I: Vectores 169
6.1. Introducción 170
6.2. Introducción a arreglos 170
6.3. Vectores 171
6.3.1. Acceso a vectores 175
6.3.2. Importancia del tamaño de un vector 179
6.4. Vectores en Python 180
6.4.1. Vectores por comprensión 181
6.5. Ejemplos con vectores en Python 182
6.5.1. Importancia del tamaño de un vector en Python 193
6.5.2. Inicialización de los vectores 194
6.5.3. La instrucción append 194
6.6. Ordenamiento de vectores 196
6.6.1. Ordenamiento por selección 196
6.6.2. Ordenamiento de burbuja 200
6.7. Búsquedas 204
6.7.1. Búsqueda binaria 205
6.8. Instrucciones de Python del Capítulo 6 208
6.9. Conclusiones 208
6.10. Ejercicios 208

7. Arreglos II: Matrices **211**
7.1. Introducción . . . 212
7.2. Matrices . . . 212
7.3. Arreglos en Python . . . 214
7.3.1. Generación de arreglos por indexación . . . 215
7.3.2. Generación de arreglos por comprensión . . . 217
7.4. Métodos alternos de escritura de matrices . . . 218
7.5. Selección de filas y columnas de un arreglo . . . 220
7.5.1. Filas de un arreglo . . . 220
7.5.2. Columnas de un arreglo . . . 221
7.6. Suma, resta y multiplicación de matrices . . . 221
7.6.1. Suma y resta de matrices . . . 222
7.6.2. Suma y resta de matrices en Python . . . 223
7.6.3. Multiplicación de matriz por un escalar . . . 225
7.6.4. Multiplicación de matriz por una matriz . . . 227
7.6.5. Producto de matrices en Python . . . 229
7.7. Matrices especiales . . . 231
7.7.1. La matriz identidad . . . 231
7.7.2. La matriz identidad en Python . . . 231
7.7.3. La matriz transpuesta . . . 233
7.7.4. Transpuesta de un arreglo en Python . . . 234
7.7.5. Obtención de la transpuesta por comprensión . . . 236
7.7.6. La matriz simétrica . . . 237
7.8. Ejemplos . . . 239
7.9. Conclusiones . . . 245
7.10. Ejercicios . . . 245

8. Subalgoritmos **249**
8.1. Introducción . . . 250
8.2. Subalgoritmos . . . 250
8.3. Funciones . . . 252
8.4. Funciones en Python . . . 255
8.4.1. Funciones de Python . . . 256
8.4.2. Funciones definidas por el usuario . . . 256
8.5. Procedimientos . . . 261
8.5.1. Recursividad . . . 266

8.6. Funciones lambda 267
8.7. Llamado por valor y llamado por referencia 268
8.8. Variables locales y globales 269
8.9. Ejemplos adicionales 272
8.10. Instrucciones de Python del Capítulo 8 292
8.11. Conclusiones 292
8.12. Ejercicios 292

9. Entrada y salida y de datos con archivos 295
9.1. Introducción 296
9.2. Escritura de datos en un archivo 296
9.2.1. Escritura de datos alfanuméricos 297
9.2.2. La instrucción `with` 300
9.3. Escritura de datos numéricos 300
9.4. Lectura de datos de un archivo 303
9.4.1. Lectura de datos de un archivo 304
9.5. Lectura y escritura de datos en Excel 308
9.6. Instrucciones de Python del Capítulo 9 313
9.7. Conclusiones 314
9.8. Ejercicios 314

10. Programación orientada a objetos 315
10.1. Introducción 316
10.2. Conceptos asociados a la POO 316
10.3. Primera clase en Python 319
10.4. Creación de la clase `NumeroComplejo` 321
10.5. Declaración y uso de Setters y Getters 322
10.6. Sobreescritura de operadores 324
10.7. Herencia 326
10.8. Sobreescritura de métodos 330
10.9. Ejemplos 331
10.9.1. Creación y movimientos de una cuenta bancaria 331
10.9.2. Cálculo de la media aritmética y la desviación estándar 334
10.9.3. Operaciones con matrices 337
10.9.4. Figuras geométricas 341
10.9.5. Lista enteros 344

10.10. Instrucciones de Python del Capítulo 10 349
10.11. Conclusiones 350
10.12. Ejercicios 350

11. Graficación en Python 351
11.1. Introducción 352
11.2. Visualización de datos 352
11.3. Gráficas en 2 dimensiones 353
11.4. Figuras múltiples 360
11.5. Subgráficas 362
11.6. Otros tipos de gráficas bidimensionales 365
11.6.1. Gráfica polar 365
11.6.2. Gráfica de pie 366
11.6.3. Gráfica de histograma 367
11.6.4. Gráfica de `stem` o de puntos 368
11.7. Opciones de gráficas 370
11.8. Gráficas tridimensionales 371
11.9. Instrucciones de Python del Capítulo 11 379
11.10. Conclusiones 380
11.11. Ejercicios 381

12. Geolocalización y Análisis de Sentimientos 383
12.1. Geolocalización 384
12.2. El módulo `geopy` 384
12.2.1. Geolocalización de un punto de interés 385
12.2.2. Distancia de dos puntos de interés 387
12.2.3. Visualización de distintos puntos de interés 388
12.3. Análisis de sentimientos de Twitter 390
12.3.1. Extracción de tweets 391
12.3.2. La base de datos MongoDB 395
12.3.3. Análisis de los tweets 398
12.4. Conclusiones 403

A. Instalación y configuración 405
A.1. Introducción 406
A.2. Instalación de Python 406

A.3. Instalación de easy_install y pip 408
A.4. Instalación de Numpy 411
A.5. Instalación de Scipy 412
A.6. Instalación de Matplotlib 413
A.7. Instalación de Tweepy 415
A.8. Instalación de Pymongo 416
A.9. Instalación de Geopy 417
A.10. Instalación de Matplotlib Basemap 418

B. Creación de ejecutables en Python **421**
B.1. Introducción 422
B.2. Instalación de `pyinstaller` 422
B.3. Creación de ejecutables 422
B.4. Ejemplo 423

Prólogo

Cuando nos propusimos escribir un libro sobre programación para alumnos de primer semestre de licenciatura, examinamos las tendencias en el uso de lenguajes de programación en la industria y en las universidades. El resultado fue que Python era el lenguaje que estaba tomando fuerza y colocándose en los primero lugares en la lista de los lenguajes más usados. Hay varias razones para este hecho. Python es software libre y se cuenta con una gran cantidad de módulos para realizar cómputo científico que encuentra aplicaciones en muchas áreas del conocimiento científico, desde medicina, ingeniería, ciencias, finanzas, economía, por mencionar algunas cuantas. Además, Python se ha erigido como un rival de respeto de MATLAB® con la ventaja de tener una distribución gratuita. Al momento de la publicación de este libro, Python se encuentra posicionado en el cuarto lugar de la lista de lenguajes de programación más usados por programadores, pero colocado en primer lugar entre los lenguajes de programación de muy alto nivel. Adicionalmente, el IEEE lo ha colocado en tercer lugar entre los lenguajes de programación. Estas estadísticas nos hicieron seleccionar a Python como el lenguaje de programación para este libro.

El contenido del presente libro ha sido parte del curso introductorio de programación para estudiantes de ciencias, ingeniería y animación digital a nivel licenciatura de la Universidad de las Américas Puebla. Constituye para muchos estudiantes el primer contacto con un lenguaje de programación. Durante el libro hacemos énfasis en la práctica de las instrucciones, inicialmente en algoritmos sencillos y aumentando el nivel de complejidad de acuerdo al avance de los temas. El presente material consiste en una versión de los temas que hemos impartido durante los últimos dos años. La importancia del lenguaje de programación Python se hace patente dado el crecimiento en su uso en las universidades y en la industria, a un grado tal que las nanocomputadora Raspberry Pi lo usan como lenguaje de programación.

Un aspecto importante del libro consiste en desarrollar desde los primeros capítulos la descripción, análisis y desarrollo de los algoritmos que posteriormente se implementan en el lenguaje de programación Python. De esta manera el usuario va de una descripción del problema en el idioma español, pero con una estructura definida, a una implementación formal en un lenguaje de programación. De esta manera el lector ve como se va construyendo un algoritmo para de una manera sencilla poder implementarlo en Python. Con este propósito el libro desarrolla más de 150 ejemplos, con los cuales el lector aprende los conceptos presentados en cada capítulo.

Como en toda rama de la ingeniería, en la programación con Python, es necesario tener un concepto integral de aprendizaje que complemente la teoría y los ejemplos resueltos. Por lo tanto, cada capítulo tiene como complemento una serie de ejercicios que el lector puede resolver y que le permiten ganar más experiencia en la solución de problemas implementando sus algoritmos en lenguaje Python.

Tenemos confianza en que el libro permitirá a todos sus lectores aprender a diseñar algoritmos implementándolos en el lenguaje Python.

Competencias Generales

Con el material de este libro el lector desarrollará las siguientes competencias:

1. Desarrollar algoritmos en pseudocódigo.
2. Implementar los algoritmos en lenguaje Python.
3. Graficar resultados de los algoritmos.
4. Usar Python para distintas aplicaciones como gelocalización, análisis de sentimientos, algebra lineal, cadenas de Markov, procesado de señales, operaciones en finanzas, entre otras.

Al final del libro se incluyen tres Apéndices. El primer Apéndice explica como instalar módulos adicionales para realizar otras funciones en Python tales como graficar, realizar otras operaciones numéricas, financieras, geolocalizació, entre otras. El segundo explica como realizar archivos ejecutables en Pyhton, de tal manera que no se requiera tener instalado Python para su ejecución. El tercer y último apéndice señala las principales diferencias entre Python 2 y Python 3.

Se espera que el lector aprenda a desarrollar algoritmos en pseudocódigo, los cuales se puedan traducir al lenguaje de programación Python o a cualquier otro lenguaje que el lector requiera implementar.

Plataforma de contenidos interactivos

Para tener acceso al material de la plataforma de contenidos interactivos de este libro, siga los siguientes pasos:

- Ir a la página: http://libroweb.alfaomega.com.mx
- Ir a la sección Catálogo y seleccionar la imagen de la portada del libro, al dar doble clic sobre ella tendrá acceso al material descargable.

NOTA: Se recomienda respaldar los archivos descargados de la página web en un soporte físico.

Capítulo 1
Introducción

1.1 Introducción
1.2 Evolución de la computadora
1.3 Arquitectura de una computadora
1.4 El lenguaje de Python
1.5 Organización del libro

Al igual que los hijos biológicos de generaciones anteriores, las máquinas representan la mejor esperanza de la humanidad para un futuro a largo plazo.

Moravec

Objetivos

En este capítulo aprenderemos los conceptos básicos de la arquitectura de una computadora típica y de cómo se controlan los datos y se usan las instrucciones para realizar las aplicaciones diseñadas por los usuarios.

1.1 Introducción

Los primeros años del siglo 21 han visto un crecimiento sorprendente en el uso y creación de nuevas computadoras. Esto requiere una gran cantidad de expertos que sepan afrontar los retos de dicho crecimiento. Estos expertos deben conocer entre otras cosas lenguajes de programación modernos que permitan que una computadora ejecute aplicaciones, que tenga acceso a Internet, que realice cálculos matemáticos complicados, que comunique con colegas y amigos en cualquier parte del mundo, entre otras actividades que realiza un usuario con una computadora. Este libro aborda el lenguaje de programación Python a nivel introductorio. Sin embargo, el lector será capaz de realizar gráficas, decisiones, manipular archivos y ver como Python se puede usar en distintas aplicaciones.

1.2 Evolución de la computadora

La historia de la computación se remonta muchos siglos atrás. El instrumento de cálculo más antiguo que se conoce es el ábaco el cual fue usado en la antigua China. En el siglo 17 Pascal y Leibnitz desarrollaron calculadoras mecánicas que son las antecesoras de las calculadoras electrónicas de nuestros días.

El primer dispositivo que se considera una computadora fue propuesto por Charles Babbage en 1830; véase la figura 1.1. Un excéntrico matemático inglés, Babbage deseaba automatizar el cálculo de tablas matemáticas. Para este propósito diseñó la Máquina de Diferencias. Aunque nunca la pudo construir totalmente, la Máquina de

Diferencias contenía las partes fundamentales de una computadora moderna, como son la memoria para datos e instrucciones, una unidad de entrada/salida y la posibilidad de realizar decisiones en la secuencia de instrucciones.

Figura 1.1 Charles Babbage.

Con el paso del tiempo surgieron las grandes compañías como IBM, entre otras, que se dedicaban a diseñar, producir y vender equipo de cómputo usando dispositivos electromecánicos. Fundada en 1911 con el nombre de Computing Tabulating Recording Corporation que se cambió luego a International Business Machines, IBM, se dedicaba a realizar calculadoras electromecánicas.

Es hasta 1937 cuando Howard Aiken propuso la Calculadora Automática Controlada Secuencialmente (Automatic Sequence Controlled Calculator). La construcción de esta máquina se inició en 1939 y se terminó en 1944 y se conoció como Mark I.

Esta máquina era principalmente electromecánica y se usaba para cálculos de trayectorias de misiles por el ejercito de los Estados Unidos. Para acelerar los cálculos, la Mark I tenía varios procesadores en paralelo controlados por una unidad de control.

En la misma época en la Gran Bretaña se daba el desarrollo de la computadora Colossus bajo la guía de Alan Turing. Esta máquina contribuyó a la decodificación de la máquina Enigma la cual era usada por los alemanes durante la Segunda Guerra Mundial para codificar sus mensajes. Desafortunadamente, el gobierno de la Gran Bretaña ordenó su destrucción al final de la guerra.

Mientras tanto, los usuarios de Mark I se dieron cuenta de la necesidad de una máquina más poderosa y rápida por lo que el ejército de los Estado Unidos contrató con la Universidad de Pennsylvania para desarrollar la computadora ENIAC, considerada como la primera computadora electrónica ya que estaba formada por bulbos electrónicos principalmente. El asesor matemático de este proyecto fue John von Neumann a quien muchos investigadores de la computación consideran el más importante personaje en la historia de la computación.

Aunque ENIAC fue basicamente una versión electrónica de la Mark I, la rapidez de la electrónica eliminó la necesidad de los procesadores en paralelo[1], además almacenaba en memoria la secuencia de instrucciones.

Muchos desarrollos siguieron a estas máquinas pioneras de la computación, pero las bases de las computadoras modernas estaban ya trazadas. Compañías como IBM,

[1] Las computadoras actuales realizan cálculos en paralelo para aumentar la velocidad de cómputo.

Digital, Sperry Rand, Control Data Corporation (CDC), Data General, Apple, HP, se erigieron como fabricantes de computadoras, cada vez más rápidas, poderosas y compactas.

Las computadoras de aquellos años pertenecían a lo que se conoce como **Main Frames**. Dicho término se aplica a aquellas computadoras que dan servicio a grandes corporativos, oficinas gubernamentales, bancos, entre otros. Algunos fabricantes de este tipo de computadoras fueron IBM y Control Data Corporation.

En 1960 se creó la primera minicomputadora. Se consideraba que una minicomputadora era una computadora que tenía un precio menor a $25,000.00 dólares. Es la compañía Digital Equipment Corporation (DEC) que produjo la primera minicomputadora PDP-8, a la que le siguieron otras como la PDP-11 y la VAX1130. Otras compañías también produjeron minicomputadoras como es el caso de Data General y Wang Laboratories.

En la década de los 1970 se presentaron las primeras redes de computadoras en Estados Unidos y en Europa.

Es en la década de 1970 cuando empezaron a surgir las computadoras de escritorio que se podían conectar a un televisor para que funcionara como monitor. Estas computadoras tenían una memoria limitada. Las compañías más conocidas eran Tandy Corp. y Commodore. En esa misma década nació Apple Computer Inc., fabricante de las computadoras Macintosh.

Los últimos años han visto un aumento en la velocidad de las computadora así como una reducción en el costo y tamaño. Además, el Internet se ha puesto al servicio del público en general a través de tabletas y teléfonos inteligentes que hacen que las computadoras estén presentes en todos los aspectos de la vida moderna a tal grado que es imposible imaginarnos la vida sin computadoras y sin Internet.

Tabla 1.1 Crónica de la computación

Año	Nombre	Hecha por	Comentarios
1830	Máquina Analítical	Babbage	Primer intento de construir una computadora digital.
1936	Z1	Zuse	Primera máquina calculadora con relevadores.
1943	Colossus	Gobierno Británico	Primera computadora electrónica.
1944	Mark I	Aiken	Primera computadora de propósito general.
1946	ENIAC I	Eckert/Mauchley	Primera computadora de la historia moderna.
1949	EDSAC	Wilkes	Primer programa almacenado en la computadora.

Continúa

Tabla 1.1 Continuación

Año	Nombre	Hecha por	Comentarios
1951	Whirlwind I	M.I.T	Primer programa en tiempo real.
1952	IAS	Von Neumann	La mayoría de las computadoras usan este diseño.
1960	PDP-1	DEC	Primera minicomputadora (se vendieron 50).
1961	1401	IBM	Máquina para negocios muy popular.
1962	7094	IBM	Dominó el cómputo científico en los 60's.
1963	B5000	Burroughs	Primera máquina en usar lenguaje de alto nivel.
1964	360	IBM	Primera familia de computadoras.
1964	6600	CDC	Primera supercomputadora.
1965	PDP-8	DEC	Primera minicomputadora popular.
1970	PDP-11	DEC	Minicomputadora más vendida en los 70s.
1974	8080	Intel	Primera computadora de 8 bits en un circuito integrado.
1974	CRAY-1	Cray	Primera supercomputadora.
1975	Microsoft Corp.		Bill Gates y Paul Allen fundan Microsoft Corp.
1976	Apple Computer Inc.		Steve Jobs y Steve Wozniak son los fundadores.
1978	VAX	DEC	Superminicomputadora de 32 bits.
1981	IBM PC	IBM	Empieza la era de la computadora personal (PC).
1981	Osborne-1	Osborne	Primera computadora portátil.
1983	Lisa	Apple	Primera computadora personal con interfase gráfica y mouse.
1984	Macintosh	Apple	Primera computadora para todo público.
1985	386	Intel	Primer circuito integrado de 32 bits.
1985	MIPS	MIPS	Primera máquina RISC.
1987	SPARC	Sun	Primera estación de trabajo.
1990	WWW	Tim Berners-Lee	En el CERN de Ginebra se crea la World Wide Web.
1991	Python	Guido von Rossum	Lanzamiento del lenguaje.
1992	Alpha	DEC	Primera computadora personal de 16 bits.

Continúa

Tabla 1.1 Continuación

Año	Nombre	Hecha por	Comentarios
1993	Newton	Apple	Primera computadora del tipo Palmtop.
2001	PowerBook G4	Apple	Línea de notebooks de Apple.
2010	iPad	Apple	Hace el acceso a Internet y Apps muy sencillo.

1.3 Arquitectura de una computadora

Una computadora está formada por 5 partes principales. Estas partes son:

- Unidad de memoria. Aquí se guardan los datos y las instrucciones. Tipicamente es un disco duro, una memoria RAM, o una unidad externa como un disco duro o una memoria USB.
- Unidad de entrada. La parte por donde se reciben los datos y las instrucciones de la aplicación. Puede ser un teclado o un archivo de datos o de instrucciones.
- Unidad de salida. Es la sección por donde se despliegan los resultados de la aplicación. Puede ser una pantalla, un listado, una hoja de cálculo o un archivo de datos.
- Unidad Lógica Aritmética (Arithmetic Logic Unit, ALU). Es la parte encargada de realizar las operaciones aritméticas, lógicas y de decisión de acuerdo al diseño de la aplicación. Junto con la unidad de control forma la Unidad de Procesamiento Central o CPU (Central Processor Unit).
- Unidad de control. Es la parte encargada de decodificar las instrucciones y realizar las operaciones que se hayan programado en la aplicación. Junto con la Unidad Lógica Aritmética forman el CPU (central processor unit).

La manera en cómo están interconectadas estas partes se conoce como la arquitectura de la computadora. Una arquitectura básica típica se muestra en la figura 1.2. Las diferentes unidades están interconectadas por el **bus de datos** y por el **bus de instrucciones**. Por el bus de datos se transmiten los datos entre las distintas unidades de la computadora. Por el bus de instrucciones se transmiten las instrucciones que se decodifican para efectuar las operaciones programadas en la aplicación.

Figura 1.2 Arquitectura básica de una computadora.

1.3.1 Instrucciones y datos de una computadora

Una computadora necesita de instrucciones para realizar una tarea dada por el usuario. Estas instrucciones, así como los datos, están escritas en una notación binaria. De esta manera, una computadora realiza todas las operaciones necesarias usando el sistema numérico binario o de base 2. La siguiente sección explica en detalle este sistema numérico.

1.3.2 Sistemas numéricos decimal y binario

Para explicar el sistema binario primero explicamos el sistema numérico decimal o de base 10. Consideremos un número como el 2943. En este número tenemos cuatro dígitos decimales. El de mayor valor es el 2 que corresponde a los millares o 10^3. Sigue el nueve que corresponde a las centenas o 10^2. El 4 corresponde a las decenas o $10 = 10^1$. Finalmente el 3 corresponde a las unidades $1 = 10^0$. Cada uno de estos dígitos decimales ocupa una posición. Las posiciones se numeran del cero en adelante. Empezando con las unidades que ocupan la posición 0, las decenas la posición 1, las centenas la posición 2, y finalmente, los millares la posición 3. Otra manera de escribir este número es desglosando cada uno de ellos usando potencias de diez de la siguiente manera :

$$2943 = 2 \times 10^3 + 9 \times 10^2 + 4 \times 10^1 + 3 \times 10^0$$

Es decir, según la posición del dígito decimal, este se multiplica por una potencia de

10 elevada a un exponente que depende de la posición. De esta manera, el dígito de la izquierda corresponde a la tercera potencia de 10 mientras que el dígito de la derecha corresponde a la potencia cero de 10. Esta es la razón del nombre sistema decimal o de base 10.

Por lo que respecta a números fraccionarios, las posiciones se numeran de izquierda a derecha a partir de la posición uno (no hay posición cero) que es la de la izquierda y aumentan conforme se mueve hacia la derecha. La otra diferencia es que las potencias de diez son con exponente negativo. Por ejemplo, consideremos el número decimal 0.8162. Este número se puede escribir como:

$$\begin{aligned} 0.8162 &= 8 \times 10^{-1} + 1 \times 10^{-2} + 6 \times 10^{-3} + 2 \times 10^{-4} \\ &= 0.8 + 0.01 + 0.006 + 0.0002 \end{aligned}$$

En los dos ejemplos anteriores notamos dos cosas. La primera es que los números decimales están formados por una combinación de los dígitos del 0 al 9. La segunda es la posición de los dígitos. Para el caso de la parte entera la posición del dígito de la derecha, también llamado dígito menos significativo, es la posición 0 y conforme la posición se mueve hacia la izquierda decimos que la posición se incrementa hasta llegar al último dígito, que se conoce como el dígito más significativo. El nombre de dígito más significativo se da ya que es el que corresponde a los millares, mientras que el dígito menos significativo solamente se refiere a las unidades. Si el bit menos significativo cambia, por ejemplo de 1 a 2, solamente cambia en una unidad. Por otro lado, si el bit más significativo corresponde a los millares y cambia de 5 a 6, el cambio es entonces muy grande en el resultado final.

Para la parte decimal el dígito de la izquierda es el más significativo y corresponde a las décimas, el siguiente corresponde a las centésimas y así sucesivamente hasta llegar al último dígito que corresponde al dígito menos significativo de la parte decimal.

Para el caso de los números binarios, estos están formados por los dígitos 0 y 1. Reciben el nombre de **bit** que es la abreviatura de ***bi***nary digi***t***. Para entender como funcionan los números binarios examinemos el número binario 1011. Para diferenciar entre números decimales y binarios añadimos un subíndice, 10 para números decimales y 2 para números binarios, al final del número. Entonces, la representación de 10101_2 es:

$$\begin{aligned} 10101_2 &= 1 \times\ 2^4 + 0 \times\ 2^3 + 1 \times\ 2^2 + 0 \times\ 2^1 + 1 \times\ 2^0 \\ &= 1 \times\ 16 + 0 \times\ 8 + 1 \times\ 4 + 0 \times\ 2 + 1 \times\ 1 \\ &= 16 + 0 + 4 + 0 + 1 \\ &= 21_{10} \end{aligned}$$

Es decir, el número binario 10101_2 corresponde al número decimal 21_{10}. Para el caso de un número binario fraccionario, por ejemplo, el número 0.11101_2 tenemos la siguiente representación:

$$
\begin{aligned}
0.11101_2 &= 1\times\ 2^{-1} + 1\times\ 2^{-2} + 1\times\ 2^{-3} + 0\times\ 2^{-4} + 1\times\ 2^{-5} \\
&= 1\times 0.5 + 1\times 0.25 + 1\times 0.125 + 0\times 0.0625 + 1\times 0.03125 \\
&= 0.5 + 0.25 + 0.125 + 0 + 0.03125 \\
&= 0.90625_{10}
\end{aligned}
$$

La tabla 1.2 nos muestra números decimales y su equivalente en binario para números decimales enteros. Para números decimales fraccionarios, la tabla 1.3 muestra los equivalentes.

Tabla 1.2 Números enteros en decimal y binario

Decimal	Binario
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Tabla 1.3 Números fraccionarios en decimal y binario

Decimal	Binario
0	0
0.5	0.1
0.25	0.01
0.125	0.001
0.0625	0.0001
0.75	0.11

La representación anterior nos permite convertir un número binario a número decimal. Por otro lado, para convertir un número entero a binario usamos el siguiente método.

1. Dividir el número decimal entre 2. El residuo es el bit menos significativo.
2. El cociente del resultado anterior dividirlo entre 2. El residuo es el bit siguiente.
3. Se continúa repitiendo el paso 2 hasta terminar. El bit más significativo es el último cociente.

Por ejemplo, consideremos el número 73_{10}.

1. El resultado del paso 1 es $73/2 \rightarrow$ cociente $= 36$, residuo 1, por lo que 1 es el bit menos significativo.
2. El resultado de $36/2 \rightarrow$ cociente $= 18$, residuo $= 0$.
3. El resultado de $18/2 \rightarrow$ cociente $= 9$, residuo $= 0$.
4. El resultado de $9/2 \rightarrow$ cociente $= 4$, residuo $= 1$.
5. El resultado de $4/2 \rightarrow$ cociente $= 2$, residuo $= 0$.
6. El resultado de $2/2 \rightarrow$ cociente $= 1$, residuo $= 0$.

El número binario es entonces:

$$1001001_2$$

Para convertir un número decimal fraccionario a binario fraccionario usamos el siguiente método:

1. Se multiplica la parte fraccionaria por dos.
2. Si el resultado es mayor que la unidad, el bit más significativo es 1.
3. Si el resultado es 0 el bit más significativo es 0.
4. Se toma la parte fraccionaria del resultado y se repite el proceso hasta terminar.
5. El proceso se termina cuando se alcanza el número máximo de bits permitido.

Como un ejemplo consideremos el número 0.3457_{10} el cual queremos convertir a base 2 con un máximo de 8 bits. Usando el método descrito anteriormente tenemos:

1. $0.3457 \times 2 = 0.6914 \leftarrow$ bit más significativo es 0.
2. $0.6914 \times 2 = 1.3828 \leftarrow$ bit de la posición 2 es 1.
3. $0.3828 \times 2 = 0.7656 \leftarrow$ bit de la posición 3 es 0.
4. $0.7656 \times 2 = 1.5312 \leftarrow$ bit de la posición 4 es 1.
5. $0.5312 \times 2 = 1.0624 \leftarrow$ bit de la posición 5 es 1.
6. $0.0624 \times 2 = 0.1248 \leftarrow$ bit de la posición 6 es 0.
7. $0.1248 \times 2 = 0.2496 \leftarrow$ bit de la posición 7 es 0.
8. $0.2496 \times 2 = 0.5992 \leftarrow$ bit de la posición 0 es 0.

Entonces el número en binario es 0.01011000_2. Notamos que la parte fraccionaria podría tener más bits pero solamente nos limitamos a lo requerido que era 8 bits. Al reconvertir el resultado a decimal obtenemos que:

$$0.01011000_2 = 0.34375_{10}$$

lo que nos indica que necesitamos un mayor número de bits para representar mejor el número decimal 0.3457_{10}. Por ejemplo si extendemos a 16 bits obtenemos:

$$0.0101100001111111_2 = 0.3456878662109375_{10}$$

De este resultado vemos que entre más bits usemos para representar un número decimal en el sistema binario, mejor aproximación se obtiene.

1.3.3 Otros sistemas numéricos

Existen otros sistemas numéricos, además del sistema binario, que encuentran uso en los sistemas computacionales. Uno de ellos es el **sistema hexadecimal**. Este sistema tiene 16 caracteres para representar los números. Los caracteres son:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Para representar estos caracteres en el sistema binario necesitamos agrupar en grupos de cuatro bits y a cada grupo se le asigna un número decimal del 0 al 9 y las letras A, B, C, D, E, F si corresponde a valores de 10 a 15. Si la representación en binario no tiene paquetes de cuatro bits se acompleta con ceros. Por ejemplo el número decimal 89 se presenta en binario como:

$$89_{10} = 011001_2 = 00011001_2 = 19_{16}$$

donde hemos acompletado con ceros para tener dos bloques de cuatro bits.

Otro sistema numérico en uso en las computadoras es el **sistema octal**. En este sistema, los bits de la representación binaria se agrupan en bloques de tres bits y a cada bloque se le asigna un número decimal. Si no hay grupos de 3 bits se acompleta con ceros. Por ejemplo, el número 23_{10} tiene la representación en binario:

$$23_{10} = 10111_2$$

Lo acompletamos con un cero a la izquierda para tener dos bloques de tres bits. Entonces, tenemos:

$$23_{10} = 010\ 111_2$$

Como $010_2 = 4_{10}$ y $111_2 = 7_{10}$

Por lo que tenemos:

$$23_{10} = 010111_2 = 47_8$$

De la misma manera tenemos:

$$23_{10} = 00010111_2 = 17_{16}$$

La tabla 1.4 nos muestra los códigos de estos caracteres.

Tabla 1.4 Caracteres hexadecimales

Hexadecimal	Decimal	Binario	Octal
0	0	0	0
1	1	1	1
2	2	10	2
3	3	11	3
4	4	100	4
5	5	101	5
6	6	110	6
7	7	111	7
8	8	1000	10
9	9	1001	11
A	10	1010	12
B	11	1011	13
C	12	1100	14
D	13	1101	15
E	14	1110	16
F	15	1111	17

1.3.4 Estructura de la memoria

La memoria de una computadora, ya sea un disco duro, una memoria flash o caché, una memoria ROM, un USB, o cualquier otro tipo de memoria, está organizada como se muestra en la figura 1.3. Cada espacio corresponde a una localidad de memoria que en esta figura está numerada usando el sistema binario. Por lo general, existe un espacio de la memoria dedicado a almacenar instrucciones y otro espacio para almacenar datos del programa o algoritmo que se desea ejecutar en la computadora.

Nótese que en esta figura ilustrativa, las direcciones de memoria ocupan hasta cuatro bits, pero en computadoras modernas pueden ocupar hasta 64 bits y en ocasiones

ocupan hasta 128 bits, para lo que emplean dos localidades de memoria. Los datos o instrucciones se almacenan dentro de la memoria con una cierta longitud de palabra. A la longitud o número de bits de la información almacenada en una localidad de memoria se le conoce como **palabra**.

Una manera de representar la longitud de una palabra es por medio de bytes. Cada byte tiene una longitud de 8 bits. De esta manera una palabra de 64 bits requiere 8 bytes. Aunque el byte ya es una longitud muy pequeña, el byte se sigue usando como la unidad básica de la longitud de palabra.

El tamaño de una memoria se mide en miles, millones o miles de millones de bytes. Dado que las computadoras usan el sistema binario, la base 2 es la que se usa para contar los bytes. La cantidad más cercana a 1000 es 2∧10 que es igual a 1024 bytes y que corresponde a la cantidad de bytes en un kilobyte. De manera similar un megabyte = 2∧20 = 1,048,576. Entonces tenemos lo siguiente:

Unidad	**Número de bytes**
Kilobyte	$2^{10} = 1024$
Megabyte	$2^{20} = 1,048,576$
Gigabyte	$2^{30} = 1,073,741,824$
Terabyte	$2^{40} = 1,099,511,627,776$

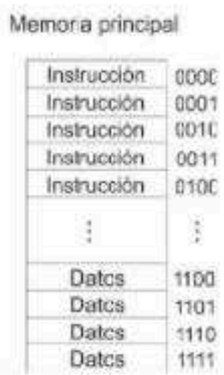


Figura 1.3 Estructura simplificada de una memoria. Se muestra a la derecha la dirección de cada localidad de memoria.

1.3.5 Lenguajes de una computadora

Como lo mencionamos en la sección 1.3.1, las computadoras manejan sus datos o instrucciones en forma de números binarios, un formato que es imposible de usar por el usuario de una computadora y que se conoce como **lenguaje de máquina**. Para resolver este problema, se han desarrollado lenguajes de programación fáciles para el usuario y que indican a la computadora lo que se desea realizar.

Para programar una computadora usamos lenguajes de alto nivel como Python, C, C++, MATLAB, Java, entre otros muchos existentes.

Al lenguaje que el usuario escribe se le llama **código fuente** (source code). Para convertir el código fuente a lenguaje de máquina se usa otro programa que se conoce como el compilador. Este compilador es el encargado de realizar la conversión a lenguaje de máquina y el resultado es un archivo que se conoce como **código objeto** (object code). Los compiladores producen adicionalmente un **archivo ejecutable** que se puede ejecutar independientemente del lenguaje de programación utilizado. Otros lenguajes de programación compilan el archivo fuente instrucción por instrucción. A este tipo de compiladores se les conoce como **intérpretes**.

Una tarea que realiza el compilador es checar la sintaxis del código fuente. Si existen errores, la compilación indica todos los errores que se tengan. Si no existen errores se crea el archivo ejecutable para poder ser usado por el usuario. En el caso de los intérpretes, al ir compilando una instrucción a la vez, cuando se encuentra un error de sintáxis, se interrumpe el proceso hasta que se corrija el error. La desventaja es que si se tienen 10 errores, se tendrían que realizar 10 correcciones y 10 compilaciones por el intérprete en lugar de una sola.

1.4 El lenguaje Python

Python es un lenguaje de programación de computadoras que nació en 1991 y que ha ido ganando adeptos por dos razones principales. La primera es que es un lenguaje de alto nivel muy fácil de usar y la segunda es que es código libre. Fue concebido y desarrollado por Guido von Rossum y la primera versión, la version 0.9.0, estuvo disponible en 1991. Actualmente, las versiones más usadas son la versión 2 y la versión 3. Este libro trata sobre la versión 3.

Una de las grandes ventajas de Python es que es independiente de la plataforma que se use, ya sea Windows, Mac o Unix. Los programas escritos en una plataforma corren en las otras plataformas. El único requisito es que haya sido realizado en la misma versión.

Python tiene disponible una biblioteca estándar que le da al usuario una gran cantidad de funciones para escribir código fuente muy simple pero poderoso. Además, otros usuarios han desarrollado módulos, la mayoría de los cuales son de código abierto (open source) y gratuitos. Estos módulos se pueden usar para desarrollar interfases gráficas, interfases a bases de datos, aplicaciones de páginas de Internet, para mencionar unas cuantas.

1.4.1 Descarga de Python

Para obtener Python solamente es necesario descargarlo de la página oficial de la Fundación Python (Python Software Foundation) en **www.python.org**. La versión que este libro usa es la más reciente de las versiones 3.x (3.5 a finales de 2016).

Para instalarlo siga las intrucciones que se indican. Una vez terminada la instalación, Python está listo para utilizarse.

1.4.2 La interfase de Python

Al correr Python se abre la ventana principal o Shell que se denomina la **IDLE** y que es la abreviatura de **I**ntegrated **D**eve**L**opment **E**nvironment o Ambiente Integrado de Desarrollo. Este ambiente se muestra en la figura 1.4. Una vez que se muestran los símbolos >>>, Python está listo para usarse.

Figura 1.4 IDLE de Python.

En esta ventana podemos ver que la versión de Python es una versión de 64 bits de longitud de palabra. En Mac la versión es de 64 bits.

En los capítulos siguientes se presenta como usar Python para realizar operaciones sencillas, calcular funciones, usar bibliotecas y realizar algoritmos. En el libro veremos como Python se puede usar para realizar algoritmos más poderosos así como diferentes aplicaciones.

1.5 Organización del libro

El libro está organizado para ser usado por estudiosos con nada o poco conocimiento de la programación. Para aquellos que ya tienen conocimiento de algún lenguaje de programación, el estudio es más rápido y los ejemplos les ayudarán a rapidamente adquirir un conocimiento de las instrucciones de Python. En ambos casos el libro les permite implementar algoritmos que tomen decisiones, que realicen iteraciones, que lean y escriban datos en archivos y que grafiquen resultados, entre otras cosas.

En el capítulo 2 se realizan operaciones sencillas en el IDLE. Se definen los conceptos de algoritmo y pseudocódigo. Se muestran realizaciones de algoritmos en pseudocódigo y su conversión a código en Python.

El capítulo 3 cubre el concepto de condiciones para ejecutar una instrucción o un conjunto de instrucciones. En un algoritmo casi siempre hay que tomar una decisión de que instrucción ejecutar dependiendo de los valores de una o varias de las variables. Es este el concepto de condición. Se usa el lenguaje de pseudocódigo para plantear el algoritmo y su posterior realización en Python.

En algunos casos es necesario repetir un conjunto de instrucciones un determinado número de veces, y en ocasiones un número indeterminado de veces. El capítulo 4 trata sobre ciclos. Cada algoritmo también se describe primero en pseudocódigo y se implementa después en Python.

El libro continúa con un tratamiento de vectores y matrices para continuar con la creación de funciones.

Como obtener gráficas cubre en el capítulo 11 y se usa la biblioteca **numpy** que se descarga de la página **www.python.org**. Con esta biblioteca se pueden realizar gráficas desde Python.

La última parte cubre aplicaciones de Python en matemáticas e ingeniería.

Capítulo 2
Fundamentos de algoritmos y de Python

2.1 Introducción
2.2 ¿Qué es un algoritmo?
2.3 Pseudocódigo
2.4 Variables
2.5 Partes de un algoritmo
2.6 Algoritmos en pseudocódigo
2.7 Lenguaje Python
2.8 Estructura de un algoritmo en Python
2.9 Variables
2.10 Bibliotecas y encabezados
2.11 Operadores
2.12 Comentarios
2.13 Entrada y salida de datos en Python
2.14 Algoritmos sencillos
2.15 Variables alfanuméricas en Python
2.16 Listas
2.17 Instrucciones de Python del Capítulo 2
2.18 Conclusiones
2.19 Ejercicios

Ves cosas y dices "¿Por qué?" Pero yo sueño cosas que nunca fueron y digo, "¿Por qué no?".

George Bernard Shaw

Objetivos

Conocer el concepto de algoritmo, su concepción, su descripción por medio de un lenguaje conocido como pseudocódigo. Así mismo su realización por medio del lenguaje Python.

2.1 Introducción

En este capítulo damos los conceptos básicos de algoritmos para usarse en la solución de problemas. Nuestros primeros algoritmos los resolvemos usando pseudocódigo. Una vez que está realizado el razonamiento y planteada la solución del problema, procedemos a codificar el algoritmo en lenguaje de Python. Mostraremos el procedimiento con ejemplos ilustrativos de la sintaxis del pseudocódigo y de Python.

2.2 ¿Qué es un algoritmo?

Un algoritmo es una manera de resolver un problema. Ejemplos de algoritmos son las recetas de cocina donde el problema que se resuelve es la preparación de un platillo. En las recetas se nos dice que ingredientes se requieren, la forma en que se deben mezclar los ingredientes así como los tiempos de cocción para obtener una comida sabrosa y saludable. Otro ejemplo lo constituye una reparación de un automóvil por un mecánico. En este caso, el mecánico debe realizar un procedimiento en el que se debe incluir un diagnóstico o análisis, la compra de las partes a reemplazar (si es que esto es necesario) y armar y desarmar en un orden preciso. Finalmente, se debe realizar una prueba del funcionamiento del vehículo para comprobar que la reparación fue exitosa. Con estos ejemplos en mente podemos ahora dar una definición más precisa de lo que es un algoritmo enunciándolo de la siguiente manera:

Un algoritmo es una secuencia precisa de pasos que nos permite alcanzar un resultado o resolver un problema.

Los pasos necesarios para realizar un algoritmo son:

1. Análisis del problema.
2. Diseño del algoritmo para resolverlo.
3. Verificación del algoritmo.
4. Implementación del algoritmo en algún lenguaje de programación.

Un ejemplo de algoritmo lo presentamos para un postre.

Ejemplo 2.1

Algoritmo para preparar un flan de caramelo

Para preparar un flan de caramelo se necesita una serie de pasos que aseguren una postre de buen sabor y con buena presentación. Primero debemos tener los ingredientes disponibles:

Ingredientes:

- 120 gramos de chocolate,
- 1.5 cucharadita de agua,
- 1 lata de leche evaporada,
- 1 lata de leche condensada,
- 4 huevos,
- 1 cucharadita de vainilla,
- 1/4 cucharadita de sal,
- 1/4 extracto de almendra.

Podemos ahora enumerar los pasos del algoritmo de la siguiente manera:

1. Calentar el horno a 350°F.
2. Poner 48 gramos de chocolate junto con el agua, y poner en baño maría hasta que se derrita el chocolate.
3. Vierta la mezcla en un molde para flan para que se cubra el fondo.
4. Vierta la leche evaporada en una taza medidora. Agregue agua hasta llegar a una taza y cuarto.
5. Coloque en un vaso de licuadora una taza de leche condensada, los huevos, la vainilla, el extracto de almendra y la sal.
6. Derretir a fuego lento el chocolate restante en la leche restante.
7. Agregarlo al vaso de la licuadora y licuar.
8. Tapar muy bien con una hoja de aluminio y colocar a baño maría en otro molde de mayor tamaño por dos horas
9. Esperar a que se enfríe.
10. Desmoldar al terminar.
11. Servir y disfrutar.

Notamos que los pasos se deben realizar en el orden indicado para poder llegar al objetivo planteado. De esta manera el algoritmo para preparar un flan lo podemos enunciar de la siguiente manera:

Algoritmo: Preparar un flan.
Entrada: 120 gramos de chocolate, 1.5 cucharadita de agua, 1 lata de leche evaporada, 1 lata de leche condensada, 4 huevos, 1 cucharadita de vainilla, 1/4 cucharadita de sal, 1/4 extracto de almendra;
Salida: flan;
INICIO

1. Calentar el horno a 350°F.
2. Poner 48 gramos de chocolate junto con el agua y poner en baño maría hasta que se derrita el chocolate.
3. Vierta la mezcla en un molde para flan para que se cubra el fondo.
4. Vierta la leche evaporada medidora. Agregue agua hasta llegar a una taza y cuarto.

5. Coloque en un vaso de licuadora una taza de leche condensada, los huevos, la vainilla, el extracto de almendra y la sal.
6. Derretir a fuego lento el chocolate restante en la leche restante.
7. Agregarlo al vaso de la licuadora y licuar.
8. Tapar muy bien con una hoja de aluminio y poner a baño maría en otro molde de mayor tamaño por dos horas
9. Esperar a que se enfríe.
10. Desmoldar al terminar.
11. Servir y disfrutar.

FIN

De manera similar a este ejemplo, podemos realizar algoritmos para resolver problemas de matemáticas numéricas o de otras áreas de la ingeniería, la computación, la física, las matemáticas, las finanzas, la biología, la medicina, las ciencias, etc.

Hoy en día, el uso de la palabra algoritmo se refiere a cualquier procedimiento que permita resolver un problema dado.

La palabra algoritmo es en honor del matemático árabe **Al Khuarismi** al que se le atribuye el primer algoritmo para la obtención de raíces de una ecuación.

2.3 Pseudocódigo

El pseudocódigo es una manera de representar las instrucciones de un algoritmo sin recurrir a ningún código de programación en particular. Para empezar, vamos a definir los componentes y las operaciones básicas de asignación de variables, entrada y salida de datos, y tipos de variables.

La ventaja de describir un algoritmo en pseudocódigo es que es más fácil escribir un algoritmo de lenguaje natural a pseudocódigo y de ahí se puede codificar más facilmente en Python o en cualquier otro lenguaje de programación.

2.4 Variables

Una variable es un dato que puede tomar distintos valores según el algoritmo. Las variables de un algoritmo tienen un nombre para identificarlas en el algoritmo. El nombre es una cadena de caracteres alfanuméricos que debe empezar con una letra de la A a la Z y pueden ser mayúsculas o minúsculas. Dentro del nombre puede haber números o guiones bajos. Es muy recomendable que los nombres de las variables tengan un nombre que tenga relación con lo que representa. No puede haber dos variables con nombres iguales dentro de un algoritmo. Dado que una letra mayúscula es distinta de una minúscula, la variable *A1* es distinta de la variable *a1*.

En todos los lenguajes existe un conjunto de palabras reservadas llamadas palabras clave que no se pueden utilizar como nombres de variables. En el transcurso del libro iremos conociendo muchas de las palabras reservadas de Python.

Los nombres de variables deben empezar con una letra y solamente pueden contener letras, números y guiones bajos.

2.4.1 Tipos de variables

En Python, así como en la mayoría de los lenguajes de programación, cada variable corresponde a un tipo de variable. Esto quiere decir que una variable puede ser de tipo entera, real, alfanumérica o lógica.

Una variable es entera si tiene valores que solamente tienen parte entera y **NO** tienen parte decimal. Por ejemplo, `a = 5` y `b = -7` son variables enteras.

Una variable real es aquella que **SÍ** tiene una parte decimal como las variables `b1 = -3.14` y `b2 = -17.23`. También son variables reales aquellas que incluyen una potencia como en `a_real` $= -1.23 \times 10^2$. A la parte fraccionaria de una variable real se le llama la mantisa.

Una variable alfanumérica es aquella que no tiene un valor numérico pero que es una cadena de caracteres alfanuméricos como `a = "universitario_2013"`. Las variables alfanuméricas se encierran entre comillas dobles o comillas como en el caso de la cadena `a`, o entre comillas simples, como en las cadenas `calificación = 'A'`, `valor = 'T'`.

Una variable lógica es aquella que solamente toma los valores `Verdadero` o `Falso`. Ejemplos de variables lógicas son:

```
x = Verdadero, aviso = Falso, b = Falso.
```

Las variables conservan su tipo durante la ejecución del algoritmo a menos de que nosotros cambiemos su tipo[1]. Es decir, una variable puede ser entera y la podemos redefinir para que sea de otro tipo en otra parte del algoritmo. Este es el caso de Python por lo que decimos que Python es un lenguaje con tipos dinámicos ya que se puede cambiar el tipo de las variables dentro del algoritmo.

Los **datos** son variables cuyos valores se dan al algoritmo por el usuario o por otro algoritmo. Al igual que las variables los datos pueden ser de tipo entero, real, alfanumérico y lógico.

2.4.2 Asignación de valores a variables

Supongamos que deseamos asignar el valor de 4 a una variable entera a. Esto lo hacemos con:

$$a \leftarrow 4;$$

De manera que a tiene ahora el valor de 4. Esto también lo podemos hacer si el valor de la variable $b = 8$ se desea almacenar en a. Esto lo hacemos con

$$a \leftarrow b;$$

Al hacer esto el valor de 4 que había en a se pierde y ahora a tiene el valor de 8 que es el valor de b. Nótese que en ambas asignaciones hemos terminado el renglón con un punto y coma ";" lo que indica que ahí se ha terminado la instrucción, pero esto no es necesario. El punto y coma se puede utilizar para separar instrucciones que escribimos en el mismo renglón.

2.4.3 Inicialización de variables

Una manera de inicializar una variable es darle valor desde el inicio del algoritmo. Esto lo podemos hacer al declarar las variables. Para el algoritmo que obtiene la solución de la ecuación de segundo grado podemos tener:

Variables:
Real: a ← 1, b ← 2, c ← 1, x1, x2, discr;

Esto quiere decir que el valor de a es 1, el valor de b es 2 y el valor de c es 1. Las variables x1, x2 y discriminante no se inicializan. Una manera alterna de definir una

[1]Algunos lenguajes, como Java y C++, no permiten cambiar el tipo de una variable.

variable e inicializar es:

Variables:
Real: a(1), b(2), c(1), x1, x2, discr;

2.4.4 Operaciones básicas

Las operaciones básicas se realizan de la manera tradicional. El resultado se asigna a la variable donde se almacena el resultado. Por ejemplo, para realizar la suma de las variables a y b y almacenar este resultado en la variable c usamos

c ← a + b;

Ejemplo 2.2

Algoritmo para resolver una ecuación de primer grado.

Para resolver una ecuación de primer grado

$$ax + b = 0$$

necesitamos despejar x. Esto nos lleva a que la solución está dada por

$$x = -\frac{b}{a} \tag{2.1}$$

un algoritmo para realizar esta ecuación puede ser:

Algoritmo: Solución de una ecuación de primer grado.

INICIO

Variables: a, b, x;
Recibir: a, b;
x ← −b/a;
Mostrar: x;

FIN

En este algoritmo hemos usado **Recibir** para la entrada de datos y **Mostrar** para la salida del resultado. Además, el principio y el final del algoritmo se indican con las palabras clave **INICIO** y **FIN**, respectivamente.

2.4.5 Operaciones con variables enteras

Las operaciones, con excepción de la división, que se realizan con variables enteras producen resultados enteros. Por lo tanto, tenemos que a y b son de tipo entero, los resultados de a + b, a − b y a * b son de tipo entero. Por ejemplo, si a = 2 y b = 5, tenemos

$$\begin{array}{lcr} a + b & \rightarrow & 7 \\ a - b & \rightarrow & -3 \\ a * b & \rightarrow & 10 \end{array}$$

El resultado de la división en Python nos da un número real con punto decimal:

$$a/b \rightarrow 0.4$$

Si deseamos obtener un resultado entero debemos usar doble símbolo de división:

$$a//b \rightarrow 0$$

Usando la división entera nuestro resultado sólo toma la parte entera y por lo tanto a//b produce el resultado 0 (cero).

2.4.6 Operaciones con enteros y reales

Cuando tenemos operaciones donde las variables son de los tipos entero y real, SIEMPRE se cambian las variables enteras a reales antes de hacer la operación. De la misma manera al guardar un resultado real en una variable que era entera, esta se cambia a tipo real.

Al almacenar un resultado real en una variable entera se convierte a variable real.

2.5 Partes de un algoritmo

De los ejemplos dados hasta ahora podemos ver que un algoritmo debe tener ciertas características. Estas son

- Debe ser preciso: no contener ambigüedades.
- Definido: los mismos datos producen el mismo resultado
- Finito: siempre termina.

Además, las partes en las que se estructura un algoritmo son:

- Entrada: información de partida.
- Procesos: operaciones y cálculos a realizar.
- Salida: resultados obtenidos.

Ilustramos con un ejemplo.

Ejemplo 2.3

Algoritmo para calcular el perímetro y la superficie de un triángulo

Para calcular el perímetro y la superficie de un triángulo dados sus lados l_1, l_2, l_3 usamos las ecuaciones:

$$P \leftarrow l_1 + l_2 + l_3$$
$$S \leftarrow \frac{P}{2}$$
$$superficie \leftarrow \sqrt{S(S-l_1)(S-l_2)(S-l_3)}$$

donde P es el perímetro y S es una variable intermedia. Un algoritmo que implemente estas ecuaciones es:

- Leer los lados del triángulo.
- Calcular el perímetro.
- Calcular la variable S.
- Calcular la superficie.
- Mostrar P y superficie.

2.6 Algoritmos en pseudocódigo

Para describir un algoritmo en pseudocódigo es necesario que se incluyan las siguientes partes:

1. Un título.
2. Es recomendable, pero no obligatorio, una muy breve descripción de lo que hace el algoritmo.
3. Recibir datos de entrada.
4. Realizar las operaciones necesarias.
5. Mostrar datos de salida.
6. Fin del algoritmo.

Se pueden incluir comentarios dentro del algoritmo para explicar, clarificar o recordar lo que hacen partes del algoritmo, así como para dar una explicación de lo que representan los datos y las variables. Los renglones de comentario empiezan con el símbolo numeral #.

Los renglones de comentario empiezan con el símbolo numeral #.
Los renglones de comentario NO se ejecutan.

A continuación explicamos nuestro primer algoritmo. Las palabras clave o reservadas se escriben en negritas.

Ejemplo 2.4

Algoritmo para escribir un texto.

En este primer algoritmo solamente deseamos escribir la frase:

Bienvenidos al diseño de algoritmos.

Esta frase es una cadena de caracteres y en el algoritmo no realizamos ningún cálculo. Entonces, nuestro algoritmo es:

Algoritmo: Escritura de un texto.

```
# Este algoritmo escribe un texto de bienvenida.
# En este algoritmo no hay datos constantes ni variables.

INICIO
    Mostrar: "Bienvenidos al diseño de algoritmos.";
FIN
```

Como vemos, el algoritmo no recibe datos ni realiza cálculos numéricos, solamente muestra un letrero. También contiene palabras clave (keywords).
El texto es una cadena de caracteres y se escribe entre comillas dobles o sencillas.

Las palabras clave son: **INICIO**, **Mostrar** y **FIN**, y se escriben en negritas.

Ejemplo 2.5

Solución de una ecuación de segundo grado

La ecuación de segundo grado proporciona un ejemplo de como se pueden resolver problemas más complejos usando un algoritmo. La ecuación de segundo grado está dada por

$$ax^2 + bx + c = 0$$

Y sus dos soluciones están dadas por

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.2}$$

Los pasos para diseñar un algoritmo que obtenga la solución de la ecuación de segundo grado son entonces:

1. Leer los coeficientes a, b y c.
2. Calcular la raíz cuadrada del discriminante $\sqrt{b^2 - 4ac}$.

3. Calcular los valores de x_1 y x_2.
4. Mostrar los valores de x_1 y x_2.

De esta manera el algoritmo en pseudocódigo es el siguiente (usamos la función raiz2 para la raíz cuadrada):

Nombre: Solución de una ecuación de segundo grado.
Este algoritmo encuentra la solución de una ecuación de segundo grado.
Se requieren los coeficientes a, b, c de la ecuación.
El algoritmo muestra los valores de x_1 y x_2.
INICIO
Variables: a, b, c, raíz_discriminante, x1, x2;
Se piden los valores de los coeficientes.
Mostrar: "Dame los valores de a, b y c";
Recibir: a, b, c;
# Se calcula la raíz cuadrada del discriminante.
raíz_discriminante = raiz2(b*b - 4*a*c)
# Ahora se calculan x1 y x2.
x1 = (−b + raíz_discriminante)/(2*a);
x2 = (−b − raíz_discriminante)/(2*a);
# Finalmente se despliegan los resultados:
Mostrar: "Los valores de x1 y x2 son: ", x1, x2;
FIN

Un ejemplo posterior nos presenta con otras opciones dependiendo de los valores del discriminante.

2.7 Lenguaje Python

En esta sección damos una introducción al lenguaje de programación Python. En este lenguaje estaremos convirtiendo nuestros algoritmos escritos en pseudocódigo a lenguaje Python para poder ejecutarlos en una computadora. La ventaja de usar Python como lenguaje de programación es que se puede usar en una gran cantidad de plataformas como PCs, microcontroladores, estaciones de trabajo y muchos sistemas operativos modernos.

Python fue creado por Guido van Rossum. Lo empezó a desarrollar a finales de 1989 y se considera hoy en día como el tercer lenguaje más ocupado por los desarrolladores. Además es de código abierto lo que lo hace accesible a un mayor número de estudiantes, investigadores y desarrolladores.

En este libro tratamos con el lenguaje de Python, al que nos referimos simplemente como Python.

2.7.1 El ambiente de Python

Python corre desde un ambiente de desarrollo integrado conocido como IDLE que son las iniciales de **I**ntegrated **D**eve**L**opment **E**nvironment. En este ambiente, Python puede ejecutar instrucciones como si se tratara de una calculadora o correr algoritmos previamente almacenados como scripts en un archivo de Python. Python no compila y ejecuta los algoritmos ya que es un intérprete el que va checando la sintaxis y el formato de cada instrucción y ejecutándola. El IDLE de Python se muestra en la figura 2.1. En este ambiente se definen las variables y se ejecutan instrucciones. Por ejemplo, si tenemos las variables a = 2.3, b = -7, c = a + b, tenemos entonces en el IDLE de Python:

```
>>> a = 2.3
>>> b = -7
>>> c = a + b
```

Figura 2.1 Ambiente de desarrollo de Python.

Figura 2.2 Ejemplo de cálculo en Python

Para poder escribir el resultado, debemos usar la instrucción `print` de la siguiente manera:

```
>>> print (c)
```

Para obtener el resultado:

```
-4.7
```

Lo que vemos en la figura 2.2. De la misma manera podemos realizar más cálculos en el IDLE de Python, por ejemplo, el cálculo del valor absoluto, la división, etc. Otras funciones más especializadas como las funciones trigonométricas, logarítmicas, raíces, la potenciación, etc. se encuentran en bibliotecas que deben ser cargadas o importadas antes de usarse. Una pequeña lista de funciones nativas de Python se muestra en la Tabla 2.1. En otras secciones describimos otras funciones elementales de Python.

Tabla 2.1 Funciones básicas

Función	Python	Significado
$\|x\|$	`abs(x)`	Valor absoluto de x
a^b	`pow(a, b)`	Potenciación
cociente de a/b	`a//b`	Obtiene el cociente entero de a/b
residuo de a/b	`a%b`	Obtiene el residuo de a/b
a/b	`divmod(a,b)`	Obtiene el cociente y el residuo de a/b

2.8 Estructura de un algoritmo en Python

Un algoritmo en Python tiene la misma estructura que hemos presentado para un algoritmo en pseudocódigo. Lo único que cambia son las palabras clave y algunos otros signos de sintaxis y puntuación.

Al igual que en pseudocódigo, un algoritmo empieza con un nombre, que en este caso no forma parte del código de Python y por lo tanto debe ser escrito como comentario.

La estructura de un algoritmo en Python está agrupada en un algoritmo principal. Los renglones de comentario deben principiar con un símbolo de numeral #. Para mostrar datos se usa la instrucción `print`. Para recibir datos se tiene la instrucción `input`.

Cuando empezamos a programar en una computadora es muy fácil pensar que la parte complicada es escribir el código que va a efectuar las instrucciones y operaciones necesarias para resolver un problema. Esto está lo más alejado de la realidad de lo que nos podemos imaginar. En realidad, la parte más difícil es plantear los pasos que nos van a llevar a resolver nuestro problema de una manera exitosa. Esta secuencia de pasos, en donde cada paso se realiza una serie de instrucciones, se conoce como un **algoritmo**. Esta secuencia de instrucciones recibe diferentes nombres dependiendo del contexto en el que estemos. Por ejemplo, en un restaurant se le da el nombre de **receta**. El chef debe seguir una secuencia de pasos que lo lleve a deleitar a los comensales con un platillo con buen sabor y buena presentación. Un entrenador deportivo proporciona una **rutina** para que los deportistas adquieran y mantengan una excelente condición física y además les da **instrucciones** a los jugadores para que realicen jugadas que los lleven a obtener la victoria. Esta manera de nombrar un algoritmo son solamente algunos de los ejemplos que podemos encontrar de algoritmos.

2.9 Variables

Las variables en Python son de los cuatro tipos: enteras, reales, alfanuméricas y lógicas, para las que se usan las palabras clave **int**, **float**, **chr** o **str**, y **bool**, respectivamente.

Cada variable debe tener un nombre el cual debe empezar con una letra y solamente puede tener una letras, números y guiones bajos. El nombre de una variable no puede tener acentos ni espacios.

Cada variable del algoritmo debe constar de tres partes: 1) el nombre o identificador, 2) un tipo, y 3) un valor. En Python, una variable puede cambiar de tipo dentro del mismo algoritmo. En un principio puede ser entera y más adelante pode-

mos redefinirla como cadena o como real. Por esto decimos que Python usa tipos dinámicos.

Una operación básica de un algoritmo es la asignación. Esta consiste en asignarle un valor a una variable, ya sea desde la definición de la variable o posteriormente. Para la asignación usamos el signo de igual =, por ejemplo:

```
>>> a = 3.4              # Define una variable flotante.
>>> b = 3                # Define una variable entera.
>>> coche = 'grande'     # Define una cadena o string.
>>> letra = 'a'          # Define un caracter.
```

Para saber que tipo de variable se trata usamos la instrucción **type** de la siguiente manera:

```
>>> type (coche)
class 'str'
```

Donde vemos que la variable `coche` es del tipo cadena o string. En las variables `a`, `b`, `coche` y `letra` hemos asignado valores a estas variables. Al mismo tiempo hemos designado el tipo de la variable.

Como otro ejemplo de asignación muy usado es el incremento de una variable. Este consiste en aumentar su valor en la unidad, es decir, que el nuevo valor de ella sea incrementado en la unidad. Tenemos entonces que si `a` es la variable, su nuevo valor sea:

```
a + 1
```

Esto lo realizamos en pseudocódigo con:

$$a \leftarrow a + 1$$

Que en Python es

```
a = a + 1;
```

Que quiere decir que el nuevo valor de `a` es el valor original de `a` más la unidad. Si `a` es igual a 10, el resultado final después de `a = a + 1` es `a = 11`. Esta asignación la usaremos más a menudo en el capítulo 4. De manera general, una variable recibe su valor de una expresión por lo que se tiene.

$$x = \text{expresión}$$

Por ejemplo,

```
x = 2*x + 3/2;
```

Tabla 2.2 Funciones de la biblioteca `math`

Función	Código	Ejemplo
Valor absoluto[2]	abs	math.abs(-3)
	fabs	math.fabs(-3)
Función exponencial	exp	math.exp(1)
Potencia a^b	pow(a, b)	math.pow(2, 3)
Raíz cuadrada	sqrt	math.sqrt(3.0)
Coseno de un ángulo en radianes	cos	math.cos(0.7)
Seno de un ángulo en radianes	sin	math.sin(0.707)
Tangente de un ángulo en radianes	tan	math.tan(1)
Conversión de radianes a grados	degree	degree(0.7)
Obtención de valor de pi	pi	math.pi
Obtención de valor de e	e	math.e
Arco seno de un valor (ángulo)	acos	math.acos(0.56)
Arco seno de un valor (ángulo)	asin	math.asin(0.7)

2.10 Bibliotecas y encabezados

Muy a menudo, Python requiere usar funciones que vienen en bibliotecas. Estas bibliotecas deben llamarse antes de usar las funciones de ella. Por ejemplo, las funciones trigonométricas, la raíz cuadrada, los logaritmos, entre otras, vienen incluidas en la biblioteca de funciones matemáticas denominada `math`. Para poder usar sus funciones en nuestro algoritmo usamos al principio del algoritmo lo siguiente:

```
>>> import math
```

Y para usar las funciones usamos, por ejemplo, para obtener el seno de $\pi/4$:

```
>>> math.sin(math.pi/4)
0.7071067811865475
```

Donde hemos usado la función `sin` y la constante π = `pi` que son parte de la biblioteca `math`. Otra forma de importar las funciones de una biblioteca es usando:

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

Al hacer esto ya no necesitamos escribir el nombre de la biblioteca cada vez que usamos una función. Ya simplemente escribimos la función, como en

```
>>> b = log10(3.479)
```

Algunas funciones de la biblioteca `math` se dan en la siguiente tabla:

2.11 Operadores

Los operadores indican las operaciones que deseamos realizar sobre las variables y los datos. Existen cuatro tipos de operadores:

1. Aritméticos.
2. Relacionales.
3. Lógicos.
4. De asignación.

2.11.1 Operadores aritméticos

Los operadores aritméticos requieren dos datos para poder efectuarse. La tabla 2.3 proporciona una lista de los operadores aritméticos básicos.

Tabla 2.3 Operadores aritméticos

Operación	Operador	Ejemplo	Precedencia
Suma	+	a + b	3
Resta	-	a - b	3
Multiplicación	*	a*b	2
División	/	a/b	2
División entera	//	a//b	2
Potencia a^x	pow	pow(a, x)	1

Las operaciones indicadas dan un resultado del mismo tipo si es que ambos operadores son del mismo tipo. En el caso de que uno sea entero y otro sea real el resultado es real.

2.11.2 Operadores relacionales

Los operadores relacionales aparecen cuando comparamos. En el Capítulo 3 usamos este tipo de operadores. La tabla 2.4 describe los operadores relacionales.

Tabla 2.4 Operadores relacionales

Operación	Operador	Ejemplo
Mayor que	>	a > b
Menor que	<	a < b
Mayor o igual que	>=	a*b
Menor o igual que	<=	a/b
Igual a	==	a == x
Distinto de	!=	a != x

2.11.3 Operadores lógicos

Los operadores lógicos se usan con variables lógicas (que también reciben el nombre de variables booleanas). En estos operadores ambos datos deben ser datos lógicos. Si a y b son variables lógicas, los operadores lógicos son los que se muestran en la tabla 2.5. En la comparación de relaciones lógicas se usan las relaciones de comparación de la Tabla 2.4.

Ejemplo 2.6

Uso de operadores lógicos

Supongamos que tenemos variables lógicas definidas por

```
a = True
b = False
c = True
d
```

Las siguientes operaciones nos dan los resultados:

```
d = a & b          d = False
d = a | b          d = True
d = b & c          d = False
d = not( a & b )   d = True
d = not( a | b )   d = False
d = a ∧ b          d = True
```

En las operaciones compuestas también tenemos que respetar la precedencia.

> La operación **not** tiene precedencia sobre **and** y
> la operación **and** tiene precedencia sobre **or**.

Por ejemplo, consideremos la siguiente expresión:

```
a & b | c
```

En este caso primero se efectúa la operación AND (por su precedencia sobre el OR) y que resulta en falso, y con el resultado se efectúa la operación OR con c que produce verdadero. Esto se realizaría usando paréntesis como

```
( a & b ) | c
```

que dependiendo de los valores de las variables pueden dar un resultado distinto de

```
a & ( b | c )
```

Desplegar los resultados con variables booleanas produce un 0 si es falso y un **1** si es verdadera.

Para desplegar SIEMPRE es conveniente agrupar entre paréntesis la expresión. Por ejemplo si deseamos desplegar la última expresión es conveniente hacerlo como:

```
print ( ( a & b ) | c )
```

2.11.4 Otras operaciones booleanas

Hay otras ocasiones que el resultado es booleano aun cuando las variables no lo sean. Por ejemplo, consideremos las siguientes variables definidas en Python por:

```
a = 1; b = 2
x = 3; y = 4
CINCO = 5
```

Las siguientes operaciones son verdaderas SI se cumplen o falsas cuando NO se cumplen. En el caso de que se SÍ se cumplan dan el valor de 1 (verdadero) y cuando NO se cumplan dan el valor 0 (falso):

`x < 2`	NO se cumple.
`not(CINCO = a + b)`	NO se cumple.
`(x <= 2) & (a = CINCO − b) | not(y = x − 1)`	NO se cumple.
`((CINCO − b) > −4 ) | not(a/x = 3) & not(b − y < 2 )`	SÍ se cumple.

Tabla 2.5 Operadores lógicos

Operación	Operador	Ejemplo	Precedencia
not	not	not b	1
and	&	a & b	2
	and	a and b	
or exclusivo	∧	a ∧ b	3
or	|	a | b	4
	or	a or b	

2.11.5 Operadores de asignación

Otros operadores de asignación que encuentran uso frecuente en Python se muestran en la tabla 2.6. Estos operadores no requiere mayor explicación con excepción.

Tabla 2.6 Operadores de asignación

Pseudocódigo	Python	Operador
a ← b	a = b	=
a ← a + 1	a = a + 1	a+ = 1
a ← a − 1	a = a − 1	a− = 1
a ← a + b	a = a + b	a += b
a ← a − b	a = a − b	a −= b
a ← a ∗ b	a = a ∗ b	a ∗= b
a ← a / b	a = a / b	a /= b
a ← a % b	a = a % b	a %= b

El operador % obtiene el residuo de la división de dos números enteros y se conoce como la función módulo. **Solamente se puede usar con números enteros.** Por ejemplo, el resultado de 18 % 4 es 2 que es el residuo de 18/4.

2.12 Comentarios

En Python existen dos maneras de escribir líneas de comentarios. La primera es usando el símbolo de numeral # al principio de cada renglón de comentario, como en:

```
# Este es un renglón de comentario.
```

Otra forma de usar los comentarios es después de una instrucción, por ejemplo:

```
a = b**c # Ejemplo de potencia.
```

Lo que aparece despues del símbolo de numeral # no se ejecuta.

La segunda manera es por lo general más usada cuando hay varias líneas de comentarios. El párrafo de comentarios abre con tres apóstrofos "' y se termina con tres apóstrofos "', por ejemplo:

```
'''Este es un conjunto de líneas
   de comentarios que usan la forma
   alterna de indicar comentarios.'''
```

2.13 Entrada y salida de datos en Python

En esta sección presentamos las instrucciones que sirven para realizar en Python las instrucciones **Mostrar** y **Recibir** que nos sirven para desplegar los resultados de nuestros algoritmos y para recibir los datos de entrada del algoritmo.

2.13.1 Despliegue de datos

La realización de la instrucción **Mostrar** es a través de la instrucción `print`. Por ejemplo:

```
print (a)
```

Que despliega el valor de la variable `a` y que debe ir entre paréntesis. Si deseamos imprimir dos variables podemos usar

```
print( a, b )
```

Si deseamos mostrar la cadena alfanumérica "Todos los colores", usamos:

Mostrar: "Todos los colores"; `print ("Todos los colores")`

Para escribir las variables reales con los valores: `a = 270.8` y `b = −130.4`, pero que deseamos que se muestren en dos líneas, usamos el salto de renglón "\n", como en:

```
print ( a, " \n" , b, "\n" )
```

El uso de dos instrucciones `print` produce la impresión de las variables con un salto de renglón equivalente al ejemplo previo:

```
print ( a )
print ( b )
```

O bien

```
print ( a ) , print ( b )
```

También podemos desplegar salida con formato que incluya un texto. Por ejemplo, si tenemos la velocidad `v = 10.478` m/s podemos mostrarla con:

```
print("La velocidad es %3.2f m/s. \n" % v)
```

Aquí se despliega el mensaje alfanumérico y el valor de la variable v.

En la instrucción `print`, dentro de la cadena alfanumérica existe un campo `%3.2f` que indica que al final de la cadena alfanumérica existe una variable de tipo real que se va a imprimir con dos cifras decimales. El número tres no tiene ninguna relevancia.

También se tiene un salto de renglón que se logra con `\n`. Para el ejemplo obtenemos:

```
La velocidad es de 10.478 m/s.
```

La letra `f` se conoce como especificador de formato. La tabla 2.7 da una lista parcial de especificadores de formato.

Tabla 2.7 Especificadores de formato

Especificador	Salida mostrada	Ejemplo
d ó i	Entero con signo	-123
u	Entero sin signo	123
f	Decimal de punto flotante	123.45
e	Notación científica	1.2345 e+002
g	La representación más corta entre f y e	123.45
c	Caracter	'a'
s	Cadena de caracteres	"ejemplo"

El número entre el símbolo % y el especificador indica con cuantos decimales se despliega la variable. Para x = 873.27 tenemos los siguientes resultados:

```
print("El valor de x es%2.5f "% x) → 873.27000
print("El valor de x es%2.5e "% x) → 8.73270e+002
print("El valor de x es%2.5g "% x) → 873.27
```

Para un caracter con nombre `incognita = 'x'`:

```
print("El caracter es%c."% incognita) → El caracter es x.
```

Y para la cadena definida en string `mi_cadena = "libro"`,

```
print("La cadena es%s."% mi_cadena) → La cadena es libro.
```

2.13.2 Entrada de datos

Para los datos de entrada tenemos la instrucción `input`. El formato de input, por ejemplo para leer `x1` es

```
>>> x1 = input("Dame el valor de x1 ")
```

El resultado es después de escribir, por ejemplo, `2.3798`:

```
'2.3798'
```

Es decir, lo que se lee es una cadena. Si lo se quiere leer es un entero o real (flotante) o booleano lo que tenemos que hacer es convertirlo a entero, a real o a booleano de la siguiente manera:

`x1 = int( input("Dame el valor de x1 "))`	para entero.
`x1 = float( input("Dame el valor de x1 "))`	para real.
`x1 = bool( input("Dame el valor de x1 "))`	para booleano.

De la misma manera podemos convertir de tipos. Por ejemplo, para convertir una variable entera a real usamos:

```
>>> a = 1
>>> a = float( a )
>>> a
 a = 1.0
```

que ahora ya es real. Y para convertir de real o entero a cadena usamos:

```
>>> a = 3.2791
>>> a = str( a )
>>> a
```

```
a = '3.2791'
```

Que ya es cadena.

2.13.3 Tabulador

En ocasiones necesitamos escribir varias variables separados por una cierta cantidad de espacios, para esto usamos el tabulador (debe estar entre comillas dobles o sencillas):

"\t" o '\t'

Por ejemplo, si deseamos desplegar las variables a = 2.4, b = −3.21 y c = 47.8, usamos:

```
>>> print( "a \t b \t c")
>>> print( a "\t" b "\t" c)
```

Que nos produce:

```
a       b       c
2.4     -3.21   47.8
```

Como vemos hemos hecho uso del tabulador.

2.14 Algoritmos sencillos

En esta sección cubrimos algunos algoritmos sencillos para ilustrar cómo se escribe un algoritmo.

Para correr el algoritmo en Python debemos escribirlo en otro archivo nuevo, el cual se crea desde el IDLE de Python con:

File → New Window

Como se muestra en la figura 2.3. En esta figura podemos escribir nuestro algoritmo.

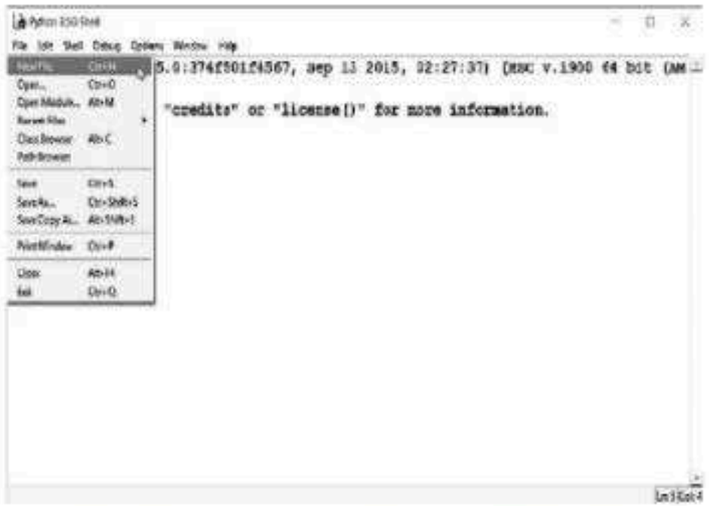


Figura 2.3 Creación de nueva ventana para escribir algoritmo.

Ejemplo 2.7

Intercambio de datos

Se desea intercambiar datos almacenados en dos variables. Si tenemos dos variables `A` y `B` con valores `3.56` y `-7.15`, respectivamente y deseamos que nuestro algoritmo almacene el valor de `3.56` en `B` y que `A` tenga ahora el valor de `-7.15`. Es decir, lo que tenemos es:

$$\begin{aligned} A &\leftarrow 3.56 \\ B &\leftarrow -7.15 \end{aligned}$$

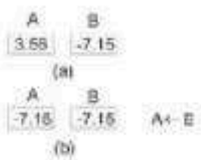


Figura 2.4 Resultado de la operación A ← B, donde se pierde el valor de A.

Lo que se muestra en la figura 2.4a. Si directamente hacemos `A` ← −`7.15` habremos sustituido el valor de `3.56` por el de −`7.15` habiendo perdido el valor que antes tenía la variable `A`, como se muestra en la figura 2.4b. Para evitar esto guardamos temporalmente el valor de `A` en una nueva variable temporal `C` y hacemos la sustitución de `B` en `A` como se ve en la figura 2.5a.

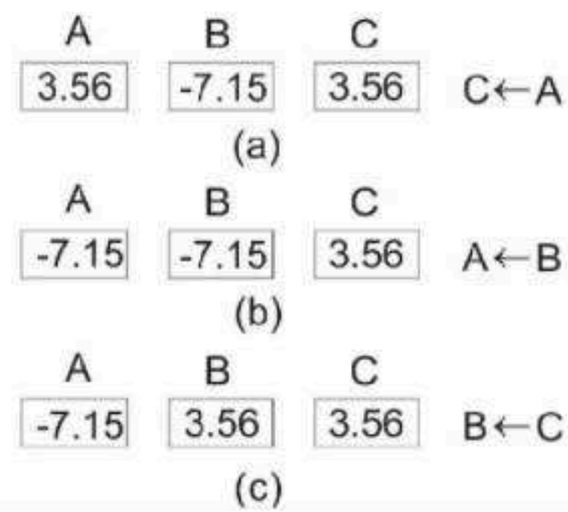


Figura 2.5 Intercambio de datos entre dos variables A y B.

C ← A; # El valor de `C` es `3.56`.

A ← B; # El valor de `A` es −`7.15` como se desea.

Con esto ahora tenemos el valor de `B` almacenado en `A` y el valor de `A` almacenado en `C` (ver figura 2.5b). Finalmente hacemos:

B ← C; #El valor de `B` es `3.56` como se desea.

Lo que se aprecia en la figura 2.5c. Nuestro algoritmo se muestra a continuación. En el vemos que al final se han intercambiado los valores de `A` y `B`. La variable temporal `C` almacena también el valor de `A` pero eso no se usa después. La ventana del archivo final en Python se muestra en la figura 2.6.

Pseudocódigo	Python
Nombre: Intercambio de valores. "'Este algoritmo intercambia los valores de a y b."'	"'Algoritmo de intercambio. Este algoritmo intercambia los valores de a y b. "'
INICIO **Variables:** **real:** A, B, C;	# **INICIO**
A ← 3.56; B ← −7.15;	A = 3.56 B = −7.15
C ← A; #A se almacena en C.	C = A # A =3.56, B = −7.15, C =3.56.

A ← B; #B se almacena en A.
B ← C; # C se almacena en B.
	A = B # A =−7.15, B = −7.15.
	B = C # A =−7.15, B = 3.56.

Mostrar: "Los valores de A y B son:", A, B;	print ("Los valores de A y B son: ") print ("El valor de A es:%5.3f " % A) print ("\n El valor de B es%5.3f" %B)
FIN	# FIN

Figura 2.6 Algoritmo en Python para el intercambio de datos a y b.

Ejemplo 2.8

Conversión de temperaturas

Para convertir temperaturas de grados Fahrenheit (°F) a grados centígrados (°C) usamos, la fórmula

$$°F = 1.8 \times °C + 32$$

Un algoritmo que efectúa la conversión requerida debe realizar los siguientes pasos:

Pseudocódigo	Python
Nombre: Conversión de temperatura.	"'Conversión de temperatura.
"'Este algoritmo convierte de temperatura Celsius a Fahrenheit."'	Este algoritmo convierte de temperatura Celsius a Fahrenheit."'
INICIO	# INICIO
Variables:	
real F, C;	
Recibir: C;	C = `input("Dame la temp. C:\n")`
F← 1.8*C + 32;	F = 1.8*C + 32;
Mostrar:"La temp. F es: ", F;	`print("La temp. F es%0.2f: "%F)`
FIN	# FIN

Ejemplo 2.9

Interés simple y compuesto

Deseamos calcular el interés simple y compuesto que ganará un capital a una tasa de interés dada en un periodo de n meses. Las fórmulas para el interés simple y compuesto son:

$$\text{Interés simple} = \text{capital} \times n \times \text{Tasa de interés}$$
$$\text{Interés compuesto} = \text{capital} \times (1 + \text{Tasa de interés})$$

Para realizar la potencia usamos la función `pow` que requiere que importemos la biblioteca `math`. El algoritmo en pseudocódigo y en Python es:

Pseudocódigo	Python
Nombre: Cálculo de interés.	"' Cálculo de interés.
"'Este algoritmo calcula el interés simple y compuesto."'	Este algoritmo calcula el interés simple y compuesto. "'
INICIO	# INICIO
import math	
Variables:	
real capital, tasa, int_simple;	
real int_compuesto;	
entero n; # número de meses.	
Recibir: capital, tasa, n;	capital=input("Dame el capital:\n")
	tasa=input("Dame la tasa: \n")
	n =input("Dame número de meses:\n")
# Interés simple.	# Interés simple.
int_simple ← capital*n*tasa;	int_simple = capital*n*tasa
# Interés compuesto.	# Interés compuesto.
int_comp ← capital*(1+tasa)∧n;	int_comp = capital*pow(1+tasa, n)
Mostrar:"Int.simple",int_simple;	print("Int.simple:%0.2f"% int_simple)
Mostrar:"Int.comp.",int_comp;	print("Int.comp.:%0.2f"% int_comp)
FIN	# **FIN**

En la realización del algoritmo en Python estamos usando la instrucción para realizar potencias **pow**(a, x) que tiene el siguiente significado:

$$\texttt{pow(a, x)} \rightarrow a^{x}$$

2.15 Variables alfanuméricas en Python

Las variables alfanuméricas en Python se manejan de dos maneras: caracteres y cadenas (en inglés characters y strings). Un caracter es una variable o dato alfanumérico consistente en un solo símbolo, y se encierra entre comillas sencillas o apóstrofe o comillas dobles, por ejemplo: 'a', '1', 's', "c". Una cadena consiste de más de un caracter y también se encierra entre comillas sencillas o dobles como en "libro", 'a123b',

"programa", 'febrero_28'.

En Python, se dice que los caracteres son de tipo chr. Ejemplos de caracteres son como en:

b = "3"; c_2 = 'x'; r1 = "#";

Las cadenas o strings son de tipo str. Ejemplos de cadenas son :

z = "radio" ; x_z = 'variable' ; ab = "12#3";

Un ejemplo simple de leer y escribir un caracter y una cadena nos ilustra su manejo. En capítulos posteriores iremos ampliando nuestro conocimiento de ellos.

```
a = 'q'; # Se define el caracter a y se inicializa.
perro = "Zeus"
print ( a )
print ( perro )
```

Si en nuestra cadena existe un apóstrofe, como en la cadena don't, usamos

'don\'t'

Para especificar la cadena. También podemos usar dobles comillas para no usar \', como en

"don"t"

La longitud de una cadena, es decir, el número de caracteres que forman la cadena, se obtiene con la instrucción len. Por ejemplo,

```
cadena_1 = "Universidad"
len(cadena_1)
11
```

El resultado corresponde al número de caracteres en la cadena.

2.15.1 Operaciones con cadenas

Hay dos operaciones que se pueden realizar con cadenas y caracteres. Estas operaciones son la suma o concatenación y la multiplicación.

2.15.2 Concatenación o suma de cadenas

Para realizar la suma usamos el símbolo + entre las cadenas. Por ejemplo,

```
>>> a = "Victor" ; b = "Hugo"
>>> a + b
' VictorHugo'
```

Vemos que las cadenas se concatenan. Otra forma de hacerlo es simplemente escribiendo las cadenas una después de la otra, como en:

```
>>> 'Victor' 'Hugo'
'VictorHugo'
```

En los dos casos no se generó espacio entre las dos cadenas. Si deseamos formar la cadena 'Victor Hugo' necesitamos incluir el espacio en la suma:

```
>>> 'VictorHugo'  ' '  'Hugo'
'Victor Hugo'
```

O definiendo una cadena de espacio en blanco c = ' ' y sumarla a las otras dos cadenas:

```
>>> c = ' '
>>> a + c + b
'Victor Hugo'
```

2.15.3 Multiplicación de cadenas

La multiplicación de cadenas se realiza entre una cadena y un entero. El resultado es una repetición de la cadena dada por el valor del entero. Por ejemplo:

```
>>> m = "loro"
>>> 3*m
'loroloroloro'
```

Vemos que la cadena 'loro' se repite 3 veces. En el caso de que el factor sea una variable booleana d, obtenemos la cadena si el valor de b es True y una cadena sin caracteres si b es False.

```
>>> b = True; cadena = "perico"
>>> b*cadena
'perico'
```

Se obtiene la misma cadena. Si usamos False o not(b) obtenemos:

```
>>> cadena*False
''

>>> cadena*(not b)
''
```

Se obtiene una cadena vacía en ambos casos.

Pero si multiplicamos por cero o por una cadena vacía, obtenemos también una cadena vacía.

```
>>> cadena*0
''
>>> cadena* ''
''
```

2.16 Listas

En ocasiones necesitamos trabajar con un conjunto de números, letras, o algun otro conjunto de símbolos. Por ejemplo, el conjunto de los números del 1 al 9 es una lista si lo escribimos entre corchetes como:

```
>>> mi_lista = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Otro tipo de lista es un conjunto de nombres de amigos como en :

```
>>> amigos = ['Pedro', 'Rosa', 'Juan', "Luis" ]
```

Con las listas también podemos realizar las mismas operaciones que se realizan con las cadenas. Esto es, podemos concatenar o sumar cadenas y multiplicarlas por un entero. Por ejemplo,

```
>>> a = ["Antonio"]
>>> amigos + a
['Pedro', 'Rosa', 'Juan', 'Luis' 'Antonio']
>>> a*3
['AntonioAntonioAntonio']
```

Para evaluar el número de elementos en una lista usamos la instrucción `len`. Por ejemplo, para las listas `a = [ 1, 2, 3, 4]` y `b = ['ciudad', 'estado', 'pais', 'continente', 'planeta']` tenemos:

```
>>> len(a)
4
>>> len(b)
5
```

En capítulos posteriores haremos uso de listas.

2.16.1 Diccionarios

Los diccionarios son listas formadas por pares. Cada par está formado por una llave (key) o clave y su valor. La llave y su valor se separan por dos puntos. Los pares se separan por comas. Los pares van encerrados entre llaves. Por ejemplo:

```
>>> diccionario1 = { 'a' : 23, 'luna' : 'unica', 'x' : 7.3 }
```

Para imprimir los valores de un par usamos:

```
>>> print( diccionario1['a'])
23
```

que es el valor de 'a'. La longitud de un diccionario se encuentra con la instrucción `len`, como en las listas y las cadenas:

```
>>> len(diccionario)
2
```

Ya que sólo contiene dos pares. Un diccionario también puede tener como elementos listas. Consideremos el siguiente diccionario:

```
x = {" Álgebra":[20, 30, 40] ,"Aritmética":[17, 41], "Cálculo":[9] }
```

Vemos que el diccionario tiene claves que son cadenas y los elementos son listas. Para desplegar un elemento usamos:

```
>>> x ["Álgebra" ]
```

que nos produce el valor:

```
[20, 30, 40]
```

A lo largo del libro usaremos diccionarios y listas donde presentaremos otras instrucciones para manejarlos.

2.17 Instrucciones de Python del Capítulo 2

Instrucción	Descripción
#	Indica línea de comentario.
**	Símbolo de potenciación.
" "	Delimitador de cadena.
' '	Delimitador de cadena.
""" """	Delimitador de comentarios.
&	Instrucción booleana AND.
|	Instrucción booleana OR.
and	Instrucción booleana AND.
bool	Variable de tipo booleano.
False	Valor falso de una variable booleana.
float	Variable de tipo real.
import	Importa una biblioteca.
input	Recibir una variable.
int	Variable de tipo entero.
len	longitud de una cadena, lista, etc.
list	Variable de tipo lista.
math	Biblioteca con funciones de matemáticas.
None	Ningún valor ni tipo.
not	Instrucción booleana NOT.
or	Instrucción booleana OR.
print	Mostrar una variable.
pow	Potenciación.
str	Variable de tipo cadena.
True	Valor verdadero de una variable booleana.
type	Despliega el tipo o clase de la variable.

2.18 Conclusiones

En este capítulo cubrimos los conceptos básicos del diseño de algoritmos, así como su realización en pseudocódigo. Se vio la importancia del pseudocódigo ya que a partir del pseudocódigo se puede traducir a cualquier lenguaje de programación. El lenguaje que usamos en este libro es Python y se presentaron los conceptos básicos de este lenguaje y se dieron algunos ejemplos.

2.19 Ejercicios

2.1 ¿Cuál es la salida del algoritmo que está a continuación? Trate de encontrar el resultado sin correr el código en la computadora.

```
n = 4; k = 2
print ( n )
print ( -n )
print ( n- )
print ( n + k )
print ( k )
print ( n , k )
print ( " " )
print ("n%3.2g" % n)
print (" n * n =%3.2g"% n*n)
print ( n * n )
print ( ' n' )
```

2.2 Indique cuál es la salida en los siguiente segmentos de pseudocódigo:

a) **Enteros:** x;
x ← 3;
x ← x + 2;
Mostrar: x;

b) **Real:** y;
y ← − 5.3;
y ← y*7.0;
Mostrar: y;

c) **Entero**: x ← 8, y ← 3;
x ← mod(x, y);
Mostrar: x;

2.3 Encuentre el resultado para los siguientes segmentos de código en Python.

a)

```
x = 3;
x = x + 2;
print ( x )
```

b)

```
y = - 5.3;
y = y*7.0
print ( y )
```

c)

```
x = 8; y = 3
x = x% y;
print ( x )
```

2.4 La función `getsizeof( )` se usa para calcular el tamaño de una variable. Calcule el tamaño de la variable "a" con `getsizeof( )`.

a) `a = 2.0;`

b) `a = 2;`

c) `a = 20000;`

d) `a = '2';`

e) `a = True;`

2.5 Indique qué resultado produce el siguiente código:

```
x = 42.0
y = 42.3 e-14
a = 42.3 e-14
b = −57.8
m = +87
```

```
n = +249
print ( m*n )
print ( x )
print ( y )
print ( a )
print ( b )
```

2.6 Indique el resultado que produce el siguiente código :

a)
```
x = 90.34;
print ( x )
```

b)
```
z = 2;
y = 7;
x = y + 2*z;
print ("El resultado es:%5.2g" % x)
```

2.7 ¿Cuál es el resultado, Verdadero o Falso, 1 o 0, que se produce en el siguiente código por inspección y luego compruebe en la computadora:

```
x = 8; y = 5
a = 1; b = 2
```

a) `x = 1 & a <= 2 |y < x`

b) `(x = 1 & a <= 2) | y > x`

c) `x = 1 & (a <= 2 | y = x)`

2.8 Escriba el resultado que se obtiene en el siguiente código

```
a = 2; b = 3
x = 8.3; y = -4.6; z = 2.3
print ("a*b =%5.2f" % a*b )
print ("a/b =%5.2f" % a/b )
print ("x/y =%5.2f" % x/y )
```

```
print ("a*z =%5.2f" % a*z )
```

2.9 Escriba el resultado que se produce

```
c = '8';
r = 3;
print ( c*r )
```

2.10 Calcule la respuesta para las siguientes operaciones (sin usar las computadora):

En todos los casos:

```
x = 7; y = 3;
```

a. `x % y;`

b. `x = x + y`

c. `x += y`

d. `x =+ y`

e. `x *= y`

f. `x =* y`

g. `x% = y`

2.11 Escriba un algoritmo que lea los datos: `7.0, 2.0, 8.3, 3.0, 5.6, 4.0` y calcule lo siguiente:

a. El mayor
b. El menor
c. La suma
d. La media

Capítulo 3
Condiciones

3.1 Introducción

3.2 Condiciones

3.3 La condición Si-Si_no

3.4 Condiciones anidadas

3.5 Casos

3.6 Instrucciones de Python del Capítulo 3

3.7 Conclusiones

3.8 Ejercicios

Si yo he visto más allá, es porque logré pararme sobre hombros de gigantes.

Sir Issac Newton

Objetivos

En la mayoría de los algoritmos, el resultado va a depender de qué valor toman las variables. La toma de decisiones en un algoritmo se efectúa por medio de **condiciones**. Python tiene instrucciones para realizar las condiciones.

3.1 Introducción

En la vida real, a menudo es conveniente tomar una decisión acerca de cuál es el siguiente paso a tomar dependiendo de la situación que se presenta. Por ejemplo, a dónde ir de vacaciones. Muchas veces la decisión se toma dependiendo de la cantidad de dinero con la que se cuenta. En otras ocasiones depende del clima apropiado para la actividad que deseamos realizar en las vacaciones. Y en otras ocasiones depende de la disponibilidad del transporte y si hay alojamiento disponible para todos los miembros del grupo que desea ir a vacacionar. Esta toma de decisiones que dependen de una o más condiciones también se presenta en el diseño de un algoritmo. Este tipo de situaciones se conocen como CONDICIONES y es el tema de este capítulo.

En el capítulo 2 se vieron algoritmos muy sencillos que se ejecutaban de manera secuencial. Como se describió en el capítulo 1, ese es el nivel más sencillo de diseñar algoritmos. En este capítulo vemos el siguiente nivel en el diseño de algoritmos: el uso de condiciones que nos permiten tener diferentes alternativas en el orden de ejecución de los pasos del algoritmo.

El capítulo empieza con la definición de la instrucción para una condición, posteriormente vemos como se realiza una condición en Python. A continuación se trata el tema de ciclos anidados. Se cubren las condiciones conocidas como **casos** que en Python se implementan con condiciones anidadas. Finalmente, se muestran ejemplos del uso de las condiciones.

3.2 Condiciones

Las condiciones surgen cuando se trata de efectuar una operación o cálculo dependiendo del **valor de verdad**[1] de una **expresión lógica**. Por ejemplo, en el cálculo de una división podemos decidir que la división no se efectúe cuando el divisor es igual a cero para evitar un error y que la corrida se interrumpa. Otra situación se presenta cuando se quiere calcular la raíz cuadrada de un número negativo. En este caso se checa SI la condición de que el radicando sea positivo se cumpla para poder calcular la raíz cuadrada y enviamos un mensaje cuando el radicando es negativo indicando que no podemos efectuar la operación de calcular la raíz cuadrada.

En pseudocódigo, una CONDICIÓN usa la palabra clave **Si** y tiene el siguiente formato:

Si (expresión lógica):
Entonces:
    instrucciones;
FinSi

La palabra clave **Si** antecede a la expresión lógica. La expresión lógica puede incluir una combinación de expresiones lógicas que pueden contener varios operadores lógicos. Debe escribirse el signo de dos puntos al terminar la expresión lógica.

Si la expresión lógica es VERDADERA, es decir se cumple, entonces se ejecutan las instrucciones y al terminar se continúa con la instrucción que sigue abajo de la instrucción **FinSi**, pero si la expresión lógica es FALSA entonces se continúa con la instrucción que está después del **FinSi**. Nótese que los valores de la expresión lógica pueden ser **Verdadero** o **Falso**. La expresión lógica puede estar encerrada entre paréntesis pero no es obligatorio.

Ejemplo 3.1

Condición simple con Si.

Si realizamos la división de dos números pero queremos checar que el divisor no sea cero podemos usar una condición de la siguiente manera:

INICIO
Entero: a, b;

[1] Verdadero o Falso, True o False para Python.

3.2 Condiciones

Las condiciones surgen cuando se trata de efectuar una operación o cálculo dependiendo del **valor de verdad**[1] de una **expresión lógica**. Por ejemplo, en el cálculo de una división podemos decidir que la división no se efectúe cuando el divisor es igual a cero para evitar un error y que la corrida se interrumpa. Otra situación se presenta cuando se quiere calcular la raíz cuadrada de un número negativo. En este caso se checa SI la condición de que el radicando sea positivo se cumpla para poder calcular la raíz cuadrada y enviamos un mensaje cuando el radicando es negativo indicando que no podemos efectuar la operación de calcular la raíz cuadrada.

En pseudocódigo, una CONDICIÓN usa la palabra clave **Si** y tiene el siguiente formato:

Si (expresión lógica):
Entonces:
    instrucciones;
FinSi

La palabra clave **Si** antecede a la expresión lógica. La expresión lógica puede incluir una combinación de expresiones lógicas que pueden contener varios operadores lógicos. Debe escribirse el signo de dos puntos al terminar la expresión lógica.

Si la expresión lógica es VERDADERA, es decir se cumple, entonces se ejecutan las instrucciones y al terminar se continúa con la instrucción que sigue abajo de la instrucción **FinSi**, pero si la expresión lógica es FALSA entonces se continúa con la instrucción que está después del **FinSi**. Nótese que los valores de la expresión lógica pueden ser **Verdadero** o **Falso**. La expresión lógica puede estar encerrada entre paréntesis pero no es obligatorio.

Ejemplo 3.1

Condición simple con Si.

Si realizamos la división de dos números pero queremos checar que el divisor no sea cero podemos usar una condición de la siguiente manera:

INICIO
Entero: a, b;

[1] Verdadero o Falso, True o False para Python.

Mostrar: "Ingrese a y b: ";
Recibir a, b;
Si b $\neq$ 0:
    Entonces:
    c $\leftarrow$ a/b;
    Mostrar: "Resultado: ", c ;
FinSi
FIN

Notamos que en el caso de que se tenga b $\neq$ 0, la condición es Verdadera porque se cumple la expresión lógica y entonces se realiza la división indicada b/a. Por otro lado si b = 0, entonces NO se ejecuta ninguna acción, ya que la condición es Falsa.

Ejemplo 3.2

Condición con variable lógica

Otro caso se presenta cuando simplemente se tiene una variable lógica en la expresión lógica. Por ejemplo, si se desea realizar una división, podemos checar que el divisor sea distinto de cero, es decir, que b $\neq$ 0 y si se cumple entonces usamos la variable **checar** que sea `Verdadera` y procedemos a ejecutar un conjunto de instrucciones. Entonces tenemos el siguiente algoritmo:

INICIO
Lógica: checar $\leftarrow$ Falsa ; # Inicializa a falsa.
Entero: a, b;
Mostrar: "Ingrese a y b: ";
Recibir: a, b;
Recibir: checar;
Si (checar): # Aquí se ve si la variable checar es verdadera.
    Entonces:
    c $\leftarrow$ a/b;
    Mostrar: "`Resultado`: ", c ;
FinSi
FIN

La expresión lógica se cumple si checar es Verdadera y entonces efectuamos la instrucción que sigue que es c ← a/b. En este ejemplo NO se efectúa ninguna acción si no se cumple la condición checar = Verdadera.

3.2.1 La condición en Python

Las condiciones en Python usan la palabra clave `if`. La condición simple tiene el formato siguiente:

```
if (expresión lógica):
        instrucciones
```

Si la **expresión lógica** se cumple, es decir, es verdadera, entonces se ejecutan las **instrucciones** las cuales **aparecen con sangría** después de la condición y que pueden ser varios renglones de instrucciones.

Las instrucciones de la condición se terminan cuando se elimina la sangría.

Las instrucciones dentro de la condición **deben llevar sangría**. Se debe usar el mismo tamaño de sangría para todas las instrucciones de la condición.

Después de la expresión lógica **SE** escribe el signo de dos puntos.

La expresión lógica termina cuando se escribe el símbolo de dos puntos " : ". Un error muy común es NO escribir los dos puntos después de la condición, como en

if (expresión lógica) ⟵ NO SE ESCRIBIÓ EL SÍMBOLO DE DOS PUNTOS " : ".

lo cual Python reportará como un error.

La expresión lógica puede estar entre paréntesis pero no es necesario hacerlo. Se recomienda hacerlo por claridad de la expresión lógica.

Las expresiones lógicas pueden ser comparar que dos variables sean iguales, una mayor o menor que la otra, o que sean distintas. Todas estas requieren usar los operadores relacionales de la tabla 3.1. Las variables pueden ser reales, flotantes, dobles, alfanuméricas o lógicas.

Observamos de la tabla 3.1 que la comparación de igualdad se escribe con doble signo igual, es decir, usamos **a** == **b** para comparar si dos variables **a** y **b** son iguales. Si son iguales en valor entonces el resultado es Verdadero. En caso contrario cuando son diferentes, el resultado es Falso.

Para comparar igualdad se requiere doble signo igual ==.

Al no escribir doble signo igual en la comparación **a** == **b**, **NO** se realiza la comparación. Lo que se ejecuta es una asignación:

$$\mathbf{a} \leftarrow \mathbf{b}$$

Por lo tanto, debemos observar que:

if(a == b):	**CORRECTO**	Es una compararación.
if(a = b):	**incorrecto**	Es una asignación. Se marca error.
if(a == b)	**incorrecto**	No se escribió **:**

Tabla3.1 Operadores de relación.

Operador	Definición	Ejemplo	Notación Tradicional
>	Mayor que	A > B	A > B
>=	Mayor que o igual a	A >= B	A ≥ B
<	Menor que	C < 4	C < 4
<=	Menor que o igual a	X <= 8	X ≤ 8
==	Igual a	X == Z	X = Z
!=	Distinto a	a != 8.2	a ≠ 8.2

Tabla3.2 Conectivos lógicos.

Operador	Definición	Ejemplo	Precedencia
!	No lógico (NOT)	!X	1
not		not X	1
&	Y lógico (AND)	A & B	2
and		A and B	2
∧	OR exclusivo	A ∧ B	3
|	O lógico (OR)	A | B	4
or		A or B	4

Para construir expresiones lógicas compuestas se usan los conectivos lógicos que se muestran en la tabla 3.2. Como ejemplos de expresiones lógicas con su correspondiente resultado de evaluación tenemos:

```
a = 1; b = 2; c = 3
if( a < b ):              expresión lógica verdadera
if( a < b or c > 0):      expresión lógica verdadera
if( b > 1 and c == 4):    expresión lógica falsa
```

Ejemplo 3.3

Código en Python del Ejemplo 3.1

Este ejemplo checa si el denominador de una división es distinto de cero. De ser así efectúa la división. En el siguiente recuadro se muestra el código Python del pseudocódigo del Ejemplo 3.1. Notemos la similitud sintáctica entre la versión en pseudocódigo y Python.

	Pseudocódigo	Python
1	**Algoritmo:** Condiciones	# Condiciones.
2	**Variables:**	
3	**Entero:** a, b;	
4	**INICIO**	# INICIO
5	**Mostrar:**'Ingrese a y b:'	a = float(input("Ingrese a"))
6		b = float(input("Ingrese b"))
7	**Recibir** a, b;	
8		
9	# Empieza la condición.	# Empieza la condición.
10	**Si** b $\neq$ 0 :	if (b != 0):
11	c $\leftarrow$ a/b	c = a/b
12	**Mostrar:**'Resultado:', c	print("Resultado:", c)
13	**FinSi**	# Termina la condición.
14		
15	**FIN**	# FIN

Vale la pena hacer notar que lo que se requiere en una expresión lógica es que se cumpla o que sea **verdadera**. Por ejemplo, si tenemos lo siguiente:

```
a = 1
b = int(input("Dame el valor de b: "))
if (b):
    a = 2
```

Al analizar este algoritmo, antes de la condición tenemos `a = 1`. Después se recibe el valor de b. Si b es verdadera, se cumple la expresión lógica y se ejecuta `a = 2`. Pero si b es falsa, no se ejecuta nada y NO se modifica el valor de a.

De los algoritmos vistos hasta este punto, el lector puede notar las similaridades y las diferencias entre el pseudocódigo y el código en Python.

Ejemplo 3.4

Código en Python del Ejemplo 3.2

Para el Ejemplo 3.2 tenemos que el pseudocódigo y el código en Python se genera de la siguiente manera:

	Pseudocódigo	Python
1	**Algoritmo:** Cond. lógica.	# Condición lógica.
2	**INICIO**	# INICIO
3	**Variables:**	
4	**Entero:** a, b	
5	**Lógica:** checar	checar = False
6	**Mostrar:** "Ingrese a y b:"	a = float(input ("Ingrese a:"))
7		b = float(input ("Ingrese b:"))
8	**Recibir** a, b	
9	**Si** (b != 0):	if (b != 0):
10	checar ← Verdadera	checar = True
11	**FinSi**	# Fin de la condición.
12	# Empieza la condición.	# Empieza la condición.
13	**Si** (checar):	if (checar):
14	c ← a/b	c = a/b
15	**Mostrar:**"Resultado:", c	print("Resultado:", c)
16	**FinSi**	# Fin de la condición.
17	**FIN**	# FIN

Ejemplo 3.5

Solución de una ecuación de primer grado

Una ecuación de primer grado es de la forma $ax + b = 0$. La solución de esta ecuación es $x = -b/a$, lo cual requiere que a sea distinta de cero. Esto quiere decir que debemos realizar una comparación por medio de un **Si**. Esta condición la usamos para checar si $a \neq 0$. Las acciones a realizar son las siguientes:

- Recibir a y b

- Checar si a $\neq$ 0
- Resolver $x \leftarrow b/a$
- Mostrar x

En pseudocódigo y en Python tenemos entonces:

	Pseudocódigo	Python
1	**Algoritmo:** Ec.primer grado	# Ec. de primer grado.
2	**INICIO:**	# INICIO:
3	**Variables:**	
4	**Real:** a, b, x	
5	**Mostrar:** "Dame a :"	a = float(input("Dame a:"))
6	**Mostrar:** "Dame b :"	b = float(input("Dame b:"))
7	**Recibir:** a, b	
8	**Si** a != 0	if (a != 0):
9	x ← − b/a	x = − b/a
10	**Mostrar:**"Resultado:", x	print ("Resultado:", x)
11	**FinSi**	# Fin de la condición.
12	**FIN**	# FIN

Al ejecutar este algoritmo para $2x + 3 = 0$ tenemos $a = 2$, $b = 3$ y el valor de x es 0.6666. Para la ecuación $0x + 3 = 0$ se tiene $a = 0$ y $b = 3$ y el algoritmo no da ninguna respuesta.

3.2.2 Ejercicios de condiciones simples

Ejemplo 3.6

Evaluación de expresiones lógicas

Para la evaluación de expresiones lógicas consideremos las variables:

```
a = True, b = True, c = 23.2, d = -18.34, e = -98
```

Las variables a y b son booleanas o lógicas, c y d son reales y e es entera. Las siguientes expresiones lógicas nos producen los resultados:

1	`>>> a and b` `True`	Porque a y b son True.
2	`>>> a and not b` `False`	Primero se ejecuta el not.
3	`>>> c > d` `True`	Porque c es mayor que d.
4	`>>> c < e & e < d` `False`	Porque c es mayor que e.
5	`>>> c < e or a == b` `True`	Porque a es igual a b aunque c <e es falso.
6	`>>>not(c > e)` `False`	Porque c e es verdadero y luego se efectua el not .
7	`>>> (a)` `True`	Porque a es verdadera.

Ejemplo 3.7

Fórmulas químicas

Se desea realizar un algoritmo que reciba fórmulas químicas de compuestos y despliegue su nombre en español. Los compuestos químicos son sal común o cloruro de sodio NaCl, agua H_2O y ácido clorhídrico HCl.

El algoritmo debe tener los siguientes pasos:

1. Leer la fórmula del compuesto.
2. Usando una condición seleccionar el nombre del compuesto químico.
3. Desplegar el nombre del compuesto químico.

El algoritmo en pseuocódigo y en Python es el siguiente:

	Pseudocódigo	Python
1	**Algoritmo:** Comp. químicos	# Compuestos químicos
2	**INICIO:**	# INICIO:
3	**Variables:**	
4	**Cadena:** a	
5	**Mostrar:** "Dame la fórmula:"	a = input("Dame la fórmula:")
6	**Recibir:** a	
7	**Si** a == 'NCl'	if (a == 'NCl'):
8	**Mostrar:**"Sal común"	print ("Sal común")
9	**FinSi**	# Fin de la condición.
10	**Si** a == 'H2O':	if (a == 'H2O'):
11	**Mostrar:** " Agua"	print ("Agua")
12	**FinSi**	# Fin de la condición.
13	**Si** a == 'HCl'	if (a == 'HCl'):
14	**Mostrar:** "Á. clorhídrico"	print ("Á. clorhídrico")
15	**FinSi**	# Fin de la condición.
16	**FIN**	# FIN

Ejemplo 3.8

Otras soluciones para la ecuación de segundo grado

La ecuación de segundo grado tiene otro tipo de soluciones cuando el discriminante es menor que cero. En ese caso las raíces son complejas conjugadas y se pueden obtener con:

$$x_{1,2} = \frac{-b \pm i\sqrt{4ac - b^2}}{2a}$$

Con estas ecuaciones podemos mejorar el algoritmo del Ejemplo 2.5. De la siguiente manera:

1. Leer los coeficientes a, b, c.

2. Calcular el discriminante $d = b^2 - 4ac$.

3. Checar **si** $d > 0$

   - Las soluciones son reales y distintas dadas por:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

4. Checar **si** $d == 0$

   - Las soluciones son reales e iguales dadas por:

$$x_1 = x_2 = \frac{-b}{2a}$$

5. Checar **si** $d < 0$

   - Las soluciones son complejas conjugadas dadas por:

$$x_{1,2} = \frac{-b \pm i\sqrt{4ac - b^2}}{2a}$$

donde i es el número imaginario dado por $i = \sqrt{-1}$ y por lo tanto en este caso las raíces son complejas conjugadas. El algoritmo en pseudocódigo y Python se presenta a continuación. La división `b/(2*a)` la hemos implementado como `b/2/a`.

	Pseudocódigo	Python
1	**Algoritmo:**Ec. Segundo grado	# Ec. Segundo grado
2	**INICIO:**	# INICIO:
		from math import *
4	**Variables:**	
5	**Real:** a, b, c, d, x1, x2	
6	**Mostrar:** "Dame a:"	a = float(input("Dame a:"))
7	**Recibir:** a	
8	**Mostrar:** "Dame b:"	b = float(input("Dame b:"))
9	**Recibir:** b	
10	**Mostrar:** "Dame c:"	c = float(input("Dame c:"))
11	**Recibir:** c	
12	# Calcular discriminante	# Calcular discriminante
13	d = b∧ 2 - 4*a*c	d = b∧2 - 4*a*c
14	**Si** d > 0 :	if (d > 0):
15	**Mostrar:**"Raíces reales"	print("Raíces reales")
16	**Mostrar:**" y diferentes:"	print("y diferentes:")
17	x1 =(-b + sqrt(d))/2/a	x1 =(-b + sqrt(d))/2/a
18	x2 =(-b - sqrt(d))/2/a	x2 =(-b - sqrt(d))/2/a
19	print(x1)	print(x1)
20	print(x2)	print(x2)
21	**FinSi**	# Fin de la condición.
22	**Si** d == 0 :	if(d == 0):
23	**Mostrar:**"Raíces reales"	print("Raíces reales")
24	**Mostrar:**" e iguales:"	print("e iguales:")
25	x1 = - b /2/a	x1 = - b /2/a
26	x2 = x1	x2 = x1
27	print(x1)	print(x1)
28	print(x2)	print(x2)
29	**FinSi**	# Fin de la condición.
30	**Si** d < 0 :	if (d < 0):
31	**Mostrar:**"Raíces complejas"	print('Raíces complejas')
32	**Mostrar:**" y conjugadas:"	print("y conjugadas:")
33	x1_parte_real = -b/2/a	x1_parte_real = -b/2/a
34	x1_parte_im =sqrt(-d)/2/a	x1_parte_im =sqrt(-d)/2/a
35	x2_parte_real = -b/2/a	x2_parte_real = -b/2/a
36	x2_parte_im=-sqrt(-d)/2/a	x1_parte_im=-sqrt(-d)/2/a
37	print(x1_parte_real)	print(x1_parte_real)
38	print(x1_parte_im)	print(x1_parte_im)
39	print(x2_parte_real)	print(x2_parte_real)
40	print(x2_parte_im)	print(x2_parte_im)
41	**FinSi**	# Fin de la condición.
42	**FIN**	# **FIN**

Notamos que al terminar cada condición empieza otra para tener en total tres condiciones.

3.3 La condición Si - Si_no

En el ejemplo de la división podría ser deseable realizar alguna otra acción cuando no se pueda realizar la división, por ejemplo, desplegar un mensaje de que la división no se puede realizar. Esto lo podemos lograr usando la forma de la condición **Si-Si_no** cuyo formato es:

```
Si (expresión lógica):
     instrucciones 1;
Si_no:
     instrucciones 2;
FinSi
```

Si se cumple la condición se procede a ejecutar las `instrucciones 1` y si no se cumple se ejecutan las `instrucciones 2` y al terminarse cualquiera de los dos conjuntos de instrucciones, entonces se continúa con la instrucción que sigue al **FinSi**.

3.3.1 Condición `if - else`

En Python, la condición **Si-Si_no** se realiza con las palabras clave `if - else` y su formato es el siguiente:

```
if (expresión lógica):
     instrucciones 1;
else:
     instrucciones 2;
```

Vemos que la expresión lógica se escribe dentro de paréntesis y los conjuntos de instrucciones se encuentran después de los dos puntos con una sangría. Después de la expresión lógica SE escriben dos puntos.

También se escriben dos puntos después del `else`.

Ejemplo 3.9

División de dos números con Si-Si_no

El ejemplo 3.1 se puede mejorar si le agregamos un mensaje en caso de que no se pueda realizar la división porque el denominador sea igual a cero. La condición **Si-Si_no** es ahora:

	Pseudocódigo	Python
1	**Algoritmo:** Uso de **Si-Si_no**.	# Uso de **if-else**.
2	**INICIO**	# **INICIO**
3	**Variables:**	
4	**Real: a, b, c**	
5	**Mostrar:** "Ingrese a y b:"	`a = float(input("Ingrese a: "))` `b = float(input("Ingrese b: "))`
6	**Recibir:** a, b	
7	**Si** b $\neq$ 0:	`if(b != 0):`
8	c $\leftarrow$ a/b	`c = a/b`
9	**Mostrar**: "Resultado: ", c	`print( "Resultado:", c)`
10	**Si_no**	`else:`
11	**Mostrar:** "La división no"	`print("La division no")`
12	**Mostrar:** "se puede"	`print( "no se puede")`
13	**Mostrar:** "realizar."	`print( "realizar. ")`
14	**FinSi**	`# Fin del if-else.`
15	**FIN**	`# FIN`

Vemos que el algoritmo es más completo ya que da más información al usuario.

3.3.2 Ejemplos adicionales de condiciones if-else

Ejemplo 3.10

Intereses de una cuenta bancaria

Un banco paga intereses a los depósitos dependiendo del saldo. Si el saldo es menor a $10,000.00 se paga un interés anual de 3 %, pero si es mayor a $10,000.00 se paga un interés anual de 4 %. Escriba un algoritmo que reciba el saldo, calcule el interés correspondiente y muestre el saldo de la cuenta al final del primer año.

Para realizar nuestro algoritmo usamos una condición **Si-Si_no** para poder determinar el interés que se va a pagar al cuentahabiente y Mostrar su saldo final.

- Primero se da el nombre del algoritmo.
- El segundo paso es iniciar el algoritmo.
- Recibir el saldo.
- Empezar la condición:
  - Si el saldo es menor a $10,000.00 calcular un interés de 3 %.

    Saldo ← Saldo * (1 + 0.03);
  - Si el saldo es mayor a $10,000.00 calcular un interés de 4 %.

    Saldo ← Saldo * (1 + 0.04);
- Mostrar el saldo final.

Como sabemos, el interés que se paga se suma al saldo anterior multiplicando el saldo actual por (1 + interés) para obtener el saldo nuevo.

El algoritmo sería de la siguiente manera:

	Pseudocódigo	Python
1	**Algoritmo:** Cálculo del saldo.	# Cálculo del saldo
2	**INICIO**	#**INICIO**
3	**Variables:**	
4	**Real:** Saldo;	
5	**Mostrar:**"Dame saldo actual:"	`print('Dame saldo actual:')`
6	**Mostrar:**"Dame saldo actual:"	`Saldo = float(input( ))`
7	**Recibir:** Saldo;	
7	# Empieza la condición.	# Empieza la condición.
8	**Si** Saldo < 10000.00	`if (Saldo < 10000.00):`
9	Saldo ←Saldo*(1 + 0.03)	`Saldo = Saldo*(1 + 0.03)`
10	**Si_no**	`else:`
11	Saldo ←Saldo*(1 + 0.04)	`Saldo = Saldo*(1 + 0.04)`
12	**Fin_Si**	# Fin del if.
13	**Mostrar:** "Saldo final:", Saldo	`print("Saldo final es %5.2f " %Saldo)`
14	**FIN**	# **FIN**

Ejemplo 3.11

Solución de una ecuación de primer grado con if-else

El ejemplo 3.2 resolvía una ecuación de primer grado de la forma $ax + b = 0$. La solución de esta ecuación es $x = -b/a$. Para mejorar el algoritmo del ejemplo 3.2 también checamos que $b \neq 0$, para evitar que si $a = 0$ tengamos una división entre cero o tengamos $0/0$ que es una indeterminación. El algoritmo entonces requiere dos condiciones: que a y b sean al mismo tiempo distintos de cero. Esto quiere decir que debemos calcular dos expresiones lógicas. La primera para checar si $a \neq 0$ y la segunda para checar si $b = 0$. Las acciones a realizar son las siguientes:

- Recibir `a` y `b`
- Checar si `a` $\neq$ 0
- Checar si `b` es igual a cero
- Resolver `x` $\leftarrow$ `b/a`
- Mostrar `x`

En pseudocódigo y en Python tenemos entonces:

	Pseudocódigo	Python
1	**Algoritmo:** Ec. primer grado.	# Ec. de primer grado.
2	**INICIO:**	#**INICIO:**
3	**Variables:**	
4	**entera:** a, b, x	
5	**Mostrar:** "Dame a y b"	`a = float(input("Dame a:"))`
6	**Recibir:** a, b	`b = float(input("Dame b:"))`
7	**Si** a != 0	`if (a != 0):`
8	x ← − b/a	`x = − b/a`
9	**Mostrar:** x	`print(x)`
10	**Si_no**	`else:`
11	**Si** b == 0;	`if (b == 0):`
12	**Mostrar:**"a = 0 y b=0"	`print("a =0 y b =0")`
13	**Si_no**	`else:`
14	**Mostrar:**'No hay solución'	`print('No hay solución')`
15	**FinSi**	#Fin del if anidado.
16	**FinSi**	#Fin del if
17	**FIN**	#**FIN**

Al ejecutar este algoritmo para $2x + 3 = 0$ tenemos $a = 2$, $b = 3$ y el valor de x es 0.6666. Para $2x = 0$ tenemos $b = 0$ y obtenemos $x = 0$. Para $0x + 3 = 0$ se tiene $a = 0$ y $b = 3$ y el algoritmo regresa el mensaje "No hay solución". Y si le damos $a = 0$ y $b = 0$ obtenemos el mensaje: "No hay solución".

3.4 Condiciones anidadas

Una condición es anidada si está dentro de otra condición. El formato de una condición anidada en pseudocódigo es:

Si (expresión lógica A):
    instrucciones 1;
Si_no:

Este es el **Si** anidado

Si (expresión lógica B):
    instrucciones 2;
Si_no:
    instrucciones 3;
FinSi

FinSi

3.4.1 if anidado en Python

El **if** anidado en Python se forma por lo general usando el **if-else**. El formato es:

```
if (expresión lógica A):
   instrucciones 1;
else:
```

Este es el **if** anidado

```
   if (expresión lógica B):
      instrucciones 2;
   else:
      instrucciones 3;
# Instrucción siguiente.
```

Si la `expresión lógica A` es verdadera, entonces, se ejecutan todas las `instrucciones 1` y se prosigue con la `Instrucción siguiente`. Si la `expresión lógica A` no se cumple, entonces, se checa la `expresión lógica B` y si esta es verdadera se ejecutan las `instrucciones 2`. Al terminar se procede a ejecutar la `Instrucción siguiente`. Si la `expresión lógica B` NO se cumple, es decir, es **falsa**, entonces se ejecutan las `instrucciones 3` y al terminar se pasa a la `Instrucción siguiente`.

Una manera más común de escribir el `if` anidado es por medio de la contracción de `else-if` que es `elif`. Después de un `elif` debemos añadir una condición. El formato es el siguiente:

```
if (expresión lógica A) :
   instrucciones 1;
elif expresión lógica B:
   instrucciones 2;
else:
   instrucciones 3;
# Instrucción siguiente;
```

Un ejemplo ilustra el uso de los `if` anidadados.

Ejemplo 3.12

Depósitos con interés con Condición Anidada

Como ejemplo consideremos el Ejemplo 3.10 pero ahora supongamos que si el saldo es mayor a un millón de pesos se le paga un interés de 6 %. El algoritmo, en la parte de la condición, es ahora de la siguiente manera:

Si Saldo <10,000.00
  Saldo ← Saldo*(1 + 0.03);
Si_no
    Si Saldo < 1,000,000.00 # Aquí se sabe que el Saldo < 1 millón.
      Saldo ← Saldo*(1 + 0.04);
    Si_no # Aquí se sabe que el Saldo > 1 millón.
      Saldo ← Saldo*(1 + 0.06)
    FinSi
FinSi

En Python es:

```
if (Saldo < 10,000.00):
   Saldo = Saldo*(1 + 0.03)
elif (Saldo < 100,000.00):    # Aquí empieza el if anidado.
   Saldo = Saldo*(1 + 0.04)
else:
   Saldo = Saldo*(1 + 0.06)
    # Fin del if anidado
# Fin del if principal
```

Vemos que el **if** anidado tiene su expresión lógica que si se cumple se efectúa

$$\text{Saldo} = \text{Saldo}*(1 + 0.04)$$

ya que la cantidad es mayor de $10,000.00 pero menor de $100,000.00. Pero si no se cumple entonces vamos al **else** que indica que la cantidad es mayor a un millón y el interés que se paga es de 6 %.

Para un Saldo inicial de $12000.00, el saldo final es de $12,480.00 y para un saldo inicial de 2 millones el saldo final es de $2,120,000.00.

Un caso sencillo de anidamiento es cuando la condición anidada es simple.

Si expresión lógica A :	`if (expresión lógica A):`
instrucciones 1	`instrucciones 1`
Si_no	`elif expresión lógica B:`
Si (expresión lógica B):	
instrucciones 2;	`instrucciones 2;`
FinSi # Fin del **Si** anidado	# Fin del **Si** anidado
FinSi	# Fin del **Si** principal

3.4.2 Ejemplos adicionales con condiciones anidadas

Ejemplo 3.13

Calificaciones numéricas y literales

Consideremos la asignación de una calificacón literal a un rango dado de calificaciones numéricas de la siguiente manera:

si la calificación está en el rango de:

- 9.1 a 10 asignar la calificación A Excelente.
- 8.1 a 9.0 asignar la calificación B Muy bien.
- 7.5 a 8.0 asignar la calificación C Bien.
- Menos de 7.5 asignar la calificación R Reprobado.

Para checar qué calificación literal se otorga, el algoritmo debe checar primero si la calificación numérica es mayor de 9.0. Si esto se cumple la calificación numérica correspondiente es A.

Si la condición anterior no se cumple, entonces checamos si la calificación es mayor de 8.0. En esta parte ya no se puede tener una calificación mayor a 9.0 porque esto ya se descartó en la primera condición. Si ahora se cumple la condición se asigna la letra B.

La siguiente condición se ejecuta cuando la calificación es mayor o igual a 7.5 en cuyo caso se asigna C.

En caso de que no se cumpla ninguna de las condiciones anteriores se asigna R.

Usando una condición anidada podemos codificarlo en pseudocódigo y en Python como:

	Pseudocódigo	Python
1	**Si** calificación > 9.0	`if(calificacion > 9.0):`
2	**Mostrar:**"La calificación es A"	`print("La calificación es A.")`
3	**Si_no**	`elif (calificacion > 8.0):`
4	**Si** calificación > 8.0	
5	**Mostrar:**"La calificación es B."	`print ("La calificación es B.")`
6	**FinSi**	
7	**Si_no**	`elif (calificacion >= 7.5):`
8	**Si** calificación >= 7.5	
9	**Mostrar:**"La calificación es C."	`print("La calificación es C.")`
10	**FinSi**	
11	**Si_no**	`else:`
12	**Mostrar:**"La calificación es R."	`print("La calificación es R.")`
13	**FinSi**	
14	**FIN**	# FIN

Vemos como se usan dos condiciones anidadas.

Ejemplo 3.14

Condición con variables lógicas

En este ejemplo usamos una variable lógica para la expresión lógica del **Si**. Supongamos que se necesita realizar un `Si` donde la expresión lógica es verdadera o falsa.

	Pseudocódigo	Python
1	**Lógica:** a	
2	**Entero:** B, c	
	⋮	⋮
3	a← verdadero;	`a = True`
4	**Si** a	`if (a):`
5	B ← 3*c	`B = 3*c`
6	**Si_no**	`else:`
7	B ← c + 1	`B = c + 1`
8	**FinSi**	# Fin del if.

Vemos en este caso que la expresión lógica es la variable lógica a y la expresión lógica se cumple solamente cuando a es verdadera, en cuyo caso se ejecuta `B = 3*c`. Si a es

falsa entonces se ejecuta `B = c + 1`.

Ejemplo 3.15

Condición múltiple

En ocasiones la condición consiste en dos o más expresión lógicas que se deben satisfacer. El ejemplo anterior lo podemos mejorar añadiendo el bono por asistencia perfecta en una variable lógica, que otorga una mención honorífica por asistencia perfecta.

Si (calificación > 9.0 **and** asistencia == 'T')
    Mostrar:"La calificación es A Excelente y Mención por Asistencia"
Si_no
    Si (calificación > 9.0 **and** asistencia ≠ 'T')
        Mostrar:"La calificación es A Excelente"
    Fin_Si
Si_no
    Si (calificación > 8.0 **and** asistencia == 'T')
        Mostrar:"La calificación es B Muy bien con Mención"
    Si_no
        Si (calificación > 8.0 **and** asistencia ≠ 'T')
            Mostrar:"La calificación es B Muy bien"
        Fin_Si
    Fin_Si
Fin_Si

Codificado en Python, el algoritmo completo es:

```
calificacion = float(input ( "Dame una calificación: \n"))
print('Dame la asistencia: 1 si asistió o 0 si no asistió.')
asistencia = int ( input( ) )

if (calificacion > 9.0 and asistencia == 1):
   print("La calificación es A Excelente con Mención Honorífica.")
elif (calificacion > 9.0 and asistencia != 1):
   print("La calificación es A Excelente.")
```

```
elif (calificacion > 8.0 and asistencia == 1):
    print("La calificación es B Muy bien con Mención.")
elif (calificacion > 8.0 and asistencia != 1):
    print("La calificación es B Muy bien.")
elif (calificacion = 7.5):
    print("La calificación es C Bien.")
else:
    print("La calificación es R Reprobado. Lo siento mucho.")
# Fin del if.
```

En este ejemplo hemos usado 1 cuando la variable lógica es verdadera o `True` y 0 cuando es falsa o `False`.

3.5 Casos

Hay ocasiones en las que el uso de las condiciones anidadas hace muy difícil diseñar una aplicación. En esos casos contamos con una instrucción llamada **casos** que nos permite ejecutar distintas instrucciones dependiendo del valor de una condición o de un conjunto de condiciones.

Los casos se aplican cuando las opciones a seguir, dependen de valores numéricos enteros o de variables alfanuméricas.

Cuando una decisión se toma según diferentes valores de una expresión, por simplicidad puede expresarse como:

Según expresión

{

Caso valor 1: {instrucciones en caso de que la expresión tenga valor 1}

Caso valor 2: {instrucciones en caso de que la expresión tenga valor 2}

⋮

Caso valor n: {instrucciones en caso de que la expresión tenga valor n}

default: {instrucciones en caso de que la expresión no produzca ninguno de los valores anteriores}

}

Notemos que valor 1 a valor n deben ser constantes (no pueden ser expresiones).

La descripción anterior es válida siempre que la expresión sea un entero o un caracter alfanumérico. La manera en que funciona esta condición es la siguiente:

- Primero se checa el valor de la expresión. Dependiendo del valor de la expresión la siguiente instrucción es la que corresponde al caso de ese valor.
- Después de ejecutar dichas instrucciones se sale de la instrucción y la siguiente condición es la que sigue a la llave que cierra la condición del Según-Caso.

Un ejemplo nos muestra la manera en que trabaja la instrucción Segun-Caso.

Ejemplo 3.16

Número del mes.

Se desea realizar una aplicación que muestre el nombre del mes dado un número del 1 al 12. En este caso, realizar el algoritmo usando solamente Si anidados lo hace muy complicado de seguir. En cambio usando Según-Caso lo hace muy sencillo.

Si la variable es `número_mes` y es un entero entonces vemos que podemos tener:

Según `número_mes`

Caso 1: Mostrar: "El mes es enero.";

Caso 2: Mostrar: "El mes es febrero.";

Caso 3: Mostrar: "El mes es marzo.";

Caso 4: Mostrar: "El mes es abril.";

Caso 5: Mostrar: "El mes es mayo.";

Caso 6: Mostrar: "El mes es junio.";

Caso 7: Mostrar: "El mes es julio.";

Caso 8: Mostrar: "El mes es agosto.";

Caso 9: Mostrar: "El mes es septiembre.";

Caso 10: Mostrar: "El mes es octubre.";

Caso 11: Mostrar: "El mes es noviembre.";

default: Mostrar: "El mes es diciembre.";

Como se observa, el uso de una condición Según-Caso hace más claro el proceso de decisiones.

Ejemplo 3.17

Números de un dado.

Un dado tiene seis caras. Dado un número del 1 al 6, se desea mostrar el número que aparece en la cara opuesta del dado, recordando que:

El lado opuesto del lado	es
1	"seis"
2	"cinco"
3	"cuatro"
4	"tres"
5	"dos"
6	"uno"

En este algoritmo se pueden usar condiciones anidadas pero es más fácil usar Según-Caso. El algoritmo queda como:

Según dado

Caso 1: Mostrar: "El lado opuesto es seis.";

Caso 2: Mostrar: "El lado opuesto es cinco.";

Caso 3: Mostrar: "El lado opuesto es cuatro.";

Caso 4: Mostrar: "El lado opuesto es tres.";

Caso 5: Mostrar: "El lado opuesto es dos.";

Caso 6: Mostrar: "El lado opuesto es uno.";

Default: Mostrar: "No se dio un número del 1 al 6.";

Notamos que el default incluye aquellos casos donde se da un número que no está en el dado.

En Python no se tiene una instrucción que implemente los casos. En la mayoría de los lenguajes de programación si existe bajo el nombre de `Switch-Case`. Es por esto que la instrucción `switch-case` tiene que implementarse con instrucciones `if`. Un ejemplo muestra la manera de hacerlo.

Ejemplo 3.18

Calculadora básica con menú

Una calculadora básica se puede realizar con condiciones. Se desea realizar alguna de las operaciones básicas con dos números x, y. Además se desea calcular la potencia y el logaritmo. Se deben de considerar los casos donde y $\leftarrow$ 0 donde la división x/y NO se puede realizar y cuando x <= 0 donde NO se puede calcular el log(x). Se desea generar un menú para que el usuario pueda seleccionar la operación a realizar. Una manera de hacerlo es la siguiente:

1. Se reciben los dos números `x`, `y` para realizar la operación.
2. Se recibe la operación a realizar mediante la variable `opcion` la que selecciona en el menú qué operación ejecuta el algoritmo
3. Se inicializa la variable lógica `bandera` $\leftarrow$ `False`. Si la división o el logaritmo no se pueden calcular, se hace `bandera` $\leftarrow$ `True`.
4. Mediante condiciones se realiza la operación deseada.
   - En el caso de la división, si `y` $\leftarrow$ 0, NO se puede realizar la división, se muestra un mensaje y se hace `bandera` $\leftarrow$ True.
   - En el caso del logaritmo, si `x` $\leq$ 0, NO se puede calcular el logaritmo, se muestra un mensaje y se hace `bandera` $\leftarrow$ True.
5. Se muestra el resultado en el caso de que `bandera` $\leftarrow$ False.

En código Python tenemos:

```
# Algoritmo: Calculadora.
from math import log

bandera = False
x = float( input("Dame el valor del número x: "))
y = float( input("Dame el valor del número y: "))
```

```
print("Dame la opción que deseas realizar:\n")

# Se despliega el menú para seleccionar la opción que deseas realizar:
print( " 1) Sumar (El primero más el segundo)" )
print( " 2) Restar (El primero menos el segundo)" )
print( " 3. Multiplicar (El primero por el segundo) ")
print( " 4. Dividir (El primero sobre el segundo)" )
print( " 5. Potencia (El primero a la potencia del segundo)")
print( " 6. Logaritmo (El logaritmo del primero.)")
opcion = int(input( "Dame la opción: "))

# Empieza la ejecución del menú con condiciones anidadas.
if (opcion == 1) :
   z = x + y
   print ( x , "+" , y)
elif (opcion == 2):
    z = x - y
    print ( x, "-" , y)
elif (opcion == 3):
   z = x * y
   print  ( x, " × " , y)
elif( opcion == 4 and y != 0 ):
   z = x / y
   print( x, "/" , y)
elif ( opcion == 4 and y == 0 ):
   print("El denominador es igual a cero y")
   print("NO se puede realizar la división.")
   bandera = True
elif ( opcion == 5 ):
    z = pow(x , y)
   print( x, "^" , y )
elif ( opcion == 6 and x > 0 ):
   z = log (x)
    print ( "logaritmo de" x )
elif ( opcion == 6 and x <= 0 ):
   print("El valor de x es <= a cero y" )
```

```
        print ("NO se puede calcular el logaritmo.")
        bandera = True # Con bandera = 1 indica que no se puede
                       #calcular lo deseado."
    else:
        print("Opcion desconocida.")

    # Se escribe el resultado con otra condición:
    if ( opcion < 7   and   bandera == False ):
        print (" Resultado = " z )
    # FIN
```

La figura 3.1 muestra la ventana del menú con la operación de resta.

Figura 3.1 Calculadora usando menú.

En los capítulos siguientes haremos uso de las condiciones en combinación con otras instrucciones como son los ciclos y dentro de funciones.

3.6 Instrucciones de Python del Capítulo 3

Instrucción	Descripción
`if`	Palabra para empezar una condición.
`else`	Palabra para ejecutar otra condición si la primera condición no se cumple.
`elif`	Palabra para checar una condición adicional si la condición anterior no se cumple.

3.7 Conclusiones

En este capítulo vimos las instrucciones para implementar las condiciones simples y anidadas. Con ejemplos mostramos su uso tanto en pseudocódigo como en el lenguaje Python. Los ejemplos presentados en este capítulo son de una complejidad mayor y nos permiten realizar otras tareas que involucran decisiones basadas en condiciones. Los capítulos siguientes presentan otras instrucciones que nos permiten realizar algoritmos de una complejidad mayor.

3.8 Ejercicios

3.1 ¿Qué hacemos a una asignación de Python que está inmediatamente después de una instrucción `if` para indicar que la asignación debe ejecutarse únicamente cuando la instrucción `if` es verdadera?

1. Subrayar todo el código condicional.
2. Comenzar la instrucción con un caracter "#".
3. Sangría a la línea por debajo de la instrucción `if`.
4. Empezar la instrucción con un corchete {.

3.2 ¿Cuál de estos operadores no es un operador de comparación?

1. >=
2. !=

3. <

4. =

5. ==

3.3 Qué es correcto acerca del siguiente segmento de código :

```
if x == 5 :
    print ('Es 5')
    print ('Otra vez 5')
    print ('Tercer 5')
```

1. Dependiendo del valor de x, las tres instrucciones de la impresión se ejecutarán o bien ninguna de las instrucciones se ejecutará
2. La cadena 'Es 5' siempre se imprime sin importar el valor de **x**.
3. La cadena 'Es 5' nunca se imprime sin importar el valor de **x**.
4. Unicamente dos de las tres instrucciones de impresión se imprimirán si el valor de **x** es menor que 0.

3.4 ¿Cuando tienes múltiples líneas en un bloque `if`, cómo indicas el final del bloque `if`?

1. Colocas otra sangría a la siguiente línea más allá del bloque `if` al mismo nivel de sangría como la instrucción original del `if`.
2. Pones el símbolo de dos puntos : en una línea por sí misma para indicar que hemos terminado con el bloque `if`.
3. Usas un corchete { después de la última línea del bloque `if`.
4. Haces mayúscula la primera letra de la línea siguiente al final del bloque `if`.

3.5 ¿Cuál es la palabra reservada en Python que utilizamos si la prueba lógica es falsa?

1. `switch`
2. `else`
3. `otherwise`
4. `break`

3.6 ¿Qué se imprimirá en el siguiente código?

```
x = 0
if x < 2 :
    print ('Pequeño')
elif x <10 :
   print ("Mediano')
else :
    print ('Grande')
print ('TERMINADO')
```

1. Grande
   TERMINADO
2. Pequeño
3. Pequeño
   TERMINADO
4. Mediano
   TERMINADO

3.7 Para el siguiente código:

```
if x < 2 :
    print ('Menos de 2')
elif x >= 10 :
    print ("Dos o más")
else :
    print ('Algo más')
```

¿Qué valor de 'x' causará que se despliegue 'Algo más' ?

1. x = 2
2. x = -2
3. Este codigó nunca imprimira 'Algo más' sin importar el valor de 'x'.
4. x = -2.0

3.8 En el siguiente código ¿Cuál será la última línea que se ejecuta?

```
(1)  astr = "Hola Beto"
(2)  istr = int (astr)
(3)  print ('Primero ', istr)
(4)  astr = '123'
(5)  istr = int (astr)
(6)  print ( 'Segundo ' , istr)
```

1. 3
2. 1
3. 6
4. 2

3.9 Escriba si las siguientes condiciones son falsas o verdaderas dados los valores de las variables:

```
a = 1; b = 2; c = 3
x = True
```

Expresión lógica	Resultado
`a < b`	
`a < c & b > c`	
`a < c & x`	
`b == a | a < c`	

3.10 Las ecuaciones características de un transistor MOS son:

$$I_D = K(V_{GS} - V_T)^2 \quad \text{for} \quad V_{DS} \geq V_{GS} - V_T$$
$$I_D = K[2(V_{GS} - V_T)V_{DS} - V_{DS}^2] \quad \text{for} \quad V_{DS} < V_{GS} - V_T$$

Si $K = 0.0025, V_T = 0.25, V_{GS} = 0.2, 0.5$ y 1.0, diseñe un algoritmo que calcule el valor de I_D para el valor que se recibe de V_{DS}. Use los valores de $V_{DS} = 0.5$ y 1. Realice una tabla de resultados.

3.11 Diseñe un algoritmo que reciba un número. Si el número es entero escribir "El número es entero". Use la instrucción `floor` (en la biblioteca `math`) que redondea al siguiente entero inferior, es decir, `floor(3.49) = 3`, pero para `-2.56`, `floor( -2.56) = -3` que es el entero inmediato inferior.

3.12 Diseñe un algoritmo que reciba un número. Si el número no es entero mostrar el número redondeado hacia el entero inmediato superior. Use la instrucción `ceil` (en la biblioteca `math`) que redondea el valor al entero inmediato superior. Por ejemplo, si se recibe `4.563`, se muestra `5`, es decir, `ceil(4.563) = 4` y para `ceil(-3.4) = -3` que es el entero inmediato superior.

3.13 Diseñe un algoritmo que reciba un número. Si el número es entero escribir "El número es entero". En caso contrario escribir: "El número no es un entero."

3.14 Una máquina expendedora vende dulces cuyo costo es \$2.00. Diseñe un algoritmo para que cuando la máquina reciba una moneda de \$5.00 o \$10.00 dé la orden a la máquina de despachar un dulce y el cambio correcto.

3.15 Escriba un algoritmo que convierta un ángulo de grados a radianes y viceversa. Sabemos que las ecuaciones de conversión son:

$$\text{ángulo en grados} = \text{ángulo en radianes} \times \frac{2\pi}{180}$$

$$\text{ángulo en radianes} = \text{ángulo en grados} \times \frac{180}{2\pi}$$

Capítulo 4
Ciclos en Python

4.1 Introducción
4.2 Ciclos Mientras
4.3 Ciclos Para
4.4 Ciclos Anidados
4.5 La instrucción continue
4.6 La instrucción break
4.7 Ejemplos adicionales
4.8 Instrucciones de Python del Capítulo 4
4.9 Resumen
4.10 Ejercicios

Entre las cosas que la humanidad no ha recibido como un regalo de la naturaleza tenemos el regalo de los libros.

Herman Hesse

Objetivos

El uso de los ciclos Para y Mientras dentro de un algoritmo nos permite repetir conjuntos de instrucciones dentro de un algoritmo.

4.1 Introducción

En ocasiones es necesario repetir una instrucción o un conjunto de instrucciones ya que así lo requiere el algoritmo. En lugar de repetir el código más de una vez, podemos usar unas instrucciones que nos permiten hacer esto. Estas instrucciones se conocen con el nombre de **ciclos** y su repetición se va a realizar si se cumple alguna **expresión lógica**.

Existen dos tipos de ciclos, el ciclo **Mientras** que se repite y ejecuta mientras se cumple una condición y el ciclo **Para**.

En el ciclo **Mientras**, la expresión lógica se checa antes de iniciar el ciclo y de esta manera es posible que las instrucciones dentro del ciclo nunca se ejecuten ya que es posible que la expresión lógica nunca se cumpla.

El ciclo **Para** se repite un número determinado de veces sin tener en cuenta ninguna expresión lógica. La combinación de los ciclos de este capítulo y las condiciones del capítulo anterior permiten al diseñador de algoritmos tener herramientas muy poderosas para la realización de algoritmos complejos pero a la vez sencillos y elegantes, Cada uno de los distintos ciclos se ilustra con ejemplos. Estos ciclos y los ejemplos se realizan en pseudocódigo y en lenguaje Python. Los ciclos también se conocen como **bucles** o **lazos**.

4.2 Ciclos Mientras

Un ciclo **Mientras** causa que un conjunto de instrucciones se repita siempre y cuando se satisfaga una expresión lógica, es decir, la expresión lógica sea verdadera o True.

Una vez que la expresión lógica NO se satisface, entonces se termina el ciclo **Mientras**. Un ciclo **Mientras** tiene la siguiente estructura:

Mientras (expresión lógica):

    Conjunto de instrucciones

Fin_Mientras

Se puede ver que si la expresión lógica no se cumple al llegar al ciclo **Mientras**, entonces el conjunto de instrucciones nunca se va a ejecutar, ya que lo primero que se hace es checar la expresión lógica. Todo el conjunto de instrucciones que se desea repetir dentro del ciclo debe llevar una sangría de al menos un espacio.

Un ciclo **Mientras** solamente se ejecuta si la expresión lógica es verdadera.

Puede darse el caso de que la expresión lógica siempre sea falsa y por lo tanto NUNCA se ejecuta el ciclo.

Ejemplo 4.1

Suma de enteros del 1 al 10.

En este algoritmo se desea realizar la

$$\text{suma} = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

que nos da el resultado suma = 55.

Para esta suma de 10 números podríamos estar tentados a escribir el algoritmo usando la suma como aparece descrita arriba. Sin embargo, si se tratara de una suma del 1 a 1,000,000 entonces sería impráctico hacerlo de esta manera.

La manera práctica de hacerlo es usando un ciclo y en este caso usamos el ciclo **Mientras** descrito antes. El algoritmo empieza **inicializando** la suma a cero, es decir,

suma $\leftarrow 0$

Una suma siempre se debe inicializar antes de un ciclo.

A este tipo de variable que se usa para sumar o acumular los valores de una suma se conoce como **acumulador**.

El siguiente paso es definir una variable j cuyo primer valor es el primer entero, es decir, j ← 1 y efectuar la suma:

suma ← suma + j

En este paso tendremos que el valor de la suma es 1, ya que 0 + 1 se guarda en la variable suma. Ahora necesitamos sumar el 2, lo que hacemos incrementando el valor de j como

j ← j + 1

con lo que el nuevo valor de j es 2 y para adicionarlo al valor de suma repetimos el paso suma ← suma + j. El nuevo valor de suma es 2 + 1 = 3. Si continuamos incrementando el valor de j hasta que se llegue a j = 10 y realizando la operación suma ← suma + j, obtendremos la suma de los 10 primeros enteros.

Ahora ya podemos escribir nuestro algoritmo en pseudocódigo. El primer paso es darle un nombre al algoritmo seguido de comentarios:

Algoritmo: Suma de enteros.
Este algoritmo suma enteros del 1 a N.
El usuario proporciona el valor de N.

Las variables que se usan son el resultado que se almacena en suma, que es entero, el número entero j que se adiciona al resultado parcial y el valor de N, total de números a sumar. Al algoritmo añadimos entonces:

Variables:
Entero: j, suma, N

El siguiente paso es iniciar el algoritmo y leer o recibir el valor de N con una instrucción **Recibir:**. Las siguientes instrucciones son:

INICIO
Recibir: N

En un ciclo **Mientras** implementamos la realización de la suma, pero antes **inicializamos** el acumulador, es decir, hacemos el valor inicial de la variable suma a cero. Antes de terminar el ciclo **Mientras** es necesario incrementar el contador j para evaluar la expresión lógica nuevamente:

suma ← 0
Mientras (j <= N):
    suma ← suma + j
    j ← j + 1
Fin_Mientras

Finalmente desplegamos el resultado de suma con una instrucción **Mostrar:**

Mostrar: " La suma de los N primeros enteros es: ", suma

El algoritmo completo en pseudocódigo es el siguiente:

```
1  Algoritmo: Suma de enteros.
2  # Este algoritmo suma enteros del 1 a N.
3  # El usuario proporciona el valor de N.
4  Variables:
5  Entero: j, suma, N
6  INICIO
7  Recibir: N
8  # Inicializa j y la suma.
9  suma ← 0
10 j ← 1
11 Mientras ( j <= N):
12         suma ← suma + j
13         j ← j + 1
14 Fin_Mientras
15 Mostrar: " La suma de los N primeros enteros es: ", suma
16 FIN
```

4.2.1 El ciclo Mientras en Python

En Python el código para un ciclo **Mientras** se realiza con la instrucción `while`. En un ciclo **Mientras SE** escribe el signo de dos puntos ":" después de la expresión lógica de la instrucción `while`. El conjunto de las instrucciones que se van a repetir se escribe con sangría.

El formato de la instrucción `while` es:

```
while(expresión lógica):
    Instrucciones
```

Debemos observar las siguientes reglas:

- Después de la expresión lógica debe escribirse el signo de dos puntos.
- Las instrucciones dentro del ciclo llevan sangría[1].
- La primera instrucción sin sangría después del `while` ya no pertenece al ciclo.
- Las instrucciones solamente se ejecutan si la expresión lógica es verdadera.

El algoritmo anterior, codificado en lenguaje Python es el siguiente:

[1] Al menos un espacio de sangría pero siempre la misma sangría.

```
1  "' Programa: Suma de enteros.
2   Usa la instrucción while (Mientras).
3   Esta aplicación suma los enteros del 1 al N."'

4  #INICIO
5  # Inicializa j y la suma.
6  suma = 0
7  j = 1
8  # Recibe el valor de N
9  print ("Dame el valor de N. \n")
10 N = int(input( ) )

11 while (j <= N ):
12       suma = suma + j
13       j = j + 1 # Se incrementa el contador j.
14 print ( " La suma es: ", suma )
16 #FIN
```

Al correr este algoritmo, para el caso de N = 10, se produce la salida:

```
La suma es: 55
```

Sumarizamos la instrucción `while` de la siguiente manera:

while(condición) : ← DEBE llevar dos puntos aquí.

DEBE llevar sangría. → **Instrucciones**

Es importante observar tres puntos importantes en el ciclo **while** del ejemplo anterior:

1. Se debe inicializar el acumulador o suma previamente.
2. Se debe inicializar el contador del ciclo **while**, también antes del inicio del ciclo.
3. Una vez dentro del ciclo se DEBE incrementar el contador.

Si no se inicializa el contador, el algoritmo mostrará un resultado erroneo.

Por otro lado si no se inicializa el contador, el ciclo **while** puede no ejecutarse correctamente.

Finalmente, si no se incrementa el contador, estaremos en un ciclo que no se puede terminar ya que la expresión lógica no estaría cambiando, es decir, caemos en un ciclo infinito.

Ejemplo 4.2

Obtención de números primos.

Un número primo es aquel que solamente es divisible por sí mismo y por la unidad. Una manera de saber si un número `N` es primo es dividirlo entre los números que son menores a él desde 2 hasta `N` - 1. Se puede demostrar matemáticamente que solamente es necesario dividir hasta $\sqrt{N}$, lo que reduce el número de operaciones. Si la división tiene residuo igual a cero, entonces `N` no es un número primo. Un algoritmo que realiza esto se puede realizar con un ciclo **Mientras** de la siguiente manera:

1. Primero se lee el entero `N`.
2. Se inicializa la variable divisor igual a 2.
3. Se inicia el ciclo **Mientras** checando si divisor es menor que $\sqrt{N}$ y realizando la división `N`/divisor.
4. Si `N` = divisor*cociente, entonces `N` es divisible por divisor y `N` no es primo. En este caso nos salimos del ciclo, pero antes cambiamos el valor de divisor a `N` + 1, para indicar que ya terminamos y que el ciclo **Mientras** no se siga ejecutando.
5. Si `N` $\neq$ divisor*cociente, se incrementa el valor de divisor a `divisor` + 1 y se repite el ciclo **Mientras** para continuar checando.
6. El ciclo **Mientras** se termina cuando se cumple la condición de 4 o cuando se termina de checar los cocientes en el paso 5.
7. Una vez terminado el ciclo **Mientras** se escribe el resultado.

En pseudocódigo el algoritmo es:

```
1   Algoritmo: Primo
2   # Determinar si un entero es primo o no.

3   Variables
4   Enteros: N, divisor, cociente
5
6   INICIO
7
8   divisor ← 2 #Se inicia desde 2 y llega hasta √N a menos que antes
9                #se encuentre un divisor exacto.

10  Mientras (divisor <= sqrt(num) ):
11      cociente ← num / divisor; # Verifica con cada divisor posible.

12      Si (cociente * divisor == num)
13          divisor ← num + 1
14      si_no
15          divisor ← divisor + 1 # Tal vez sea primo, verificar siguiente.
16      Fin_Si

17  Fin_Mientras

18  Si divisor >= num # En este caso se recorrieron los posibles divisores.
19          Mostrar: "Sí es primo"
20  si_no # en este caso se encontró un divisor exacto.
21          Mostrar: "No es primo"
22  Fin_Si

23  FIN
```

En este algoritmo, la variable divisor hace las veces de contador y se incrementa cada vez que se ejecuta el ciclo. En la condición Si se tienen dos posibilidades, que la condición se satisfaga porque (cociente * divisor == num) lo que quiere decir que el número N es divisible por algún número menor que N y entonces es necesario terminar el ciclo **Mientras** haciendo N ← N + 1.

Si la comparación no se cumple se incrementa el divisor y se continúa con otra repetición del ciclo **Mientras**. El programa en Python es:

```
1  # Dado un número entero positivo, determinar si es primo.

2  print ( "Numero a evaluar ? " )
3  n = int(input( ) )
4  divisor = 2 # se iniciará en 2 y llegará hasta la raíz de num,
5        # a menos que antes se encuentre antes un divisor exacto.
6  while ( divisor <= sqrt (n)):
7      cociente = n/divisor # Verifica con cada divisor posible.
8      if (n == cociente * divisor ):
9          divisor = n+1 #  Se termina porque fue divisible.
10     else:
11         divisor = divisor + 1  # Tal vez sea primo,
12                                # verificar siguiente.

13 if (divisor > n ): # En este caso se encontró un divisor exacto.
14     print( " NO es primo" )
15 else:
16     print("SÍ es primo") # Se recorrieron los posibles divisores.
```

El resultado es, para dos números que sabemos que son primo y no primo o compuesto, respectivamente, el siguiente:

```
Numero a evaluar ? 11
SÍ es primo.

Numero a evaluar ? 8
NO es primo.
```

El siguiente algoritmo es una modificación del algoritmo anterior pero ahora usamos variables lógicas o booleanas. Como sabemos, las variables lógicas o booleanas solamente pueden tomar los valores verdadero y falso, `True` y `False` en Python.

Ejemplo 4.3

Otro algoritmo para números primos

Un algoritmo que también puede diseñarse para encontrar si un número es primo emplea variables lógicas o booleanas para distinguir si es primo o no lo es. De esta

manera, introducimos una variable lógica llamada `es_primo` que inicializamos a verdadera (`True` en Python) y que cambia a falsa (`False` en Python) cuando el número es divisible por algún entero menor que `N`.

Para comparar escribimos el algoritmo anterior en pseudocódigo modificado para usar las variables lógicas y a un lado escribimos el algoritmo en Python:

	Pseudocódigo	Python
	Algoritmo: Primo.	"' Programa: Primo.
1	# Evalúa si un entero es primo.	Evalúa si un entero es primo"'
2	**INICIO**	#**INICIO**
3	**Variables:**	
4	**entero** N, divisor←2, cociente	`divisor = 2`
5	**lógica** es_primo ← **verdadero;**	`es_primo = True`
6	**Mientras** n<=sqrt(N)∧es_primo:	`while n <=sqrt(N) & es_primo:`
7	cociente ← num / divisor;	`cociente = num / divisor`
8	**Si** cociente*divisor = num:	`if cociente*divisor==num :`
9	es_primo ← **falso**	`es_primo = false`
10	**Fin_Si**	
11	divisor ← divisor + 1	`divisor = divisor + 1`
12	**Fin_Mientras**	
13	**Si** (es_primo)	`if (es_primo):`
14	**Mostrar:** "SI es primo"	`print ("Si es primo")`
15	**si_no**	`else:`
16	**Mostrar:** "No es primo"	`print ("NO es primo")`
17	**Fin_Si**	
18	**FIN**	# **FIN**

Como vemos, la comparación para checar si es primo se hace sobre la variable lógica `es_primo`. El resultado para los números a evaluar 11 y 8 es el mismo que antes.

4.3 Ciclos Para

Los ciclos **Para** son ciclos en los cuales la instrucción o el conjunto de instrucciones se repiten un **determinado** número de veces. Por esto es necesario tener un contador para llevar la cuenta de las veces que se repite el ciclo **Para**. El formato en pseudocódigo es:

La estructura del ciclo **Para** es:

Para contador ← valor_inicial **hasta** valor_final; incremento **:**

Instrucciones

FinPara

IMPORTANTE: Se debe usar el mismo contador dentro del ciclo **Para**.

En el caso de ciclos **Para**, no es necesario inicializar el contador ni incrementarlo. Estas acciones las ejecuta automáticamente el ciclo **Para**. El incremento por lo general es la unidad, pero puede ser distinto de la unidad o negativo. El conjunto de Instrucciones dentro del ciclo **Para** se ejecuta cada vez que se incrementa el valor del contador y **hasta** que se cumple el contador llega al valor_final indicado.

En Python el ciclo **Para** usa la palabra clave **for** y tiene la estructura:

La estructura del ciclo **for** es:

```
for contador in lista :

      Instrucciones
```

En este caso el ciclo **Para** solamente repite las instrucciones que **tienen una sangría**. Los valores de la variable `contador` son aquellos que se encuentran en la `lista`. Deben observarse los dos siguientes puntos para el ciclo for:

- Debe escribirse el signo de dos puntos antes de las instrucciones del ciclo.
- **NO** deben usarse paréntesis en la línea del **for**.
- Las instrucciones del **for** deben llevar sangría.

Un par de ejemplos nos ilustra la manera de usar el ciclo **Para**.

Ejemplo 4.4

Realizar la suma de los enteros del 1 al 10.

El pseudocódigo del algoritmo que suma los 10 enteros positivos del 1 al 10 requiere los siguientes pasos:

- En primer lugar, definir las variables que se van a usar.
- Realizar la inicialización de la suma.
- Usar un ciclo **Para** que efectúa la suma.
- El `contador` se puede usar como uno de los sumandos.

El pseudocódigo es:

```
1   Algoritmo: Suma de enteros.
2   # Este programa suma los enteros del 1 al 10.

3   Variables:
3   Entero: suma, contador

5   INICIO
6   suma ← 0 # Inicializa la suma.

7   Para contador ← 1 hasta contador <=10 ; incremento ← 1 :
8           suma ← suma + contador
9   Fin_Para

11  Mostrar: "La suma es:", suma
12  FIN
```

La realización de este algoritmo en lenguaje de Python nos produce el siguiente programa:

```
1   ''' Suma de enteros del 1 al 10.
2    Este algoritmo usa un ciclo Para.
3    Este algoritmo suma los enteros del 1 al 10.'''

4   INICIO
5   # Inicializa la suma.
6   suma = 0
7   lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

8   for contador in lista :
9         suma = suma + contador

10  print("La suma es: " , suma)

11  FIN
```

El resultado de este programa es:

```
suma = 55.
```

Notamos lo siguiente en el ciclo **Para** implementado en Python:

- La condición NO DEBE estar dentro de paréntesis.
- No es necesario inicializar el contador ya que se hace dentro de la primera instrucción dentro de la condición, en este caso es **contador = 1**, que es el primer valor de la lista.
- El último valor de la lista indica hasta que valor del contador se realizará el ciclo, que es **contador = 10**. Significa que el ciclo se repite mientras **contador** sea menor o igual a 10.
- El contador se incrementa en la unidad de acuerdo al valor de la lista.

Ejemplo 4.5

Número de vocales de una cadena o string

Se desea encontrar el número de vocales de una cadena llamada `frase`. Para hacer esto examinamos cada caracter de la cadena para saber si es vocal ya sea mayúscula o minúscula.

El total de vocales se va contando en la variable `cuenta`. Esto lo hacemos con un ciclo `for` comparando si cada caracter de una cadena está en la cadena `base` = "`aeiouAEIOU`". La cadena que deseamos examinar está en la variable:

```
frase = "El perro corre despacio"
```

El ciclo **for** es entonces:

```
1  """Programa: Calcula el número de vocales en una frase."""

2  frase = "El perro corre despacio"
3  base = "aeiouAEIOU"
4  cuenta = 0

5  for i in frase:# Para cada caracter en palabra.
6     if i in base  # Checa si el caracter está en la base.
7        cuenta = cuenta + 1

8  print("El número de vocales es ", cuenta)

9  FIN
```

Si usamos la frase "`El perro corre despacio`" obtenemos el resultado:

```
El número de vocales es 9
```

Ejemplo 4.6

Crecimiento de una colonia de bacterias

Se desea encontrar el número de días en que la población de una colonia de bacterias llegará a un cierto valor máximo `Mmax`. Sabemos que la población se duplica cada dos días. Para encontrar el valor máximo usamos un ciclo **Mientras**. Dentro de este ciclo usamos una condición para saber si se alcanzó el valor máximo. Finalmente, mostramos el número de días y la población final de la colonia de bacterias.

El algoritmo es el siguiente:

- Inicializamos la población inicial de bacterias a M_0.
- Inicializamos el número de días nd $\leftarrow 0$.
- Recibir el valor máximo de la población de la colonia de bacterias `Mmax`.
- Dentro de un ciclo **Mientras** calcular el incremento a la población de bacterias y sumarlo al valor previo.
- Cuando la población de bacterias alcanza el valor deseado `Mmax` salir del ciclo **Mientras**. Esto lo hacemos con una condición.
- Mostrar el número de días.
- Fin del algoritmo.

El algoritmo en pseudocódigo es:

```
1   Algoritmo:"Población de una colonia de bacterias."
2   Se calcula el número de días en que la población de
3   una colonia de bacterias se duplica.
4   Variables:
5   Entero: nd, k # nd es el número de días, k es un contador.
6   nd ← 1
7   M0 ← 10
8   Mmax ← 20
9   Mientras( M0 <= Mmax ):
10     M0 ← 2*M0 # M0 tiene la población de bacterias.

11     Si M0 <= Mmax : # Checa si todavía no se llega al máximo.
12         nd = nd + 2 # Se incrementa el número de días en 2.
13     Fin_Si

14  Fin_Mientras
15  Mostrar: "El número de días es ", nd
16  Mostrar: "La población de bacterias es ", M0

17  FIN
```

El algoritmo en Python es:

```
1   "' Programa:"Población de una colonia de bacterias."
2   Se calcula el número de días en que la población de
3   una colonia de bacterias se duplica."'
4   nd = 1
5   M0 = 10
6   Mmax = int(input("Dame la población máxima: \n" ))
7   while ( M0 <= Mmax ):
8      M0 = 2*M0
9      if M0 <= Mmax : # Checa si todavía no se llega al máximo.
10        nd = nd + 2 # Se incrementa el número de días en 2.
11  print("El número de días es ", nd)
12  print("La población de bacterias es ", M0)
13  # FIN
```

Con este programa obtenemos para `Mmax = 200` el resultado:

```
El número de días es 9
La población de bacterias es 320
```

4.3.1 La función range

Otra función que se usa en lugar de una lista es la función `range` que tiene el formato:

```
range( a, b, c )
```

Esta función crea una lista que empieza en a, termina antes de b, c es el incremento y los valores de la lista generada son:

[a, a + c, a + 2c, a + 3c, ...,]

En esta función el segundo valor b NO forma parte de la lista. Algunos ejemplos son:

```
range( 5 )          produce [ 0, 1, 2, 3, 4 ]
range( 2, 6 )       produce [ 2, 3, 4, 5 ]
range( -1, 7, 2)    produce [-1, 1, 3, 5 ]
range(3, 9, 2)      produce [ 3, 5, 7 ]
range(8, 4, -1)     produce [ 8, 7, 6, 5 ]
range(-8, -13, -1)  produce [ -8, -9, -10, -11, -12 ]
```

En todos los casos el último valor de la lista NO es b. Los dos últimos ejemplos muestran decremento. El valor del incremento NO puede ser cero.

Para desplegar los valores de una lista generada con `range` usamos, la instrucción `list`, por ejemplo:

```
>>> list( range (4 ) ) # El número 4 no forma parte de la lista.
```

Para el Ejemplo 4.4, se puede cambiar la instrucción `for` usando la instrucción `range` como range(1, 11), ya que esto es equivalente a la lista de números del 1 al 10. No se incluye el último número que es el 11. El `for` del algoritmo queda de la siguiente manera:

```
suma = 0

for contador in range(1, 11) :
        suma = suma + contador
```

4.4 Ciclos anidados

Existen algoritmos que requieren anidar ciclos, es decir, que dentro de un ciclo exista otro ciclo.

Se pueden anidar los dos tipos de ciclos, por ejemplo, se pueden anidar ciclos **Para** dentro de ciclos **Para** o **Mientras**, o cualquier combinación de ciclos. El primer ciclo recibe el nombre de ciclo exterior y a los ciclos que están dentro del ciclo exterior se les denomina los **ciclos anidados** o **ciclos interiores**.

La estructura de los ciclos anidados es:

Ciclo1
    Instrucciones 1.

    Ciclo2
        Instrucciones 2.

        Ciclo 3
            Instrucciones 3.

            # Terminan las instrucciones dentro del ciclo3.
        Fin_Ciclo3

        # Terminan las instrucciones dentro del ciclo2.
    Fin Ciclo2

    # Terminan las instrucciones dentro del ciclo1.
Fin Ciclo1

Como se puede ver, dentro del Ciclo 1 se encuentra anidado el Ciclo 2 y dentro del Ciclo 2 se encuentra anidado el Ciclo 3. Cada ciclo anidado debe estar contenido completamente dentro de un ciclo exterior. Con ejemplos ilustramos los ciclos anidados.Tratar de terminar un ciclo anidado fuera de otro ciclo exterior provoca un error, independientemente de si es un ciclo `for` o un ciclo `while`.

Ejemplo 4.7

Cálculo de calificaciones

En este ejemplo usamos un ciclo **Para** y un ciclo **Mientras** anidados para leer las calificaciones de un grupo de alumnos. El algoritmo debe ser capaz de leer las calificaciones de tres alumnos de los 6 meses del semestre. Para este ejemplo suponemos que tenemos tres estudiantes de nombres Pedro, María y Alejandro. Las calificaciones que obtienen en los 6 meses del semestre son:

Nombre	**1**	**2**	**3**	**4**	**5**	**6**
Pedro	8	9	7.8	8.1	7.2	9.5
María	8.6	10	10	9.4	9.7	9.3
Alejandro	9.5	6.4	8.9	8.9	9.9	10

El algoritmo debe leer el nombre de cada alumno para luego proceder a leer las calificaciones de cada alumno y entregar su promedio. Al terminar el primer alumno hacer lo mismo con el segundo y tercer alumnos. Los pasos a seguir son los siguientes:

1. Leer el nombre del estudiante.

   Para contar los estudiantes usamos un contador de estudiantes.

2. Leer una por una las calificaciones.

   Sumarlas y sacar el promedio.

   Para contar las seis calificaciones usamos un contador de calificaciones.

   Desplegar el promedio.

3. Repetir el proceso de los pasos 1 y 2, para cubrir todos los estudiantes y todos los meses del semestre.

Para el ciclo `Para` usamos la función `range`. El algoritmo es entonces:

```
1   Algoritmo: Promedios de calificaciones.
2      "' Este algoritmo lee calificaciones de alumnos y
3      calcula el promedio del semestre."'

4   Variables:
5   Real: suma, calificación, promedio
6   Entero: cont_est, contador_calificacion, No_est, No_califs
7   Cadena: Nombre_est

8   INICIO
9   Mostrar: "Dame el numero de alumnos: "
10  Recibir: No_est
11  Mostrar: "Dame el numero de calificacions por alumno: "
12  Recibir: No_califs

13  # Principia el ciclo exterior.
14  Para cont_est ← 0 hasta cont_est <=No_est; incremento = 1:
15        # Inicializa variables del ciclo Mientras.
16        Mostrar: "Dame el nombre del estudiante "
17        Recibir: Nombre_est
18        contador_calificacion ← 1
19        suma ← 0

20        # Principia el ciclo anidado.
21        Mientras (contador_calificacion <= No_califs):
22                Mostrar: "Dame una calificación: "
23                Recibir: calificación
24                suma ← suma + calificación
25                contador_calificacion ← contador_calificación + 1
26        Fin_Mientras # Se termina el ciclo Mientras.

27        # Se calcula el promedio de un estudiante.
28        promedio ← suma/No_califs
29        Mostrar: 'El promedio del estudiante', No_est, 'es:', promedio
30  Fin_Para
31  FIN
```

Este algoritmo también se puede realizar usando cualesquiera de los dos tipos de ciclos. Por ejemplo, se pueden usar dos ciclos **Mientras** o dos ciclos **Para**. La realización en Python para este algoritmo que usa un ciclo **Para** con un ciclo **Mientras** anidados es:

```
1  '''Programa: Promedio de calificaciones. Este programa calcula
2  el promedio de calificaciones de tres alumnos
3  durante un semestre.'''

4  No_est=int(input("Dame el número de alumnos:\n"))
5  No_califs=int(input("Dame el No. de califs por alumno: \n"))

5  # Principia el ciclo exterior.
6  for cont_est in range ( No_est ):
7     # Inicializa variables del ciclo Mientras.
8     print("Dame el nombre del estudiante." )
9     Nombre_est = input ( )
10    contador_calificacion = 1
11    suma = 0.0

12    # Principia el ciclo anidado.
13    while (contador_calificacion <= 6):
14       print("Dame una calificacion." )
15       calificacion = float(input())# La calif. puede ser real.
16       suma = suma + calificacion
17       contador_calificacion += 1
18        # Fin del ciclo Mientras.

19    # Se calcula el promedio de un estudiante.
20    promedio = suma/3.0
21    print("Promedio del est. " , Nombre_est , "es:" , promedio)
22 # Termina ciclo for.
23 # FIN
```

Si seguimos este algoritmo paso a paso encontramos que los promedios de los tres alumnos son:

Pedro	8.27
María	9.5
Alejandro	8.93

Ejemplo 4.8

Algoritmo para calcular los valores de una función

En este algoritmo deseamos calcular los valores de una función $f(x, y, z)$ para los siguientes valores de x, y, z:

Valores de: $x = 0, 1, 2, 3, 4$; $y = 0.6, 0.9, 1.2, 1.5$; $z = -1, 2, 5, 8$.

donde $f(x, y, z) = 3x^3 - 7y^2z$

Desarrollo del algoritmo:

Se tienen 5 valores de x, 4 valores de y, 4 valores de z. Una manera de implementar el algoritmo es con tres ciclos Mientras. También es posible usar otra combinación de ciclos. Usaremos un ciclo **Para** para los valores de x, un ciclo **Mientras** para los valores de y y un ciclo **Mientras** para los valores de z. Notamos que los valores de x cambian de 1 en 1 empezando de 0 hasta llegar a 4, los de y son 4 valores que cambian de 0.3 en 0.3 empezando en 0.6 y terminando en 1.5, y los valores de z son cuatro y varían de 3 en 3 empezando en -1 y terminando en 8.

La estrategia consiste en los siguientes pasos.

- Declaramos los valores de x, y, z, f como reales.
- Empezar con el valor de $x = 0$.
- Calcular f para todos los valores de y.
- Calcular f para todos los valores de z.
- Continuar con el siguiente valor de x que es $x = 0 + 1 = 1$ y calcular f para este nuevo valor de x para cada valor de y y para cada valor de z.
- Repetir el paso 2 hasta que se terminen los valores de x.
- En cada paso por los tres ciclos anidados se debe escribir el valor de f.

El algoritmo en pseudocódigo es:

```
1  ALGORITMO: Cálculo de f(x, y, z).
2  # Este algoritmo calcula los valores de f(x, y, z) = 3*x∧3 + 7*y∧2*z.
3          Variables:
4          Real: f, x, y, z
5  INICIO
6  Para (x ← 0; x <= 4; incremento ← 1):
7          y ← 0.6 # Se inicializa y a 0.6 que es su primer valor.
8          Mientras (y <= 1.5):
9                  z ← −1 # Se inicializa z a -1 para que sea
10                        # su primer valor.
11                 Mientras ( z <= 8):
12                         f ← 3*x∧3 + 7*y∧2*z
13                                 z ← z + 3; #se incrementa y.
14                 Fin_Mientras
15                 y ← y + 0.3 # se incrementa y.
16                 Mostrar: "El valor de f(", x, ", ", y, ", ", z, ") es:", f
17         Fin_Mientras
18 Fin_Para
19 FIN
```

Al codificar este algoritmo en Python obtenemos lo siguiente:

```
1  "' Programa: Cálculo de f(x, y, z).
2  Este algoritmo calcula
3  los valores de f(x, y) = 3x∧ 3 + 7y∧ 2*z."'
4  # Importa la biblioteca matemática.
5  from math import *
6  for x = 0 in range(5):
7     y = 0.3 # Se inicializa y.
8     while (y <= 1.5):
9        z = -1 # Se inicializa z.
10       while ( z <= 8):
11          f = 3*pow(x, 3)+7*pow(y, 2)*z
12          print ("f( x , y , z ) = ", f )
13          z = z + 3;  #se incrementa z después de calcular f.
14       # Fin del while anidado
15       y = y + 0.3 # se incrementa y después de calcular f.
16
17     # Fin del while (Mientras).
18 # Fin del for (Para).
19 FIN
```

Ejemplo 4.9

Triple Sumatoria

Se desea diseñar un algoritmo para calcular la triple sumatoria

$$S = \sum_{i=0}^{10} \sum_{j=5}^{9} \sum_{k=-1}^{7} i * j * k + 23$$

y al final mostrar el valor de S. El cálculo de cada suma requiere valores de i, j y k. Cada sumatoria se puede realizar con un ciclo **Para**. De esta manera tendremos tres ciclos **Para** anidados.

Como se usan ciclos **Para**, no se requiere inicializar las variables de los ciclos pero sí se necesita inicializar la suma con S ← 0. Los ciclos son:

```
1   Algoritmo: Triple sumatoria.
2   Variables:
3   Real: S
4   Entero:i, j, k
5   INICIO

6   S ← 0
7   Para (i ← 0 hasta i <= 10):
8           Para(j ← 5 hasta j <= 9):
9                   Para (k ← -1 hasta k <= 7):
10                          # El cálculo de la sumatoria es:
11                          S ← S + i*j*k
12                  Fin_Para
13          Fin_Para
14  Fin_Para
15  S ← S + 23

16  FIN
```

Finalmente, se muestra S con:

Mostrar: "El valor de S es: ", S

El algoritmo codificado en Python es el siguiente:

```
1  # Algoritmo que realiza una triple sumatoria.

2  # INICIO
3  S = 0 # Se inicializa la suma a cero.
4  # Se inician los ciclos anidados.
5  for i in range(11):
6      for j in range(5, 10):
7          for k in range(-1, 8):
8              #El cálculo de la sumatoria es:
9              S = S + i*j*k
10         # Fin del for de k.
11     # Fin del for de j.
12 # Fin del for de i.

13 S = S + 23 # Se le suma 23 a la suma.
14 print ("El valor de S es:", S )

15 # FIN
```

Al ejecutar el algoritmo obtenemos el valor de la suma como `S = 51998`.

Ejemplo 4.10

Cálculo de promedio de calificaciones.

Se desea modificar el algoritmo del Ejemplo 4.7 para calcular el promedio y el porcentaje de alumnos aprobados. Para aprobar se requiere que el promedio sea mayor de 7.5. La lectura de calificaciones termina cuando se recibe una calificación negativa, la que no se considera para los cálculos.

Como ahora tenemos que indicar el porcentaje de alumnos aprobados tenemos que usar otra variable real para este cálculo así como el número de aprobados para poder calcular el porcentaje de los aprobados. Definimos entonces estas variables con:

Enteros: numero_aprobados;

Real: porcentaje_aprobados, promedio_aprobados;

El algoritmo se modifica haciendo la comparación para saber si aprueba y de ser así se suma a la cantidad de alumnos aprobados junto con su calificación para calcular el promedio de los alumnos aprobados. Para probar el algoritmo añadimos un nuevo alumno a la lista del Ejemplo 4.7 de la siguiente manera:

Nombre	**1**	**2**	**3**	**4**	**5**	**6**
Pedro	8	9	7.8	8.1	7.2	9.5
María	8.6	10	10	9.4	9.7	9.3
Alejandro	9.5	6.4	8.9	8.9	9.9	10
Juan	6	9	5	8	7	7

El algoritmo se modifica entonces de la siguiente manera:

- Cada alumno puede tener distinto número de calificaciones.
- Se debe parar la lectura de calificaciones cuando se recibe una calificación negativa, la que no se contabiliza.
- Se calcula el promedio y se comprueba si aprueba. De ser así se suma su promedio al promedio de aprobados y se cuenta dentro del número de aprobados para calcular el promedio total y el porcentaje de aprobados.

El algoritmo en pseudocódigo es entonces:

```
1   Algoritmo: Promedios de calificaciones mejorado.
2      "'Este algoritmo lee calificaciones de alumnos y
3       calcula el promedio del semestre y el porcentaje de aprobados."'

4   Variables:
5   Real: suma, calificación, promedio
6   Real: porcentaje_aprobados, promedio_aprobados
7   Entero: cont_est, contador_calificacion, No_est
8   Entero: No_califs, número_aprobados
9   Cadena: Nombre_est

10  INICIO
11  calificación ← 10
12  Mostrar: "Dame el número de alumnos: "
13  Recibir: No_est

14  # Principia el ciclo exterior.
15  Para cont_est ← 1 hasta No_est
16          # Inicializa variables del ciclo Mientras.
17          Mostrar: "Dame el nombre del estudiante "
18          Recibir: Nombre_est
19          contador_calificación ← 1
20          suma ← 0

21          # Principia el primer ciclo anidado donde se leen las califs.
22          Mostrar: "Dame una calificación: "
23          Recibir: calificación
24          # Inicia el ciclo anidado.
25          Mientras (calificación >= 0):
26                  Mostrar: "Dame una calificación: "
27                  Recibir: calificación
28                  suma ← suma + calificación
29                  contador_calificación ← contador_calificación + 1
30          Fin_Mientras # Se termina el ciclo Mientras.

31          # Se calcula el promedio de un estudiante.
32          promedio ← suma/No_califs
33          Mostrar: 'El promedio del estudiante', Nombre,'es:',promedio
34  Fin_Para
35  FIN
```

En este algoritmo vamos a recibir la calificación de cada alumno y la vamos a sumar con las demás calificaciones para obtener la suma total.

Si la calificación leída es menor a 7.5 incrementamos el número de alumnos reprobados. Adicionalmente debemos mostrar el número N de calificaciones.

4.5 La instrucción continue

La instrucción **continue** se usa dentro de un ciclo **Para** asociado a un ciclo `Si` y no ejecuta el renglón de código que está después de que aparece la instrucción `continue`. Ilustramos su uso con un ejemplo. Por lo general, se usa después de una condición `if`.

Por ejemplo, supongamos que queremos escribir los números pares del 0 al 10 usando un ciclo `for`. Como un número par es divisible entre dos, su residuo es cero y SÍ lo escribimos, pero para un impar su residuo es 1 y NO lo escribimos usando el **continue**.

Ejemplo 4.11

Uso de `continue`.

Consideremos el caso de escribir números pares del 0 al 10. Mediante el cálculo del residuo de a/2 sabemos que si el residuo es cero entonces tenemos que a es número par. Pero si el residuo no es cero el número a es impar. Esto lo podemos hacer si en el algoritmo hacemos los siguientes pasos:

1. Dentro de un ciclo **Para** recorrer los valores del contador 1 a 10.
2. Calcular el residuo de la división de **contador/2** usando la función módulo que es **residuo%2.**
3. Si el residuo es cero se trata de número par y lo desplegamos.
4. Si el residuo es distinto de cero el número es impar y no lo deplegamos.
5. Para no desplegarlo usamos la instrucción `continue`.

El algoritmo en pseudocódigo es:

1. Iniciar un ciclo **Para** donde **x** empieza en cero y termina en 10.
2. Calcular el residuo de la división entre x y 2 y checar si es cero o uno.

3. Si el residuo es la unidad NO escribir el número (porque es impar) y si es cero escribir el número (porque es par). Por ejemplo, $7\,\%2 = 1$ y NO se escribe (es impar) pero $8\,\%2 = 0$ y SÍ se escribe (es par).

El algoritmo completo en pseudocódigo es:

```
1   Algoritmo: Selección de números pares.

2   Variables:
3   Entero: x

4   INICIO
5   Para x ← 1 hasta x <= 10:
6       Si (x %2) # Se calcula el residuo para saber si es par
7           continuar
8       Mostrar: , x
9       FinSi
10  Fin_Para

11  FIN
```

y en Python

```
1  # Algoritmo para seleccionar números pares.

2  for x in range(1, 10):
3     if(x%2):
4         continue  # Se salta la siguiente instrucción si x no es par.
5                   # El residuo de x/2 ≠ 0.
6     print( x, "\n")
7  # Fin del ciclo for.
```

4.6 La instrucción break

El uso de **break** en el ciclo Para interrumpe el ciclo, es decir, lo termina. En el caso de ciclos anidados solamente termina el ciclo donde se encuentra la instrucción **break**.

Ejemplo 4.12

Uso de la instrucción break.

Supongamos que iniciamos un ciclo donde x empieza en cero y termina en 10, deseamos escribir los impares y además interrumpir el ciclo una vez que tengamos el primer impar divisible por 5. Usamos `continue` para escribir solamente los impares y `break` para salir del ciclo Para. El algoritmo es el siguiente en pseudocódigo y en Python:

```
1   "'Algoritmo para seleccionar impares y terminar al encontrar
2   el primero divisible por 5.'"

3   Variables:
4   Entero: x

5   INICIO

6   Para x ← 1 hasta x <= 10
7      Si ( ! x % 2 ) && ( x % 5 ):
8         continuar
9      FinSi

10     Si ( x % 5 ):
11        break # Se sale del ciclo Para
12     FinSi

13     Mostrar: x

14  FinPara

15  FIN
```

En Python el algoritmo es:

```
1  "'Algoritmo para seleccionar impares y
2   terminar al encontrar el primero divisible por 5."'
3
4  for x in range(1, 10):
5       if( not(x%2) & (x%5)):
6           continue
7       if(x% 5 == 0):
8           break
9       print( x ,"\n")
```

4.7 Ejemplos adicionales

Esta sección presenta ejemplos adicionales de ciclos y ciclos anidados.

Ejemplo 4.13

Cálculo del factorial de un número entero.

Supongamos que deseamos calcular el factorial de un número entero n el cual se denota por $n!$ y se define como

$$n! = 1 \times 2 \times 3 \times 4 \cdots \times n - 1 \times n$$

Si deseamos calcular el factorial de 5, una manera de realizar el factorial 5 es la siguiente:

```
factorial_de_n = 2*3*4*5
print( " El factorial es: " , factorial_de_n )
```

Al ejecutar este algoritmo el resultado es `factorial_de_n = 120` que es el resultado esperado. Pero imaginemos que deseamos calcular el factorial de 20, calcularlo como en el algoritmo anterior sería muy complicado y habría que modificar el algoritmo cada vez que cambiamos el valor del entero n. Para solucionar este problema usamos los valores de factorial para los números previos. Este concepto emplea un valor anterior para obtener el resultado final. Una manera alterna de calcular el factorial de un entero es usar un ciclo **Para** y multiplicar un valor inicial del factorial (`factorial_de_n = 1`) por el entero siguiente (`factorial_d_n = factorial_d_n *k`) donde `k` es la variable del ciclo **Para**. De esta manera podemos implementar el algoritmo en el siguiente programa:

```
factorial_de_n = 1
print( "Dame el valor del entero n: \n")
n = int(input( ))
# Se inicia el cálculo del factorial de n.
for k in range(2, n+1):
      factorial_de_n = factorial_de_n*k
print( " El factorial es: ", factorial_de_n )
```

Si corremos el algoritmo con n = 5 nos da el mismo resultado anterior. Ahora, sin cambiar el algoritmo podemos calcular el factorial de otro entero arbitrario. En el capítulo 7 tratamos otra vez el tema del cálculo del factorial de un número entero usando el concepto de recursividad usando funciones.

Ejemplo 4.14

Cálculo de la raíz cuadrada.

Para calcular la raíz cuadrada las calculadoras usan algoritmos de métodos numéricos. Un algoritmo para calcular la raíz cuadrada de un número real a es el siguiente:

Si x es el valor de la raíz cuadrada, calculamos dentro de un ciclo **for** la siguiente ecuación:

$$x \leftarrow \frac{1}{2}\left(x + \frac{a}{x}\right)$$

El ciclo **for** lo ejecutamos desde k = 1 hasta k = 10. Para ver la exactitud imprimimos el resultado en cada iteración. El algoritmo es el siguiente:

```
x = 1.0 # Inicializamos x a la unidad.
a = float(input( "Dame el valor de a:\n" ))

# Se inicia el cálculo de la raíz de a.
for k in range(1, 10):
   x = ( x + a/x)/2
   print( " La raíz despues de ", k, " iteraciones es ", x )
```

El resultado que obtenemos es para a = 23:

```
Dame el valor de a:
23
La raíz después de 9 iteraciones es 4.795831523312719
```

Sabemos que la raíz usando la función `sqrt` es: 4.795831523312719 por lo que el error es: 0.0.

Ejemplo 4.15

Dibujo de un cuadrado.

En este ejemplo mostramos el uso de los ciclos anidados para dibujar un cuadrado en la pantalla usando asteriscos. El usuario indica el número de asterisco que cada lado del cuadrado puede tener. Una manera de hacerlo se describe en el siguiente algoritmo:

1. Primero se lee el número de asteriscos de cada lado. Se almacenan en N.

2. Luego se escribe el total de asteriscos del lado de arriba. Esto se puede hacer con un ciclo. Para esto usamos un ciclo Mientras.

3. Luego escribimos los lados laterales que tienen una longitud *N-2* ya que los lados de arriba y de abajo se escriben aparte. Esto lo hacemos con un ciclo Para y con condiciones se deja en blanco el centro del cuadro.

4. Finalmente se escriben los asteriscos del lado de abajo. Esto lo hacemos con un ciclo Mientras.

El algoritmo en pseudocódigo es:

Pseudocódigo

```
Algoritmo: Ec. primer grado. # Ec. de primer grado.

INICIO
Entero: N # Dimensión del cuadrado.
Entero: i, k, j
# i Iteración para dibujar las partes superior e inferior.
# k Iteración de columnas para el dibujo de las literales.
# j Iteración de filas para el dibujo de las literales.
# Pedimos las dimensiones del cuadrado.
Mostrar: "Escribe la dimensión N >= 2 del cuadrado a dibujar: "
Recibir: N

Para i ← 0 hasta N:
# Escribe la primera línea del cuadrado.
        Mostrar: '*' # Termina ciclo Para.
Mostrar: "\n" # Salta a segundo renglón.
# Dibuja las partes laterales del cuadrado en un ciclo Mientras.

Mientras(j <= N - 2):
        Para k ← 0 hasta N - 1:
            # Dibuja el primer asterisco.
            Si(k == 0):
                Mostrar: "* " #Dibuja el asterisco final.
            Si_no( k == N - 1):
                Mostrar: "*" # Dibuja espacios dentro del cuadrado.
            Si_no (k != 0)
                Mostrar: " "
      # Termina Para.
      j ← j + 1
      Mostrar: "\n"
# Termina ciclo Mientras.

# Dibuja la parte inferior del cuadrado.
i ← 0
Mientras ( i < N + 1 ):
    Mostrar: "* "
    i ← i + 1
#Termina ciclo Mientras.
FIN
```

El algoritmo en Python es el siguiente:

Python

```
1  "' Este algoritmo dibuja un cuadrado con asteriscos."'
2  N = int(input("Escribe la dimensión N>= 2 del cuadrado a dibujar"))
3  # Escribe la primera línea de asteriscos.
4  for i in range(N+1):
5      print('* ', end = '')
6
7  print( )
8
9  # Dibuja las partes laterales
10 j = 1
11 while j <= N - 2 :
12     for k in range(N):
13         if k == 0::
14             print('* ', end = '') # * + un espacio.
15         elif k == N-1:
16             print('  *', end = '') # Dos espacios + *.
17         else:
18             print('  ', end = '') # Dos espacios.
19         # Fin del if:
20     print( )
21     j += 1

22 # Fin del ciclo while.

23 #Dibuja el lado de abajo .
24 i = 0
25 while i <N + 1 :
26     print('* ', end = '') # * + un espacio.
27     i += 1

28 # FIN del algoritmo
```

Los resultados de correr el algoritmo se muestran en la figura 4.1.

```
Escribe la dimensión del cuadrado N:16
* * * * * * * * * * * * * * * *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
*                             *
* * * * * * * * * * * * * * * *
```

Figura 4.1 Dibujo del cuadrado cuando N = 16.

.

Ejemplo 4.16

Generación de números aleatorios

La generación de números aleatorios requiere importar la biblioteca `random`. La generación de números enteros aleatorios se realiza con la función `randint(a, b)` que genera un número aleatorio entre los enteros `a` y `b`. Adicionalmente, generar diez números aleatorios requiere el uso de un ciclo **Para**.

En todo lenguaje de programación, los llamados números aleatorios son en realidad pseudoaleatorios, lo que quiere decir que son generados con una función o subalgoritmo de manera determinística. En el caso de Python, se usa un generador que se conoce como "Mersenne Twister" que fue desarrollado por M. Matsumoto y T. Nishimura en 1998[2].

En este ejemplo deseamos generar 10 números aleatorios enteros entre 0 y 20.

Los pasos a seguir son:

1. Por medio de un ciclo generar los números aleatorios.
2. Almacenarlos en una lista.

[2] M. Matsumoto y T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Trans. Modeling and Computer Simulation, vol. 8, No. 1, pp. 3-30.

El siguiente algoritmo codificado en Python calcula 10 números aleatorios entre 0 y 20. Las instrucciones son:

```
1  import random                    # Importa la biblioteca de random.
2  num = [ ]                        # Inicializa la lista.
3  for i in range(10):              # Para los primeros 10 enteros de 0 a 9.
4      n = random.randint(0, 20)    # Se genera un número aleatorio.
5      num.append( n )              # Se aumenta la lista num.
6  print( num )                     # Se imprime el resultado.
```

Un resultado se muestra a continuación:

```
[18, 16, 14, 18, 13, 21, 15, 15, 11, 16]
```

Los resultados pueden variar de computadora a computadora y de corrida a corrida por la característica aleatoria de la función `random`.

4.8 Instrucciones de Python del Capítulo 4

Instrucción	Descripción
`randint`	Genera números aleatorios enteros.
`range`	Genera una lista.
`continue`	Se usa para saltar instrucciones de un ciclo.
`brake`	Se usa para interrumpir un ciclo.
`for`	Se usa para realizar ciclos definidos.
`while`	Se usa para realizar ciclos indefinidos.
`random`	Biblioteca con funciones que generan números aleatorios.

4.9 Conclusiones

En este capítulo hemos cubierto los fundamentos de ciclos. Los ciclos estudiados en este capítulo son los ciclos **Para** y **Mientras**. Así mismo hemos cubierto su realización en lenguaje Python. A través de ejemplos el lector aprende los detalles de cada uno de estos ciclos. También se estudia la manera de anidar los distintos tipos de ciclos para poder realizar tareas más complicadas.

4.10 Ejercicios

1. ¿Qué se produce con el siguiente ciclo?

```
n = 7
while n > 0:
    print(n)
print("Trabajo terminado.")
```

   a) Es un ciclo infinito.
   b) No deben escribirse dos puntos después de la condición del `while`.
   c) La instrucción `print` debe estar con una sangría de dos espacios.
   d) `while` no es palabra reservada en Python.

3. ¿Qué se despliega con el siguiente programa en Python?

```
total = 0
for i in [5, 4, 3, 2, 1] :
    total = total + 1
print (total)
```

   a) 5
   b) 1.00
   c) 10
   d) 15

4. ¿Cuál es la variable de la iteración en el siguiente código en Python?

```
amigos = ["Hugo", "Paco", "Luis"]
for amigo in amigos:
    print("Hola amigo ", amigo)
print("Ya terminé.")
```

   a) amigo
   b) amigos
   c) Hugo
   d) Paco
   e) Luis

5. ¿Qué imprime el siguiente código?

```
lo_mas_chico = -1
for numero in [9, 12, 3, 9, 17, 23] :
    if numero < lo_mas_chico:
        lo_mas_chico = numero
print (lo_mas_chico)
```

   a) -1
   b) 3
   c) 23
   d) 9

6. ¿Cuántas veces se ejecuta el siguiente ciclo?

```
n = 0
while n > 0 :
    print (n)
    n = n + 1
print (" Terminado ")
```

7. ¿Qué resultado despliega el siguiente código?

```
for num in range(0, 20, 2):
    print( num )
```

8. ¿Qué resultado produce el siguiente código en Python?

```
for num in range(20, 0, -1):
    if (valor%4 != 0):
        print(num,"\n")
```

9. Convertir el lazo for del ejercicio 7 a un lazo Mientras usando `while`.

10. Escriba en pseudocódigo un algoritmo que indique si un número que se recibe es impar.

11. Escriba un algoritmo que reciba dos números `a` y `b`. Si el segundo número es menor que el primero se escribe un mensaje indicando esto. Si el segundo número es mayor que el primero escriba la suma de los enteros en ese rango.

12. Escriba un algoritmo para que escriba el siguiente arreglo de asteriscos:

```
*******************
******************
*****************
****************
***************
**************
*****
***
*
```

Figura 4.2 Figura del ejercicio 4.12.

13. Escriba un algoritmo que genere 20 números aleatorios y que cheque cuántos de ellos son iguales al primer número aleatorio generado. En el ciclo use la instrucción Para. Convierta el algoritmo a lenguaje Python.

14. Repita el ejercicio 13 para saber cuantos números son iguales a la suma de los dos primeros números aleatorios generados.

15. Repita el ejercicio 13 pero use un ciclo Mientras con `while`.

16. Repita el ejercicio 14 usando un ciclo Para con `for`.

17. Escriba un algoritmo en Python que lea 10 nombres de personas y que termine cuando un nombre se repita. Use la comparación == para comparar las cadenas.

18. Haga un cálculo de la función

$$f(x) = x^2 - 1$$

en el rango de x de 1 a 2. Calcule para 20 puntos, es decir, el incremento dx debe ser

$$dx = (2 - 1)/20$$

19. Usar un algoritmo recursivo para calcular el valor de la constante $e = 2.71$ usando la ecuación recursiva

$$x_n = \left(1 + \frac{1}{n}\right)^n$$

Para checar que tan rápido converge imprima resultados en la 5ta, 10ma y 15ava iteración.

Capítulo 5
Cadenas, Listas, Diccionarios y Tuplas

5.1 Introducción

5.2 Cadenas

5.3 Listas

5.4 Definición de listas

5.5 Tuplas

5.6 Diccionarios

5.7 Instrucciones de Python del Capítulo 5

5.8 Conclusiones

5.9 Ejercicios

Lo importante es no dejar de cuestionarse.

Albert Einstein

Objetivos

Los datos se agrupan en conjuntos con distintas estructuras llamadas estructuras de datos. Las estructuras de datos más sencillas son las cadenas, las listas y los diccionarios.

5.1 Introducción

En el capítulo 2 vimos una breve introducción a las cadenas (strings), a las listas y a los diccionarios. En este capítulo vemos algunas funciones que se pueden realizar con estas estructuras de datos de Python. Ejemplos demostrativos nos ilustran cómo se pueden usar este tipo de estructuras de datos. Adicionalmente, introducimos el concepto de tuplas.

5.2 Cadenas

Las cadenas son elementos alfanuméricos que se encierran entre comillas dobles o sencillas. Por ejemplo:

```
a = "perro"      carro = 'Torino'      animal = 'caballo_23'
```

En cada uno de estos ejemplos, las cadenas tienen un nombre y cada una de las cadenas tiene un cierto número de caracteres. Así vemos que `a` tiene 5 caracteres, `carro` está formado por 6 caracteres y la cadena `animal` tiene un total de 10 caracteres.

Cada caracter ocupa una posición en la cadena. El caracter de la izquierda ocupa la posición 0 y continúan en orden ascendente para las posiciones a la derecha. De esta manera para la cadena `Torino`, el caracter 'T' ocupa la posición 0, el caracter 'o' tiene la posición 1, y así hasta llegar al último caracter que tiene la posición 5. Como

podemos ver la posición de un caracter es un número entero. Este número entero recibe el nombre de índice.

El **índice** indica la posición que tiene un caracter de la cadena.

Existe una manera alterna para numerar el índice de una cadena si consideramos el primer elemento el de la derecha. En este caso el valor del índice es -1. Los índices se decrementan conforme nos movemos hacia la izquierda. Para la cadena "`caballo_23`", el caracter '`3`' tiene índice -1, el caracter '`2`' tiene índice -2, y así sucesivamente hasta llegar al caracter '`c`' que tiene índice -10.

En ocasiones es necesario seleccionar un caracter de la cadena. Esto lo podemos hacer simplemente con el nombre de la cadena y el índice entre corchetes. Por ejemplo, para la cadena `a` tenemos que `a[3]` = '`r`' y `a[-4]` = '`e`'.

También es de interés una **porción** de una cadena. En este caso para seleccionar los primeros `k` caracteres de una cadena usamos:

```
cadena[ 0 : k ]
```

donde se obtiene los caracteres de la posición 0 a la posición `k-1`. Supongamos que se tiene la cadena `fruta` = "`manzana_roja`". Para seleccionar la porción que corresponde a "`manzana`" usamos:

```
fruta[0 : 7] = "manzana"
```

para un total de 7 caracteres, del 0 al 6. Alternativamente podemos usar:

```
fruta[-12 : -5] = "manzana"
```

A una parte de una cadena se le llama **porción** (**slice**).

x[k : n+1] selecciona los elementos del **k**-ésimo al **n**-ésimo.

Ejemplo 5.1

Porciones de cadenas

Consideremos la cadena

```
estudiante = "Pedro_Sánchez_Cantarranas"
```

para seleccionar las iniciales del nombre y apellidos usamos:

```
>>> estudiante[0]
'P'
>>> estudiante[6]
'S'
>>> estudiante[14]
'C'
```

Para seleccionar las porciones correspondientes al nombre y los apellidos, podemos usar:

```
>>> estudiante[0 : 5]
'Pedro'
>>> estudiante[6 : 13]
'Sánchez'
>>> estudiante[14:35]
'Cantarranas'
```

Notamos que aunque no existen 35 caracteres en el nombre, Python nos da solamente hasta el último existente. El mismo resultado se obtiene con:

```
>>> estudiante[14:25]
'Cantarranas'
```

Otra operación de interés es tomar una porción de la cadena de algún índice hasta el final y otra es desde el principio hasta un índice k. Esto lo hacemos con:

`x[ k : ]`	Porción desde el índice k hasta el final de la cadena.
`x[ : k+1 ]`	Porción desde el inicio de la cadena hasta el índice k.

5.2.1 Longitud de una cadena

Para calcular la longitud, o el número de caracteres de una cadena, usamos la instrucción `len`.

> La instrucción `len` calcula la longitud de una cadena.

Ejemplo 5.2

Longitud de una cadena

Se desea calcular la longitud de la cadena "`El_caballo_blanco_de_Napoleón`". Esto lo podemos hacer con:

```
>>> len( "El_caballo_blanco_de_Napoleón")
29
```

5.2.2 Separación de una cadena

En ocasiones necesitamos separar una cadena que consiste de varias palabras. Para esto usamos la instrucción `split`. El formato es:

```
cadena.split( subcadena )
```

donde subcadena es una porción de la cadena. Las cadenas resultantes se almacenan

en una lista. Si dejamos en blanco el lugar de la subcadena, la cadena se separa en los espacios entre las palabras de la cadena. Pero si escribimos una subcadena, la cadena se separa en porciones separadas por la subcadena.

Ejemplo 5.3

Separación de una cadena

Consideremos la frase de Carl Sagan almacenada en la cadena `Sagan`:

```
Sagan ="Vivimos en una sociedad profundamente dependiente de la ciencia
y la tecnología y en la que nadie sabe nada de estos temas. Ello constituye
una fórmula segura para el desastre."
```

Para separarla en palabras usamos como separador el espacio " " lo que es equivalente a no poner nada dentro del paréntesis:

```
>>> Sagan.split( )
```

```
['Vivimos','en','una','sociedad','profundamente','dependiente','de','la','ciencia',
'y', 'la', 'tecnología', 'y', 'en', 'la', 'que', 'nadie', 'sabe', 'nada', 'de', 'estos',
'temas.', 'Ello', 'constituye', 'una', 'fórmula', 'segura', 'para', 'el', 'desastre.'
]
```

Por otro lado si separamos con la sílaba `te` tenemos:

```
>>> Sagan.split( 'te' )
```

```
[ 'Vivimos en una sociedad profundamen',' dependien',' de la ciencia y la
', 'cnología y en la que nadie sabe nada de estos ', 'mas. Ello constituye
una fórmula segura para el desastre.' ]
```

Notamos que ahora se ha separado la frase con el separador `te`. En el resultado no aparece el separador `te`.

5.2.3 Operaciones con cadenas

Solamente tenemos dos operaciones con las cadenas, las cuales son: suma o concatenación y multiplicación por una constante.

5.2.4 Concatenación o suma de cadenas

El resultado de concatenar o sumar dos o más cadenas es otra cadena. La concatenación o suma se puede realizar de la siguiente manera:

Usando el operador + entre las cadenas.

Mostramos con un ejemplo.

Ejemplo 5.4

Concatenación de cadenas

Supongamos que tenemos las cadenas:

```
a = "El perro"      b = "es blanco"      c = "y negro"
```

si ahora efectuamos:

```
>>> nueva = a + b
"El perroes blanco"
```

Notamos que no existe un espacio entre las dos cadenas ya que solamente se concatenaron. Si deseamos añadir un espacio entre las cadenas " " hacemos lo siguiente:

```
>>> nueva = a + " " + b
"El perro es blanco"
```

Ahora concatenamos las tres cadenas y añadimos espacios entre las cadenas y un punto al final de las tres cadenas:

```
>>> completa = a + " " + b + " " + c + "."
"El perro es blanco y negro."
```

5.2.5 Multiplicación de cadenas

La multiplicación de una cadena por un número entero hace que la cadena se repita tantas veces como lo indique el entero.

Ejemplo 5.5

Multiplicación de cadenas

Supongamos que tenemos la cadena:

```
factor = "México"
```

si ahora efectuamos:

```
>>> pais = 3*factor
"MéxicoMéxicoMéxico"
```

Es importante recalcar que el número debe ser ENTERO.

5.2.6 Inmutabilidad de las cadenas

Las cadenas son inmutables. Esto quiere decir que una vez que se crea una cadena, ninguno de los caracteres de la cadena se puede cambiar.

Las cadenas son inmutables.

Ejemplo 5.6

Inmutabilidad de cadenas

Consideremos la cadena, escrita con P minúscula:

```
ciudad = 'puebla'
```

Si queremos cambiar la p minúscula por P mayúscula, podemos intentar cambiar:

```
>>> ciudad[0] = 'P'
Traceback (most recent call last):
File "<pyshell#60>", line 1, in <module>
>>> ciudad[0] = 'P'
TypeError: 'str' object does not support item assignment
```

El mensaje de error se debe a que no se permite cambiar ningún caracter de la cadena, ya que una cadena es **inmutable**. Lo que podemos hacer es crear una nueva cadena de la siguiente manera:

```
>>> ciudad2 = 'P' + ciudad[1:6]
'Puebla'
```

Es de hacer notar que la nueva lista podría tener el nombre de la cadena anterior `ciudad`:

```
>>> ciudad = 'P' + ciudad[1:6]
'Puebla'
```

5.2.7 Otras operaciones con cadenas

Existen otras operaciones que se pueden realizar con cadenas. La tabla siguiente presenta una lista con los métodos que se pueden usar con cadenas. La forma de usarlas es `cadena.funcion( )`.

	Operación	Descripción
1	`capitalize( )`	Convierte el primer caracter a mayúscula.
2	`center(N, 'x')`	Pone `x` a ambos lados de la cadena hasta acompletar `N` caracteres. La cadena original queda en el centro.
3	`count(x)`	Cuenta cuantas veces aparece el caracter `x`.
4	`upper( )`	Convierte minúsculas a mayúsculas.
5	`swapcase( )`	Invierte mayúsculas a minúsculas y viceversa.
6	`endswith(suf)`	Regresa `True` si la cadena termina con el sufijo "`suf`".
7	`expandtabs(tab = n)`	Cambia tabuladores por `n` espacios.
8	`find(subcadena)`	Determina si la subcadena es parte de la cadena.
9	`title( )`	Cambia a mayúscula el primer elemento.
10	`isalnum( )`	Regresa `True` si los caracteres son alfanuméricos.
11	`isalpha( )`	Regresa `True` si todos los caracteres son alfanuméricos.
12	`isdigit( )`	Regresa `True` si todos los caracteres son dígitos.
13	`islower( )`	Regresa `True` si tiene sólo letras minúsculas.
14	`isnumeric( )`	Regresa `True` si tiene sólo números.
15	`isspace( )`	Regresa `True` si tiene sólo espacios.
16	`istitle( )`	Regresa `True` si empieza con mayúscula.
17	`isupper( )`	Regresa `True` si solamente tiene mayúsculas.
18	`lower( )`	Convierte mayúsculas a minúsculas.
19	`max(cadena)`	Obtiene el caracter de mayor valor.
20	`min(cadena)`	Obtiene el caracter de menor valor.
21	`replace("a", "b")`	Reemplaza el caracter "`a`" con "`b`".
22	`rstrip( )`	Remueve los espacios de la derecha.
23	`lstrip( )`	Remueve los espacios de la izquierda.
24	`strip( )`	Realiza `lstrip()` y `rstrip()` .
25	`zfill (N)`	Añade ceros al inicio para tener `N` caracteres.

Ejemplo 5.7

Operaciones con cadenas

Supongamos que tenemos las cadenas :

```
a = "animales"

b = "flores"

c = "naciones"
```

Se pueden realizar las siguientes operaciones:

```
# Cambia a mayúscula la primera letra:
>>> a.capitalize()
'Animales'


# Rodea la cadena con 'p' hasta acompletar 20 caracteres:
>>> a.center(20,'p')
'pppppppanimalesppppppp'

# Cuenta cuantas veces aparece 'n' entre los
# caracteres de las posiciones 0 y 6:
>>> c.count('n', 0,6)
2

# Regresa True si la cadena termina con "es":
>>> a.endswith('es')
True

# Regresa True si sólo tiene caracteres alfanuméricos:
>>> a.isalnum( )
True
```

```
# Regresa True sólo si tiene caracteres numéricos:
>>> a.isnum( )
False

# Regresa el caracter de valor numérico menor:
>>> min(c)
'a'

# Regresa el caracter de valor numérico mayor:
>>> max(c)
's'

# Reemplaza el caracter "s" con el caracter "S":
>>> c.replace("s" ,"S")
'nacioneS'
```

Añade ceros al principio de la cadena hasta acompletar longitud 13.

```
>>> c.zfill(13)
'00000naciones'
```

5.2.8 Condiciones

Una manera de realizar condiciones usando cadenas es checando si un caracter forma parte de la cadena. Esto lo hacemos simplemente con el operador `in`. Si tenemos la cadena `a = "Universidad"`, podemos realizar:

```
>>> "u" in a
False
```

El resultado es `False` porque el caracter `u` no está en la cadena `a`. Sin embargo, el caracter `U` nos proporciona:

```
>>> "U" in a
True
```

Ejemplo 5.8

Condición sobre una cadena

Consideremos el siguiente problema: Se desea escribir un mensaje que sea "`Que gane el mejor`" cada vez que el caracter "`o`" está en la palabra dada y que escriba "`No hay juego`" si la letra no se encuentra en la cadena. Si la cadena es "futbolista", tenemos que:

```
if "o" in "futbolista":
     print( "Que gane el mejor." )
else:
     print( "No hay juego." )
```

El resultado es:

```
Que gane el mejor.
```

Ya que el caracter "`o`" sí está en la cadena "`futbolista`".

5.2.9 Ciclos

Los ciclos se pueden realizar con los caracteres de una cadena. En este caso el ciclo se repite tantas veces como caracteres haya en la cadena. De esta manera si la cadena es "`todas`", entonces el ciclo se ejecuta una vez para el caracter `t`, una segunda vez para el caracter `o`, una tercera vez para el caracter `d`, una cuarta vez para el caracter `a`, y finalmente una quinta vez para el caracter `s`.

También se pueden combinar condiciones sobre cadenas con ciclos. Por ejemplo, si se desea mostrar un mensaje cuando un caracter se encuentra en la cadena. Esto lo podemos hacerlo de la siguiente manera:

```
for k in "angeles": # Inicia un ciclo con los caracteres de la cadena
     if m in "angeles"
          print("La letra m SÍ está en la palabra angeles")
     else:
          print("La letra m NO está en la palabra angeles")
```

Ejemplo 5.9

Ciclo y condición con una cadena

Se desea saber cuantas veces en la palabra "`colombiano`" se encuentra el caracter "`o`". Además, deseamos que se despliegue el mensaje "`Sudamericano`' en cada ocasión que se encuentre el caracter deseado. Lo hacemos de la siguiente manera:

1. Primero inicializamos un contador.

2. Usamos un ciclo `for` para recorrer todos los caracteres de la cadena.

3. Dentro del ciclo usamos una condición para acumular el número de veces que el caracter se encuentra en la cadena.

4. Imprimimos la palabra '`Ganador`' cada vez que se encuentre el caracter '`o`'.

5. Cada vez que se cumpla la condición se incrementa el contador.

6. Al terminar el ciclo se muestra el resultado.

El algoritmo en pseudocódigo y en Python es:

Pseudocódigo	Python
Algoritmo: Buscador de caracteres. "'Busca cuántas veces el caracter o se encuentra en la palabra colombiano"'	"'Buscador de caracteres. Busca cuántas veces el caracter o se encuentra en la palabra colombiano"'
INICIO **Variables:** **Entero:** suma, k	# **INICIO**

suma ← 0
Para k **en** "colombiano"
  Si "o" **en** "colombiano":
  suma = suma +1
  Mostrar: Ganador
  FinSi
 	
  FinPara
Mostrar:
'Se encuentra", suma, "veces"
	suma = 0
	for k in "colombiano":
	  if "o"in "colombiano":
	  suma = suma + 1
	  print("Ganador")
	# **Fin de la condición**
	
	# **Fin del ciclo for**
	print ("Se encuentra", end = " ")
	print ("%5.3f veces " suma)

FIN	# **FIN**

El resultado que nos produce el algoritmo es:

```
Se encuentra 3 veces
```

5.3 Listas

Hasta ahora los datos se han manejado a través de variables simples, es decir, a variables tales como:

Factor ← 50.3	real
Edad ← 20	entero
Planilla ← 'p'	alfanumérica
es_primo ← verdadero	lógica

Cuando decimos simples, nos referimos a variables donde solamente hay un valor en ellas. Sin embargo, en ocasiones es útil organizar los datos en colecciones de datos que agrupen distintas variables, o distintos valores de una variable. Existen distintos tipos de conjuntos de datos mejor conocidos como **estructuras de datos**. Las estrucuras de datos que se describen en las secciones restantes de este capítulo son listas, tuplas y diccionarios.

5.4 Definición de listas

Las listas son colecciones de datos los cuales pueden ser de cualquier tipo. Las listas se agrupan dentro de corchetes. Los elementos de una lista se separan con comas. Ya en el capítulo 2 dimos una breve instroducción a listas. Ejemplos de listas son:

```
a = [ 23.56, -71, -68.1 ]    b = ["coche", "bici", 'patín', 14 ]
```

La lista a está compuesta de dos números reales y un número entero, mientras que la lista b consiste de tres cadenas y un entero.

Si una lista está formada por un elemento es necesario escribir una coma después del elemento. Si no se hace así entonces no se crea la lista. Por ejemplo:

`a = [ 1.2, ]` SÍ es una lista.

`b = [ 1.2 ]` NO es una lista.

A cada elemento de una lista le asociamos una posición como fue el caso de las cadenas. A esta posición se le llama ÍNDICE. De esta manera el primer elemento de izquierda a derecha es el elemento de la posición cero, el que sigue hacia la derecha ocupa la posición uno y así sucesivamente. También se pueden enumerar de derecha a izquierda. En este caso la posición de la derecha es la posición -1, la que sigue a la izquierda es la posición -2, y así sucesivamente hasta terminar.

5.4.1 Operaciones con listas

Las operaciones que se pueden hacer con listas son las mismas que se hacen con cadenas.

Ejemplo 5.10

Concatenación o suma y multiplicación de listas

Supongamos que tenemos las listas:

```
animales = [ "perros", "gatos", "ratones"]

flores = [ "nardos", 3.57, "rosas"]
```

La concatenación de estas listas es:

```
>>> suma = animales + flores
[ "perros", "gatos", "ratones", "nardos", 3.57, "rosas"]
```

Si ahora efectuamos:

```
>>> resultado = 2*flores
[ "nardos", 3.57, "rosas", "nardos", 3.57, "rosas"]
```

Es importante que el número por el que se multiplica la cadena sea un número ENTERO. Al tratar de multiplicar por un número real se marca un error.

Ejemplo 5.11

Longitud de una lista

Supongamos que tenemos la lista:

```
paises = [ "México", "Francia", "Japón", "Australia"]
```

Entonces la longitud de la lista paises es:

```
>>> len(paises)
4
```

Estos es porque la lista está compuesta de tres cadenas, pero tenemos que:

```
>>> len(paises[0])
6
```

ya que ahora obtenemos la longitud de la cadena que es el elemento 0 de la lista `paises` y esto es porque `paises[0]` = "`México`" es una cadena de longitud 6. Fin

Ejemplo 5.12

Porciones de listas

Supongamos que tenemos la lista:

```
numeros = [ 23, 45, 67, 89.4, -31 ]
```

si ahora queremos solamente una sublista o *slice* de la lista con los elementos del segundo elemento (índice 1) hasta el cuarto (índice 3), lo hacemos de la siguiente manera:

```
>>> numeros[ 1:4 ]
[ 45, 67, 89.4 ]
```

Vemos que el último elemento no se incluye en el resultado. Si lo quisiéramos incluir usamos:

```
>>> numeros[ 1:5 ]
```

5.4.2 Mutabilidad

Las listas no son inmutables, lo que quiere decir que podemos cambiar uno o más elementos de la lista.

Ejemplo 5.13

Mutabilidad de las listas

Supongamos que tenemos la lista:

```
calificaciones = [ 8.3, 9.45, 10, 9.9, 7.8 ]
```

Si ahora queremos solamente cambiar la calificación de 7.8 a 8.5, lo podemos hacer con:

```
>>> calificaciones[4] = 8.5
>>> print(calificaciones)
[8.3, 9.45, 10, 9.9, 8.5]
```

Vemos que el elemento deseado cambió su valor.

5.4.3 Conversión de lista de cadenas a cadena

Es posible convertir una lista de cadenas a una sola cadena usando la instrucción `join`. Se debe especificar la subcadena con que se van a separar las cadenas. El formato es:

```
subcadena.join( [ lista de cadenas ] )
```

Mostramos el uso de `join` con un ejemplo.

Ejemplo 5.14

Concatenación o suma y multiplicación de listas

Supongamos que tenemos la lista y la subcadena:

```
lista = [ "perros", "gatos", "ratones"]
```

```
subcadena = [ " y "] # Hay un espacio antes y despues de la y.
```

Entonces, unimos los elementos de la lista con:

```
>>> subcadena.join(lista)
```

Para obtener:

```
'perros y gatos y ratones'
```

Si ahora la subcadena esta vacía y solamente consiste de las comillas:

```
subcadena = ""
```

Obtenemos:

```
>>> subcadena.join(lista)
'perrosgatosratones'
```

5.4.4 Otras operaciones con listas

Operaciones adicionales con listas se encuentran en la siguiente tabla:

1	`len(lista)`	Longitud de la lista.
2	`max(lista)`	Regresa el elemento con el valor máximo.
3	`min(lista)`	Regresa el elemento con el valor mínimo.
4	`tuple(lista)`	Convierte una lista a tupla.
5	`lista.append(obj)`	Adiciona objetos a la lista.
6	`lista.count(obj)`	Regresa cuantas veces `obj` está en la lista.

7	`lista1.extend(lista2)`	Adiciona `lista2` al final de `lista1`.
8	`lista.index(objeto)`	Regresa el índice del primer elemento que es igual a `objeto`.
9	`lista.insert(indice, obj)`	Inserta `obj` en la posición `indice`.
10	`lista.pop(indice)`	Regresa y remueve de la lista el elemento. Si no se da un índice, regresa el último elemento.
11	`lista.remove(objeto)`	Remueve `objeto` de la lista. Si no existe se marca un error.
12	`lista.reverse( )`	Invierte el orden de los objetos de la lista.
13	`lista.sort( )`	Ordena los objetos de la lista.

Ejemplo 5.15

Ejemplos con listas

Consideremos que tenemos las siguientes listas:

```
a = [ "Pedro", 555, "Python", "Intel", "HP"]

b = [45, -67.8, 999.99, -2702.2, 1858]

c = ["mouse", "teclado"]
```

Podemos realizar las siguientes operaciones:

```
# Calcular la longitud de la lista:
>>> len(a)
5

# Obtener el máximo de una lista
>>> max(b)
1858
```

```
# Obtener el mínimo de una lista
>>> min(b)
-2702.2

# Añadir un elemento a una lista:
>>> a.append(-2015)
>>> a
['Pedro', 555, 'Python', 'Intel', 'HP', -2015]

# Extender la lista a con la lista c:
>>> a.extend(c)
>>> a
['Pedro', 555, 'Python', 'Intel', 'HP', -2015, 'mouse', 'teclado']

# Insertar 47 en la posición 2 de la lista b:
>>> b.insert(2, 47)
>>> b
b = [45, -67.8, 47, 999.99, -2702.2, 1858]

# Regresa y borra el último elemento de la lista b:
>>> b.pop()
1858

# Regresa y borra el elemento en la posición 3 de la lista b:
>>> b.pop(3)
999.99

# Borra el último elemento de la lista b:
>>> b.remove()

# Borra el elemento en la posición 3 de la lista b:
>>> b.remove(3)

# Invierte el orden de los elementos de una lista:
>>> a.reverse()
>>> a
['teclado', 'mouse', -2015, 'HP', 'Intel', 'Python', 555, 'Pedro']
```

```
# Ordena los elementos de una lista:
>>> b.sort()
>>> b
[-2702.2, -67.8, 45, 999.99, 1858]
```

Las listas alfanuméricas SÍ se pueden ordenar, pero en el caso de que la lista contenga tanto elementos alfanuméricos como numéricos NO se puede usar `sort`.

5.5 Tuplas

Una tupla es una estructura de datos conformada por elementos los cuales pueden ser de distinto tipo. Las tuplas son INMUTABLES. Las tuplas se encierran entre paréntesis y sus elementos se separan por comas. Ejemplos de tuplas son:

```
campeones = ( "Alemania", "Francia", "Brasil", "Italia" )

cantidades = ( 1, 233, 1.37e4, -456 )
```

En inglés, la palabra tupla es *tuple*.

5.5.1 Intercambio de valores

En el capítulo 2 vimos como intercambiar valores de variables. Es decir, si tenemos dos variables `a` y `b` y deseamos que los valores ellas se intercambien, ahí usamos una tercera variable para almacenar temporalmente el valor de una de ellas. Usando tuplas es mucho más fácil. Simplemente necesitamos escribir:

```
>>> (a, b) = (b, a)
```

Por ejemplo, si tenemos:

```
>>> a = 5; b = 100
>>> (a, b) = (b, a)
```

```
>>> print( a )
100
>>> print( b )
5
```

5.5.2 Operaciones con tuplas

Algunas de las operaciones sobre tuplas son las siguientes:

	Función	Descripción
1	`tuple(lista)`	Convierte una lista a tupla.
2	`len(tupla)`	Longitud de una tupla.
3	`max(tupla)`	Regresa el valor máximo de la tupla.
4	`min(tupla)`	Regresa el valor mínimo de la tupla.

Ejemplo 5.16

Ejemplos con tuplas

Consideremos que tenemos una lista y una tupla dadas por:

```
a = [ 3, 5, 6, -3, -9.45]        b =( -35, 4.67, 27.41, 98.65)
```

Podemos realizar las siguientes operaciones:

```
# Convertir tupla a lista
>>> c = list(b)
c = [ 8.3, 9.45, 10, 9.9, 7.8 ]

# Calcular la longitud de una tupla
>>>  len(b)
4

# Obtener el máximo de una tupla
>>> max(b)
```

```
98.65

# Obtener el mínimo de una tupla
>>> min(b)
-35
```

5.6 Diccionarios

Los diccionarios son estructuras de datos consistentes en **listas de pares** de variables. Cada par tiene un elemento llamado `clave` (`key`) que puede ser de cualquier tipo y otro elemento llamado `valor` que también puede ser de cualquier tipo. Los diccionarios se delimitan por llaves. El formato de un diccionario es:

```
a = { "clave1":valor1,"clave2":valor2,...,"claveM":valorM }
```

Un ejemplo de diccionario es:

```
calificaciones = { "Hugo": 9,'Sara': 9.2,"Beto": 7.8,"Xavier":10}
```

Los diccionarios no son inmutables, de tal manera que podemos cambiar los elementos de un diccionario.

Dado que en un diccionario, los valores están referenciados por las claves, puede darse el caso de que Python lo escriba en otro orden. Por ejemplo, si hacemos:

```
>>> print(calificaciones)
{ 'Beto' : 7.8, 'Xavier' : 10, 'Hugo' : 9, 'Sara' : 9.2 }
```

Vemos que Python ha cambiado el orden en que lo dimos pero no las parejas de clave y valor.

Para seleccionar un valor es solamente necesario dar la clave, como en:

```
>>> calificaciones[ 'Beto' ]
7.8
```

Dado que los diccionarios no son inmutables es posible cambiar una calificación. Por ejemplo, si se cambia la calificación de `Beto` de `7.8` a `8.9`, hacemos:

```
>>> calificaciones['Beto'] = 8.9
```

Con este cambio podemos ver lo siguiente:

```
>>> print(calificaciones)
{'Beto' : 8.9, 'Xavier' : 10, 'Hugo' : 9, 'Sara' : 9.2 }
```

donde vemos que `Beto` tiene su calificación cambiada al valor deseado.

Las operaciones suma y multiplicación que se pueden realizar para las cadenas, listas y tuplas también se pueden efectuar para los diccionarios.

5.6.1 Otras operaciones para diccionarios

Las operaciones que se pueden realizar con los diccionarios se muestran en la tabla siguiente:

	Función	Descripción
1	`str(diccionario)`	Convierte el `diccionario` en cadena.
2	`type(variable)`	Regresa el tipo de la variable.
3	`dicc.clear( )`	Remueve todos los elementos del diccionario.
4	`dicc.copy()`	Copia el diccionario en otra variable.
5	`dicc.fromkeys(lista,valor)`	Crea un nuevo diccionario. Las claves son de lista con valor.
6	`dicc.get(clave)`	Regresa el valor de `clave`.
7	`dicc.has_key(clave)`	Regresa `True` si la clave está en dicc.
8	`dicc.items()`	Regresa una lista de pares (clave, valor).
9	`dicc.keys()`	Regresa una lista con las claves (keys) del diccionario.
10	`dicc1.update(dicc2)`	Añade los pares clave-valor de dicc2 al dicc1.
11	`dicc.values( )`	Regresa una lista con los valores.

Ilustramos el uso con ejemplos.

Ejemplo 5.17

Ejemplos usando diccionarios

Consideremos el diccionario siguiente:

```
cosas = { "M" : "manzana", "P" : "platano", "L" : "limón" }
```

Podemos usar las siguientes instrucciones:

```
# Convertir diccionario a cadena:
>>> cadena = str(cosas)
cadena = " { 'M' : 'manzana', 'L' : 'limón', 'P' : 'plátano' } "

# Regresa el tipo:
>>>  type(cosas)
< class 'dict' >

# Copia el diccionario en otra variable
>>> frutas = cosas.copy( )
frutas = { "M" : "manzana", "P" : "plátano", "L" : "limón" }

# Crea un nuevo diccionario a partir de una lista:
>>> lista = [1, 2, 3, 6, 9]
>>> tres = cosas.fromkeys( 'lista' : '2' )
{'1' : 'a', '2' : 'a', '3' : 'a', '5' : 'a'}

# Regresa una lista de pares:
>>> cosas.items( 'U': 'uvas')
dict_items ( [ ( 'M', 'manzana'), ('L', 'limón'), ( 'P', 'plátano' ) ] )

# Regresa una lista con los valores:
>>> tres = cosas.values( )
dict_values( [ 'manzana', 'limón', "plátano" ])
```

5.7 Instrucciones de Python del Capítulo 5

Instrucción	Descripción
`x[n: m + 1]`	Selecciona caracteres de la cadena `x` del `n` al `m`.
`x[ : m + 1]`	Selecciona caracteres de la cadena `x` del inicio al `m`.
`x[n : ]`	Selecciona caracteres de la cadena `x` del `n` al final.
`' '`	Delimitador de cadena.
`len(x)`	Número de caracteres de la cadena `x`.
`split`	Separador de cadena.
`+`	Concatena cadenas.
`*`	Multiplica una cadena por un entero.
`join`	Une cadenas para formar una lista de cadenas.

5.8 Conclusiones

En esta sección se definieron las cadenas, listas, tuplas y diccionarios, así como algunas de las operaciones disponibles para trabajar con estas estructuras de datos. Los ejemplos presentados muestran las distintas formas de realizar operaciones con ellas. En los capítulos posteriores del libro veremos cómo se usan en el diseño de algoritmos y su realización en Python. Los ejercicios al final del capítulo proporcionan un complemento en el uso de las estructuras de datos presentadas en el capítulo.

5.9 Ejercicios

1. Concatene la cadena "Continente" con la cadena "Americano" para formar la cadena: "Continente Americano". Debe haber un espacio entre las dos palabras.

2. Realice un programa que implemente el juego de los números donde el usuario va a escoger un número entero generado de manera aleatoria. El juego debe ir de la siguiente manera:

Hola, ¿cómo te llamas?

Pedro

Muy bien Pedro, Adivina el número entre 1 y 20.

5

Tu número es muy bajo.

Dame otro número.

12

Tu número es muy bajo.

Dame otro número.

16

Buen trabajo Pedro, acertaste en tres intentos.

3. Escriba un algoritmo que lea una frase y calcule la frecuencia de cada una de las letras contando cuántas veces se encuentra cada letra en la frase.

4. En el ejercicio anterior ordene las letras de mayor a menor frecuencia.

5. Generar una cadena consistente en corchetes ordenados de manera aleatoria. Escriba un programa que genera una cadena de corchetes de longitud 10 y que cheque cuantos pares de corchetes son correctos. Ejemplos son:

`[]`	correcto
`]]`	incorrecto
`[[]]`	correcto
`[][]`	correcto
`[[][]]`	correcto
`[[]]]]`	incorrecto

6. Un hápax es una palabra que aparece una sola vez en un documento. Escriba un programa que calcule cuántos hápax hay en un texto que se encuentra almacenado en un archivo.

7. Un acrónimo es una palabra formada con las iniciales de una frase. Por ejemplo, ROM es el acrónimo de Read Only Memory. Diseñe un algoritmo que lea una frase y despliegue el acrónimo. Las letras del acrónimo deben ser mayúsculas.

8. El valor de una palabra es la suma de los valores asignados a una letra. Si a = 1, b = 2, c = 3, etc., diseñe un algoritmo y escriba el correspondiente programa para calcular el valor de una palabra recibida. Use un diccionario para asignar los valores a cada letra de la siguiente manera:

```
valores = { "a": 1, "b": 2, ..., "z" :26 }
```

9. Diga que se obtiene con el siguiente ciclo (la frase se atribuye a Galileo):

   texto = "Las matemáticas son el alfabeto con el cual Dios ha escrito el Universo."

```
for k in texto.split():
    print(k)
```

10. Repita el ejercicio anterior con:

```
for k in texto.split(' t'):
    print(k)
```

11. Sume las cadenas "23" y "58" y convierta el resultado a flotante.

12. Dada la cadena a = "`El perro se llama Zeus`", obtenga:

    a) La longitud de la cadena.

    b) La cadena que corresponde a la palabra "`perro`".

    c) Qué porción de la cadena corresponde a `a[ 17: ]`

13. Escriba un programa que lea 4 palabras, calcule la longitud de cada una de ellas y las despliegue junto con su longitud indicada con asteriscos. Por ejemplo:

```
memoria     *******
mouse       *****
computadora ***********
dos         ***
```

14. Repita el ejercicio anterior pero ahora los asteríscos deben estar alineados verticalmente y abajo debe ir el número de caracteres en cada palabra. Por ejemplo, para la frase: "Programación con Python" el resultado es:

```
*
*
*
*
*
*
*     *
*     *
*     *
*     *
*  *  *
*  *  *
*  *  *
12 3  6
```

15. Escriba un programa que calcule la longitud promedio de las palabras en un texto. El texto debe leerse de un archivo en formato .txt.

16. Convierta la cadena "Computadora" a lista y a tupla.

17. Convierta la lista `[ 2, 4, 6, 8,` "a"] a cadena. El resultado debe ser '`2468a`'

18. Convierta la cadena "Continente" a lista y a tupla.

19. Diseñe un algoritmo que implemento el código Cesariano consistente en codificar un texto. La estrategia de este código consiste en cambiar una letra de la frase por otra cuya posición en el alfabeto es adelante un número determinado de veces. Por ejemplo, si se usan dos posiciones de corrimiento, entonces la a se cambia por d, la b por e, la c por f, y así hasta el final del alfabeto donde la x se cambia por a, la y por b y la z por c. El texto se debe leer de un archivo.

20. Escriba un programa que convierta una frase en español a su equivalente en código fonético ICAO. El código fonético de la Organización Internacional de Aviación Civil (ICAO por sus siglas en inglés) es almacenado en un diccionario de la siguiente manera:

    claves = { "A" : "Alfa", "B" : "Bravo", "C" : "Canadá", "D" : "Delta", "E" : "Eco", "F" : "Florida", "G" : "Golf", "H" : "Hotel", "I" : "India", "J" : "Japón", "K" : "Kilo", "L" : "Lima", "M" : "Madrid", "N" : "Nancy", "O" : "Oscar", "P" : "Papa", "Q" : "Quebec", "R" : "Radio", "S" : "Sierra", "T" : "Tango", "U" : "Unión", "V" : "Víctor", "W" : "Whisky", "X" : "Equis", "Y" : "Yanqui", "Z" : "Zulo" }

Capítulo 6
Arreglos I: Vectores

6.1 Introducción
6.2 Introducción a arreglos
6.3 Vectores
6.4 Ejemplos con vectores en Python
6.5 Ordenamiento de vectores
6.6 Búsquedas
6.7 Instrucciones de Python del Capítulo 6
6.8 Conclusiones
6.8 Ejercicios

Solos podemos hacer muy poco;
unidos podemos hacer mucho.

Hellen Keller

Objetivos

Cuando agrupamos elementos del mismo tipo, el conjunto de ellos forman un arreglo. Los arreglos son estructuras de datos que facilitan el procesamiento de ellos. Los arreglos más sencillos se conocen como vectores. La forma de definir y usar vectores se cubre en este capítulo.

6.1 Introducción

En este capítulo tratamos con arreglos, que son colecciones o conjuntos de datos, por lo general de un mismo tipo pero no forzosamente.

Los arreglos se usan en una gran cantidad de aplicaciones donde se tengan muchos datos, por ejemplo, para representar y procesar los datos de cuentahabientes de un banco, en las estadísticas de un deporte, en la nómina de una empresa, solamente para mencionar algunos ejemplos. Los arreglos más comunes son los vectores y las matrices. En este capítulo cubriremos vectores y en el siguiente capítulo matrices. En los capítulos siguientes usaremos arreglos repetidamente. Además, los arreglos son la base de estructuras de datos más complejas.

6.2 Introducción a arreglos

Un arreglo es una colección finita de datos. Por ejemplo, podemos tener un arreglo de enteros, o de reales, o de variables alfanuméricas. De esta manera podemos definir un arreglo de la siguiente manera:

Un arreglo se define como una colección finita de datos.

- Se dice que la colección es finita porque de antemano se sabe cuántos elementos hay o cuántos puede haber.

- Los datos pueden ser del mismo tipo, por ejemplo, enteros, reales, alfanuméricos o lógicos, o pueden ser de tipos diferentes. Por ejemplo, algunos de los elementos de un arreglo pueden ser enteros, otros reales, otros alfanuméricos y otros lógicos.

Un arreglo debe tener un nombre y la cantidad de elementos que debe tener. Los arreglos más sencillos en la vida real son los vectores. Otros tipos de arreglos que aparecen en matemáticas son las matrices que se estudian en el Capítulo 7.

6.3 Vectores

Los arreglos más sencillos son los vectores y consisten en un conjunto de datos, por lo general, del mismo tipo. Para ilustrar de una manera sencilla el concepto, consideremos la cajonera de la figura 6.1. Los cajones están numerados del 0 al 4 para un total de 5 cajones. De esta manera podemos denotar cada cajón con la siguiente nomenclatura:

cajonera [0] ← calcetines.
cajonera [1] ← camisas.
cajonera [2] ← dulces.
cajonera [3] ← cinturones.
cajonera [4] ← perfumes.

De esta manera, el vector es cajonera y en cada cajón se encuentran artículos de distinto tipo, algunos de los cuales son para vestir y otros de otro tipo. Por ejemplo, para el tercer cajón tenemos.

cajonera [2] ← dulces

Figura 6.1 Ejemplo de un vector.

Ejemplo 6.1

Arreglo de calificaciones de estudiantes

Consideremos las calificaciones de estudiantes almacenadas en un vector llamado calificaciones y otro vector asociado conteniendo los números de matrícula. Estos dos conjuntos de datos se muestran en la tabla siguiente:

Indice	No. de Matrícula	Calificación
0	301834	9.3
1	301846	9.9
2	301847	8.6
3	301889	10.0
4	301996	9.6
5	301998	8.9
6	302020	10.0
7	302028	9.1
8	302067	10.0
9	302135	10.0

El vector de la derecha contiene las calificaciones de 10 estudiantes y el vector de

la izquieda contiene los números de matrícula de cada estudiante. Notamos que cada uno de ellos contiene diez valores. La primera columna de la tabla denota el índice cuyo valor señala la posición de cada elemento del vector. Los valores son enteros para No._de_Matrícula y reales para Calificación. Podemos ver rápidamente que el estudiante con el No. 301889 tiene una calificación de 10.0 mientras que el estudiante con el No. 302028 tiene una calificación de 9.1. Estos vectores se definen cuando se definen las variables como[1] :

Enteros: No_de_Matrícula[10], Calificación[10]

El número 10 entre corchetes indica que cada uno de ellos tiene diez componentes. A cada uno de los componentes de un vector se le llama un **elemento** del vector.

Un vector está formado por elementos.

Refiriéndonos a los vectores de la tabla anterior, la primera columna indica la posición que cada No. de Matrícula y cada Calificación ocupa en el vector. Este número recibe el nombre de **índice** el cual se define como:

El número que indica la posición de los elementos de un vector se llama índice y empieza a contarse desde cero.

Por ejemplo No_de_matrícula[1] indica al estudiante con No. 301846 mientras que Calificación[2] indica la calificación 8.6. Podemos entonces decir que el **índice** sirve para seleccionar los valores del arreglo.

Un índice indica qué elemento del arreglo se desea usar. Es una variable entera.

[1]Para Python no se necesita declarar los vectores al principio del programa, pero en otros lenguajes de programación es obligatorio declararlos antes de usarlos.

El siguiente ciclo **Para** en pseudocódigo despliega la información de los vectores:

Para estudiante ← 0 hasta 9 **:**
    Mostrar: No_de_matrícula[estudiante], Calificación[estudiante]
FinPara

produce el resultado:

301834	9.3
301846	9.9
301847	8.6
301889	10.0
301996	9.6
301998	8.9
302020	10.0
302028	9.1
302067	10.0
302135	10.0

En este ciclo, la variable estudiante debe ser entera y es el **índice** tanto para el ciclo **Para** como para cada uno de los vectores No_de_matrícula y calificación. La variable estudiante es el índice y debe variar de 0 hasta 9 para abarcar los 10 estudiantes.

Ejemplo 6.2

Suma de los elementos de un vector

En este ejemplo deseamos calcular la suma de las calificaciones del Ejemplo 6.1. Si consideramos que ya tenemos las calificaciones almacenadas en el vector Calificación que tiene 10 elementos entonces solamente nos falta realizar la suma. Como se vio en capítulos anteriores, para sumar es necesario inicializar la suma a cero y el índice estudiante a cero, es decir, realizar:

suma ← 0
estudiante ← 0

Ahora podemos realizar la suma en un ciclo **Mientras** (también se pueden usar un ciclo **Para**):

Mientras estudiante < 10 : # El índice es estudiante.
    suma ← suma + Calificación[estudiante]
    estudiante ← estudiante + 1
FinMientras

De esta manera mientras no se llegue al último estudiante, nos mantendremos sumándole las calificaciones hasta llegar al último estudiante.

6.3.1 Acceso a vectores

Cuando usamos vectores se puede tener acceso a cualquier elemento del vector usando el índice. Por ejemplo, si deseamos cambiar la calificación del cuarto estudiante de tal manera que su nueva calificación sea 10.0, hacemos lo siguiente:

calificación[3] ← 10.0

Notamos que no hicimos calificación[123669] ya que esto se referiría al estudiante colocado en esa posición y que no existe ya que solamente tenemos diez estudiantes y por lo tanto el índice varía de 0 a 9. El nombre de un vector, al igual que con cualquier nombre de variable, no se puede usar para designar ninguna otra variable, de ningún tipo y para ningún uso.

Ejemplo 6.3

Asignación de vectores

Para asignar nuevos valores a un vector y para realizar operaciones sobre los elementos de un vector también usamos los índices. Si deseamos bajarle una décima en su calificación a los estudiantes podemos usar:

Para estudiante ← 0 a 9 :
    Calificación[estudiante] ← Calificación[estudiante] - 0.1

FinPara

pero si en cambio deseamos aumentar una décima, dado que algunos estudiantes ya tienen 10.0 de calificación debemos checar que la nueva calificación no pase de 10.0. Para esto tenemos entonces:

Para estudiante ← 0 a 9 :
    Calificación[estudiante] ← Calificación[estudiante] + 0.1
    Si Calificación[estudiante] > 10.0 :
        Calificación[estudiante] ← 10.0
    FinSi
FinPara

Ejemplo 6.4

Entrada de datos a un vector

Un algoritmo puede recibir datos de 35 estudiantes y almacenarlos en un vector. Esto lo hacemos usando el índice. Por ejemplo, el pseudocódigo:

Para estudiante ← 0 a 34:
    Recibir: id[estudiante], calificación[estudiante]
FinPara

va a recibir un número de estudiante y su calificación y los va a almacenar en los correspondientes vectores id y Calificación.

Para la primera vez que se realiza el ciclo **Para** se tiene estudiante = 0 y se reciben los valores de id[0] y calificación[0]. Para la segunda vez que se realiza el ciclo **Para** se tiene estudiante = 1 y se reciben los valores de id[1] y Calificación[1]. Y así sucesivamente hasta que se tienen almacenados los id y calificaciones de los 35 estudiantes (desde 0 hasta 34 se tiene un total de 35 estudiantes).

Ejemplo 6.5

Lectura de vectores

Se desea desarrollar un algoritmo para una aplicación que reciba una lista de números de matrícula de 50 estudiantes y que los despliegue en el orden inverso a como los recibió. El vector donde se guardan los números se designa como:

Entero: no_estudiante[50]

ya que tiene 50 elementos y es de tipo entero. Para recibir los datos usamos un ciclo **Para** de la siguiente manera:

Para contador ← 0 a 49 **:**
    Recibir: no_estudiante[contador]
FinPara

En este ciclo la variable contador es el índice. En este paso ya tenemos guardados los números de estudiante de los 50 estudiantes. Para desplegarlos simplemente recorremos el índice de manera inversa empezando en 49 y terminando en 0. Entonces con otro ciclo **Para** podemos desplegar el vector en orden inverso con:

Para k ← 49 a 0 **:**
    Mostrar: no_estudiante[k]
FinPara

Ahora hemos usado como índice la variable k. En lugar del ciclo **Para** podemos usar también un ciclo **Mientras** como

k ← 49
Mientras k >= 0 **:**
    Mostrar: no_estudiante[k]
    k ← k − 1
FinMientras

El algoritmo completo es entonces:

Algoritmo: Invierte_lista
Recibe 50 IDs y los muestra en orden inverso.
Variables:

```
Entero: no_estudiante[50] # Arreglo de enteros
Entero: k, contador

INICIO
Para contador ← 0 a 49  : # El índice es contador.
        Recibir: no_estudiante[contador] # Aquí se llena el vector.
FinPara

# Ahora se recorre el vector desde la posición 49 a la 0.
k ← 49 # Ahora el índice es k. Se inicializa a 49.
Mientras k >= 0 :
        Mostrar: no_estudiante[ k ]
        k ← k - 1 #  Se decrementa el índice.
FinMientras

FIN
```

Ejemplo 6.6

Promedio de los elementos de un vector

Se desea desarrollar un algoritmo que reciba los precios de 16 productos y reporte los precios que se encuentren por arriba del promedio.

En este algoritmo debemos calcular la suma de los precios y dividirla entre 16 para obtener el promedio.

Lo primero es leer los precios. Cuando los vamos leyendo los almacenamos y podemos empezar la suma de los precios de los productos. Usamos un ciclo **Mientras** para realizar esto. Primero debemos inicializar la suma.

```
suma ← 0 # Inicializamos la suma.
producto ← 0 # Inicializamos el índice a 0.

Mientras producto <= 15: # Inicia el ciclo Mientras.
        Recibir: precio[producto]
```

```
        suma ← suma + precio[producto]
        producto ← producto + 1
FinMientras
```

En este punto del algoritmo tenemos la suma y lo único que falta es calcular el promedio y desplegarlo, lo que se realiza con:

```
promedio ← suma/16
Mostrar: promedio
```

Ahora recorremos la lista de productos y contamos cuántos están arriba del promedio. Esto lo hacemos con un ciclo **Para** y checando para cada uno de los elementos del vector si es mayor que el promedio de los precios y en este caso contar cuántos hay mayores que el promedio:

```
conteo ← 0 # Cuenta cuántos elementos están arriba del promedio.
Para i ← 0 hasta 15 :
    Si precio[i] > promedio : # El índice es i.
        conteo ← conteo + 1 # Se cumple que es mayor que el promedio.
    FinSi
FinPara
Mostrar: "El número de productos mayor que el promedio es: ", conteo
```

6.3.2 Importancia del tamaño de un vector

Cuando usamos un vector en los ejemplos anteriores hemos usado un ciclo **Para** o un ciclo **Mientras**. Estos ciclos siempre utilizan el índice que va desde el índice del primer elemento hasta el índice del último elemento. Esto es importante de tener en mente ya que el compilador asigna la memoria necesaria para almacenar todos los elementos del arreglo.

Si queremos extender el índice de un arreglo a un valor más alto del definido obtendremos un error.

En algunos lenguajes de programación no marcará error, pero ese error puede aparecer más adelante en la ejecución del algoritmo y hacer más difícil encontrarlo.

El índice de un vector debe estar dentro de los límites del tamaño del vector.

6.4 Vectores en Python

Los vectores en Python se implementan usando listas. Como se vio en el capítulo anterior, las listas y por lo tanto los vectores, pueden contener cualquier tipo de dato. La notación en pseudocódigo es similar a la de Python pero no especificamos el tipo de variable. Por ejemplo, en Python el vector del Ejemplo 6.6, se inicializa como:

```
precio = [None]*16 # Vector con 16 elementos.
```

Aquí generamos un vector vacío con la variable `None` y el tamaño del vector lo asignamos con el entero, en este caso el entero 16.

Ejemplo 6.7

Lectura de vectores en Python

Para ilustrar el manejo de los vectores convertimos el Ejemplo 6.5 a código Python, de la siguiente manera:

- Las dos primeras líneas son:

```
# Algoritmo Invierte Lista
```

```
# Recibe 50 IDs y los muestra en orden inverso.
```

- Ahora empezamos el programa principal:

- Inicializamos nuestra lista que se llama `no_estudiante`, con un vector de tamaño 50 pero vacío:

```
no_estudiante = [None]*50
```

- Con un ciclo empezamos a leer los números de estudiante y llenar el vector:

```
for contador in range(50): # El índice es contador.
      print ("Dame el número de estudiante: \n")
      no_estudiante[ contador] = int( input( ) )
```

- Ahora se recorre el vector desde la posición 49 a la 0 y se muestra cada uno de los elementos del vector. El nuevo índice es k. Se usa un ciclo **Mientras**:

```
k = 49 #  Se inicializa el índice.
while (k >= 0): # Cuando k sea negativo quiere decir que ya terminó.
      print ("\n",  no_estudiante[k] )
      k = k - 1 # Va decrementando la variable k.
```

6.4.1 Vectores por comprensión

Los vectores se pueden definir en Python usando el método de comprensión. El formato es:

```
[f(x) for x in range(A)]
```

genera un vector de valores de $f(x)$ para valores de $x = 0, 1, 2, \ldots,$ A-1. Es decir, el vector es:

$[f(0), f(1), \ldots, f(\text{A-1})]$

De esta manera un vector de 5 elementos de potencias cuadradas se puede definir por:

```
[x**2 for x in range(5)]
```

que arroja el resultado:

```
[0, 1, 4, 9, 16]
```

6.5 Ejemplos con vectores en Python

En esta sección presentamos ejemplos de algoritmos con vectores representados por listas.

Ejemplo 6.8

Vector con números aleatorios

Deseamos generar un vector formado con números aleatorios. Para esto usamos la función `random` que se encuentra en la biblioteca `random`. Esta función nos produce un número aleatorio entre 0 y 1. Por ejemplo:

```
>>> import random
>>> random.random( )
0.5338963331348068
```

Para generar un vector de valores aleatorios usamos un ciclo Para como en el siguiente algoritmo que genera 10 números aleatorios:

```
'''Calcula 10 números aleatorios '''
```

```
import random
x = [None]*10
for indice in range(10) :
     x [indice] = random.random()
     print ( x [indice] )
```

En este ejemplo la salida va a ser la siguiente (puede cambiar de máquina a máquina):

```
0.8805860396160705
0.1597687870793434
0.033504773336406224
0.5285309251866355
0.6670425804851393
0.6057228215661512
0.08082645166372637
0.056001271786772944
0.11913459999166542
0.008556894939864557
```

Si deseamos que el resultado sea una lista aleatoria de números enteros entre 35 y 59 entonces usamos la instrucción `randint( min, max)` de la siguiente manera:

```
x [k] = random.randint( límite_inferior, límite_superior )
```

El algoritmo se modifica a:

```
# Calcula 10 números enteros aleatorios.
import random
x = [None]*10
for indice in range(10) :
     x [indice] = random.randint(35, 59)
     print ( x [indice] )
```

El resultado es ahora la lista (puede cambiar de máquina a máquina) cuyos elementos están entre 35 y 59:

```
43
46
59
50
56
57
45
51
46
49

>>> x
 [43, 46, 59, 50, 56, 57, 45, 51, 46, 49]
```

Pero si deseamos obtener elementos reales entre un valor Máximo y un valor Mínimo, podemos multiplicar el resultado de la función `random` por un número máximo `Max` y luego el resultado sacarle su módulo con `(Max - Min + 1)` ya que la función módulo (%) obtiene el residuo, que sólo puede tener los valores entre 0 y el valor `Max` y le sumamos `Min`. Esto lo podemos hacer con:

```
x[indice] = Min + Max*random.random() % (Max - Min + 1)
```

El siguiente algoritmo genera números aleatorios entre 50 y 100.

```
'''Calcula 10 números aleatorios entre 50 y 100. '''

x = [None]*10
min = 50; max = 100
for indice in range( 10 ):
   x[indice] = min + max*random.random()%(max - min + 1)
   print ( x [indice] )
```

Ejemplo 6.9

Vector con números aleatorios sin declarar el vector

En el ejemplo anterior declaramos el vector de 10 elementos con `x = [None]*10`. Otra forma de hacerlo es ir generando el vector como lo vamos calculando. Esto lo podemos hacer con:

`x = [ ]`	Para inicializar el vector.
`x += [50 + 100*random.random()%(51)]`	Para generar el vector.

con lo que el algoritmo para generar número reales aleatorios entre 50 y 100 es:

"'Calcula 10 números aleatorios entre 50 y 100.
No se declara el tamaño del vector"'

```
import random # Se importa la biblioteca random.

lim_inf = 50; lim_sup = 100
x = [ ]
for indice in range( 10 ):
   x += [lim_inf + lim_sup*random.random()%(lim_inf + 1)]
   print ( x[indice] )
```

Ejemplo 6.10

Suma de los elementos de un vector

Para sumar los elementos de un vector primero debemos inicializar la suma a cero para empezar a añadirle uno por uno los elementos del vector. El siguiente algoritmo realiza la suma de los elementos de un vector de tamaño 5. Lo primero que tenemos que hacer es recibir los elementos del vector. El segundo paso es inicializar la suma y con un ciclo realizar la suma de los elementos. El tercer y último paso es mostrar el resultado. A continuación presentamos el algoritmo en pseudocódigo y en Python:

	Pseudocódigo	Python
1	**Algoritmo:** Suma de vector.	''' Suma de vector.
2	''' Calcula la suma de los	Calcula la suma
3	elementos de un vector.'''	elementos de un vector.'''
4	**INICIO**	
5	**Variables:**	
6	**Entero:** cont;	
7	**Real:** vector [5], suma←0	`suma = 0; vector= [None]*5`
8	# Lee vector y realiza la suma.	# Lee vector y realiza la suma.
9	**Mostrar:** "Ingrese cinco enteros"	
10	**Para** cont de 0 a 4 :	`for cont in range(5):`
11	**Recibir:** vector [cont]	`a=input('Dame un elemento:')` `vector [cont] = a`
12	# Realiza la suma.	# Realiza la suma.
13	suma ←suma+vector[cont]	`suma = suma+vector [cont]`
14	**Fin_Para**	# **Fin_for**
15	**Mostrar:** "suma", suma	`print("suma", suma)`
16	**FIN**	# **FIN**

Ejemplo 6.11

Promedio, varianza y desviación estándar de un vector

Una de las funciones más usadas en una lista o colección de números o datos es el valor medio, conocido también como valor esperado en probabilidad y estadística. Se calcula como el valor promedio de la colección de números o datos. También se puede considerar como el centro gravedad de la colección o lista de datos. Para los elementos de un vector X[i] se puede calcular el promedio el cual se define por la siguiente ecuación:

$$\overline{X} = \frac{1}{n}\sum_{i=0}^{n-1} X[i]$$

También podemos definir la varianza σ^2, para lo cual necesitamos usar el promedio que definimos arriba. La varianza también se conoce como la dispersión y es una

medida de la dispersión de las elementos de la lista alrededor del centro de gravedad o promedio de la lista. Una varianza grande indica que hay muchos valores de la lista lejos del valor del promedio. Por el contrario, si la varianza es pequeña, entonces los valores de la lista están cerca del valor promedio. Podemos definir la varianza σ^2 mediante la ecuación:

$$\sigma^2 = \text{Varianza} = \frac{1}{n}\sum_{i=0}^{n-1}\left(X[i] - X\right)^2$$

Finalmente, la desviación estándar σ está definida por la raíz cuadrada de la varianza. La desviación estandar tiene el mismo significado de la varianza, pero tiene las mismas unidades de los elementos de la lista. La desviación estándar σ está definida por:

$$\sigma = \text{Desviación Estándar} = \sqrt{\text{Varianza}}$$

El siguiente algoritmo realiza estos cálculos:

	Pseudocódigo	Python
1	**Algoritmo:** Varianza	# Varianza y desv. estándar
2	''' Calcula varianza y	
3	desviación estándar	
4	**INICIO**	# **INICIO**
5	**Variables:**	
6	**Entero:** cont, k	
7	**Real:** vector [5], suma←0	`suma = 0; vector= [None]*5`
8	**Real:** var, desv, prom	
9	# Importa la biblioteca	`from math import sqrt`
10	# Lee vector y realiza la suma.	# Lee vector y realiza la suma.
11	**Mostrar:** "Ingrese cinco enteros"	
12	**Para** cont de 0 a 4 :	`for cont in range(5):`
13	**Mostrar:**"Dame un elemento:"	`print( "Dame un elemento:" )`
14	**Recibir:** vector [cont]	`vector [cont] =int(input())`
15	# Realiza la suma.	# Realiza la suma.
16	suma ←suma+vector [cont]	`suma = suma+vector [cont]`
17	**Fin_Para**	

Continúa

	Pseudocódigo	Python
18	# Calcula el promedio	# Calcula el promedio
19	pr← suma/5	pr = suma/5
20	# Calcula varianza	#Calcula varianza
21	**Para** k de 0 a 4 :	for k in range(5):
22	var←var+(vector [k]-pr)∧2	var=var+pow(vector [k]-pr,2)
23	var←var/5	var = var/5
24	# Calcula Desviación Estándar	# Calcula Desviación Estándar
25	desv← sqrt(var)	desv = sqrt(var)
26	**Fin_Para**	# **Fin del for**
27	**Mostrar:** var, desv	print("Varianza,Desv.Est.")
28		print(var, desv)
29	**FIN**	# **FIN**

Ejemplo 6.12

Promedio de calificaciones

El siguiente algoritmo recibe una lista de calificaciones, las almacena en un vector, muestra la media así como las calificaciones arriba de la media. Lo primero que hacemos es pedir el número de estudiantes N. El siguiente paso es recibir las calificaciones, almacenarlas y realizar la suma. Esto lo hacemos con ciclos **Para**:

Para i ← 0 hasta N :
  Mostrar: "Alumno", i + 1, "Nota final"
  Recibir: notas [i]
FinPara
Para i ←0 hasta N :
  suma ← suma + notas [i]
FinPara

La media se calcula con:

media ← suma/N

Para realizar la tarea deseada, en un ciclo **Para** comparamos cada una de las calificaciones con la media. Si son mayores que la media las desplegamos:

Mostrar: "Notas superiores a la media"
  Para i ← 0 hasta i < N :
    Si notas [i] > media :
      Mostrar: "Alumno numero: ", i + 1
      Mostrar: " Nota final: ", notas [i]
    FinSi
  FinPara

El algoritmo completo tanto en pseudocódigo como en Python se muestra a continuación:

	Pseudocódigo	Python
1	**Algoritmo:** Prom. calificaciones	# **Promedio de calificaciones.**
2	**INICIO**:	# **INICIO**
3	**Real:** notas [20], suma ← 0, media	`suma = 0; notas = [None]*20`
4	**Entero:** i, N	
5	**Mostrar:** "Número de alumnos?"	`print("Número de alumnos? ")`
6	**Recibir:** N	`N = int(input( ) )`
7	**Para** i←0 hasta i < N :	`for i in range(N):`
8	**Mostrar:** "Alumno",i+1,"Nota:"	`print ( 'Alumno', i+1, 'Nota:')`
9	**Recibir:** Notas[i]	`notas[i] = float ( input( ) )`
10	**FinPara**	
11	**Para** i←0 hasta i < N :	`for i in range(N):`
12	suma ← suma + notas[i]	`suma = suma + notas[ i ]`
13	**FinPara**	
14	media = suma/N	`media = suma/N`
15	**Mostrar:** "Nota media: ", media	`print( "Nota media: " , media)`
16	**Mostrar:** "Notas superiores"	`print(" Notas superiores ")`
17	**Para** i←0 hasta i < N :	`for i in range (N):`
18	**Si** notas[i] > media :	`if(notas [i] > media):`
19	**Mostrar:**'Alumno número:', i+1	`print('Alumno número:', i+1 )`
20	**Mostrar:**" Nota: ", notas [i]	`print (" Nota: ", notas[i] )`
21	**FinSi**	
22	**FinPara**	
23	**FIN**	# **FIN**

Ejemplo 6.13

Ejemplo 6.13 Palíndromos

Una palabra o un texto es un palíndromo si se puede leer de izquierda a derecha o de derecha a izquierda y se lee lo mismo (ignorando los espacios y acentos). Para saber si el texto es un palíndromo, la primera letra de la frase y la última deben ser iguales. Lo mismo debe suceder con la segunda y penúltima letras, y así sucesivamente hasta llegar a las letras del centro. En este ejemplo queremos formular un algoritmo que lea una palabra o una frase y que nos diga si es un palíndromo. Los pasos que debemos seguir son los siguientes:

- Primero debemos saber el número de caracteres en el texto.
- A continuación debemos leer caracter por caracter del texto.
- Para saber si es palíndromo procedemos a comparar los caracteres primero y último, para seguir con el segundo y el penúltimo. Continuamos en esta manera hasta llegar al centro.
- Si en alguna de las comparaciones los caracteres no son iguales, se interrumpe el proceso y se indica que el texto dado no es palíndromo.
- Si el texto es palíndromo, se muestra un mensaje diciendo que el texto sí es palíndromo.

En pseudocódigo el algoritmo puede escribirse como:

Algoritmo: Palíndromos 1 #determina si una frase es un palíndromo.

Variables:
Arreglo de alfanuméricos: frase[50], letra[50]
Entero: número, max, inverso, contador

INICIO

Leemos la longitud de la frase sin contar los espacios.
Recibir: max
número ← 0 # Se inicializa el índice.

```
    # Leemos la frase en el vector letra.
    Mientras número <= max :
        Recibir: letra[número]
         número ← número + 1
    FinMientras

    inverso ← max
    número ← 0

# Asignamos al vector frase lo que contiene el vector letra en orden invertido.
    Mientras número <= max :
        frase[inverso] ← letra[número]
        inverso ← inverso -1
        número ← número + 1
    FinMientras

    número ← 1
    contador ← 0

# Comparamos las letras de ambos vectores.
    Mientras número <= (max/2 + 1) :
        Si frase[número] = letra[número] :
            contador ← contador + 1
            número ← número + 1
        FinSi
    FinMientras

    Si contador == (max/2 + 1):
         Mostrar: "Sí es palíndromo"
    Si_no :
        Mostrar: "No es palíndromo"
    FinSi

    FIN
```

En Python el código es:

```
# Algoritmo Palíndromos
# Determina si una frase es palíndromo.
numero = 0
contador = 0
print( " Dame el número de letras de la frase:")
max = int( input( ))
print ( "Dame las letras de la frase:")
letra = [ ]
while(numero <= max - 1 ):
      print( "Dame una letra de la frase:")
      letra += [ input( )]
      numero = numero + 1

inverso = max
numero = 0
frase = [None]*max

while(numero <= max - 1:
      frase[inverso - 1] = letra[numero]
      inverso = inverso - 1
      numero = numero + 1

numero = 0
contador = 0

while(numero <= ceil(max/2)):
      if (frase[numero] == letra[numero]):
          contador = contador + 1
       numero = numero + 1

if (contador == ceil(max/2) + 1 ):
     print(  "La frase SI es palindromo." )
```

```
else:
     print( "La frase NO es palindromo." )
```

6.5.1 Importancia del tamaño de un vector en Python

Como se mencionó antes, es muy importante tomar en cuenta el tamaño de un arreglo ya que se pueden ocasionar errores al tratar de usar elementos no definidos para un arreglo. Para ilustrarlo consideremos el siguiente ejemplo:

Ejemplo 6.14

Tamaño de un vector

Consideremos el siguiente algoritmo que genera un vector de tamaño 10 (de 0 a 9) usando un ciclo **Para** y que en otro ciclo **Para** escribe el vector pero del elemento 0 al 19. Obviamente los elementos de la posición 10 a la 19 no están definidos y ocasionará un error al momento de la ejecución.

```
# Algoritmo con error en tamaño del vector
X = [None]*10
for k in range(10): # Asigna los 10 valores de X.
   X[k] = k

# Ahora intenta asignar más elementos de X pero de la posición 10 a la 19.

 for k in range(10, 20): # Trata de asignar e imprimir.
   X[k] = k # Asigna los valores de X[10] a X[19].
   print ( X[k] ,"\n")
```

Este programa en Python causará error y no podrá ejecutarse por la mayoría de los compiladores.

Una observación importante acerca del tamaño de los vectores la enunciamos en el siguiente recuadro:

> Es muy importante respetar el tamaño de los arreglos para evitar errores al ejecutar los algoritmos.

6.5.2 Inicialización de los vectores

Un vector se puede inicializar al momento de definirlo. Por ejemplo, si `A` es un vector de tamaño 10 y queremos crearlo vacío usamos:

```
A = [ None ]*10
```

Pero si queremos inicializarlo con otros valores usamos, por ejemplo:

```
A = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

6.5.3 La instrucción append

El tamaño de un vector se puede modificar añadiendo más elementos usando la instrucción `append`. Por ejemplo, para agregar el elemento `10` al vector `A` utilizamos

```
>>> A.append(10)
>>> A
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Como vemos ahora el vector `A` tiene 11 elementos y el último elemento es `10`.

Ejemplo 6.15

Ejemplo 6.15 Serie de Fibonacci

La serie de Fibonacci es una secuencia de números en donde cada elemento de la serie se forma por la suma de los dos anteriores. Los dos primeros elementos de la serie son 0 y 1. De esta manera la serie de Fibonacci se forma mediante la ecuación

$$\text{fib}[n] = \text{fib}[n-1] + \text{fib}[n-2]$$

Usando esta ecuación los términos son: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Un algoritmo que obtiene los primeros 10 elementos de la serie de Fibonacci y los almacena en un vector `fib` necesita tener los dos primeros elementos de la serie almacenados en el vector `fib`. Estos dos primeros elementos del vector `fib` son `fib[0] = 0` y `fib[1] = 1`. Para hacer esto necesitamos inicializar el vector `fib` como:

```
fib ← [ 0, 1 ]
```

Los elementos a partir de `fib[2]` los generamos con un ciclo **Para** como:

```
Para k in range( 2, 10 ):          for k in range(2, 10) :
   fib[k]← fib[k-1] + fib[k-2]        fib.append(fib[k-1] + fib[k-2])
FinPara
```

Finalmente, desplegamos el vector `fib` con con:

Mostrar: fib

El algoritmo en pseudocódigo y en Python se muestra a continuación:

	Pseudocódigo	Python
1	**Algoritmo:** Fibonacci.	# Algoritmo: Fibonacci.
2	# Genera la secuencia de	# Genera la secuencia de
3	# Fibonacci.	# Fibonacci.
4	**INICIO**	#**INICIO**
5	**Variables:**	
6	**entero** fib[10] ← {0, 1}, k	`fib = [ ]`
7		`fib[0:1] = [ 0, 1 ]`
8	**Para** k ← 2 hasta k <10 :	`for k in range ( 2, 10 ):`
9	fib[k] ← fib[k-1] + fib[k-2]	`fib.append(fib[k-1] + fib[k-2])`
10	**Fin_Para**	
11	# Muestra la secuencia.	# Muestra la secuencia.
12	**Para** k ← 0 hasta k <10	`print ( fib )`
13	**Mostrar:** fib[k]	
14	**Fin_Para**	
15	**FIN**	# **FIN**

6.6 Ordenamiento de vectores

Los elementos de un vector se pueden ordenar de alguna manera según sea necesario. Las más comunes son el ordenamientoven orden ascendente y el ordenamiento en orden descendente. Existe un gran número de algoritmos para ordenamiento de vectores, entre los que podemos mencionar los algoritmos por selección, de burbuja, por inserción, para mencionar algunos. En esta sección describimos el ordenamiento por selección y de burbuja. Un tratamiento más extenso de algoritmos de ordenamiento está más allá de los alcances de este libro.

6.6.1 Ordenamiento por selección

El algoritmo de ordenamiento en orden descendente por selección funciona buscando el elemento más pequeño del vector e intercambiándolo por el elemento que está en la primera posición. Por ejemplo, en el vector:

$$\begin{bmatrix} 4 \\ 8 \\ 6 \\ 11 \\ 3 \\ 7 \\ 9 \end{bmatrix}$$

el elemento más pequeño es 3. En el ordenamiento por selección intercambiamos 4 por 3 para obtener el vector:

$$\begin{bmatrix} \rightarrow & 3 \\ & 8 \\ & 6 \\ & 11 \\ \rightarrow & 4 \\ & 7 \\ & 9 \end{bmatrix}$$

Ahora que ya tenemos el elemento más pequeño en la primera posición buscamos el siguiente en tamaño que en nuestro vector es el 4, el cual colocamos en la segunda posición intercambiándolo con el 8. Esto nos cambia nuestro vector a:

$$\begin{bmatrix} & 3 \\ \rightarrow & 4 \\ & 6 \\ & 11 \\ \rightarrow & 8 \\ & 7 \\ & 9 \end{bmatrix}$$

Vemos que el siguiente elemento en orden ascendente es el 6, así que ya no hacemos nada y procedemos al siguiente elemento. Ahora tenemos que intercambiar el 11 y el 7 para obtener:

$$\begin{bmatrix} & 3 \\ & 4 \\ & 6 \\ \rightarrow & 7 \\ & 8 \\ \rightarrow & 11 \\ & 9 \end{bmatrix}$$

Vemos que también el 8 está ya en su posición y el último intercambio es entre el 11 y el 9 para finalmente producir el vector:

$$\begin{bmatrix} 3 \\ 4 \\ 6 \\ 7 \\ 8 \\ 9 \\ 11 \end{bmatrix}$$

el cual ya está en orden ascendente. La complejidad del algoritmo se mide en términos de las operaciones de comparación que se realizan para completar el ordenamiento.

En el caso del algoritmo por selección se tiene una complejidad expresada por $O(n^2)$ donde n es el número de elementos en el vector. Esto quiere decir que el máximo número de intercambios es del orden de n^2.

Ejemplo 6.16

Ordenamiento por selección

Un algoritmo en pseudocódigo que realiza este ordenamiento por selección es el siguiente:

```
Algoritmo : Ordenamiento_por_selección
# Ordena ascendentemente 100 calificaciones.

Variables:
Arreglo de reales: calif[100], calif_ord[100];
Entero: paso, estudiante, mínima;

INICIO

Para estudiante ← 0 a 99 : # Almacena las calificaciones.
        Recibir: calif[estudiante]
FinPara

Para paso ← 0 a 99 :
        mínima ← 0

        Para estudiante ← 1 a 99 :
                Si calif[estudiante] < calif[mínima] :
                        mínima ← estudiante
            FinSi
        calif_ord[paso] ← calif[mínima] # copia la mínima al vector ordenado.

        calif[mínima] ← 11 # para que no se considere en el siguiente paso.

        FinPara

 FinPara
```

Para estudiante ← 0 a 99 : # despliega el vector ordenado.
    Mostrar calif_ord [estudiante]
FinPara

FIN

En Python, el código queda como se muestra a continuación (solamente para 7 estudiantes, del 0 al 6):

#Algoritmo de ordenamiento por selección

```
calif = [None]*6; calif_ord = [None]*6

for estudiante in range(6):
     print( "Dame las calificaciones:" )
     calif[estudiante] = float( input( ) )

for paso in range(6) :
      minima = 0
      for estudiante in range( 1, 6 ) :
         if (calif[estudiante] < calif[minima]) :
             minima = estudiante
         calif_ord[paso] = calif[minima]
         calif[minima] = 11

# Se despliega el vector ordenado.
print ( "El orden es: \n" )
print ( calif_ord )
```

Ejemplo 6.17

Ordenamiento de un vector aleatorio

En este ejemplo deseamos ordenar el vector aleatorio del Ejemplo 6.8. Para este propósito debemos añadir la sección del código correspondiente al ordenamiento por selección, que es la siguiente:

```
# Ordena el contenido del vector
for paso in range(n) :
   min = paso
      for i in range(paso, n) :
            if (x[i] < x[min]) :
               min = i
      # intercambia entre x[paso] y x[min]
      temp = x[paso]
      x[paso] = x[min]
      x[min] = temp
# muestra el vector ordenado
print ( )
for j in range(n) :
      print( x[ j] , "\n")
```

6.6.2 Ordenamiento de burbuja

Otro algoritmo popular aunque ineficiente es el ordenamiento de burbuja. El nombre lo recibe por el hecho de que si se ordena en orden ascendente, los números más pequeños van subiendo como burbujas mientras que los más grandes se van hundiendo. La estrategia consiste en comparar los números e intercambiar posiciones si el primero de los dos es mayor que el segundo. Este proceso se repite hasta llegar a la última posición en el vector. Una vez que se tiene esto vemos que el número mayor se colocó en la última posición. El proceso se repite hasta que los elementos estén ordenados en el orden requerido. Para ilustrar el proceso consideremos el siguiente ejemplo:

Ejemplo 6.18

Ordenamiento de precios de un producto

Se tienen 10 productos cuyos precios se quieren ordenar en orden ascendente. Los precios son:

$100.25	$97.00	$12.50	$23.75	$5.60
$4.10	$25.00	$15.40	$117.55	$75.00

Estas cantidades ordenadas en un vector llamado precios quedan como:

$$\begin{bmatrix} 100.25 \\ 97.00 \\ 12.50 \\ 23.75 \\ 5.60 \\ 4.10 \\ 25.00 \\ 15.40 \\ 117.55 \\ 75.00 \end{bmatrix}$$

El algoritmo entonces principia comparando los dos primeros elementos del vector y los intercambia ya que 97.00 es menor que 100.25, lo que lo cambia nuestro vector al primer vector de la figura 6.2. Se continúa comparando el segundo elemento que ahora es 100.25 con el siguiente elemento que es 12.50. Como el 100.25 es mayor que 12.50 se intercambian para quedar como se muestra en el segundo vector de la figura. Comparamos ahora el tercer elemento con el cuarto elemento, es decir, comparamos 100.25 con 23.75. Como 100.25 es mayor que 23.75 los intercambiamos para obtener el tercer vector de la figura. Vemos que el valor de 100.25, al ir comparándose con los elementos debajo de él se va hundiendo en el vector hasta que encuentre uno mayor que él. Esto sucede hasta llegar arriba del 117.75 donde 100.25 es el menor de los dos y hasta ahí deja de hundirse. El vector en ese instante es el cuarto vector de la figura. Notamos como los elementos más pequeños empiezar a subir como burbujas y de ahí el nombre del algoritmo. La última comparación de este primer paso es el intercambio de los elementos 117.75 y 75.00 que produce el quinto vector de la figura 6.2. Esto constituye el primer paso en el algoritmo de burbuja.

$$\begin{bmatrix} 97.00 & 97.00 & 97.00 & 97.00 & 97.00 \\ 100.25 & 12.50 & 12.50 & 12.50 & 12.50 \\ 12.50 & 100.25 & 23.75 & 23.75 & 23.75 \\ 23.75 & 23.75 & 100.25 & 5.60 & 5.60 \\ 5.60 & 5.60 & 5.60 & 4.10 & 4.10 \\ 4.10 & 4.10 & 4.10 & 25.00 & 25.00 \\ 25.00 & 25.00 & 25.00 & 15.40 & 15.40 \\ 15.40 & 15.40 & 15.40 & 100.25 & 100.25 \\ 117.55 & 117.55 & 117.55 & 117.55 & 75.00 \\ 75.00 & 75.00 & 75.00 & 75.00 & 117.55 \end{bmatrix}$$

Figura 6.2 Pasos del algoritmo de burbuja del Ejemplo 6.18.

Vemos en el siguiente paso que el 97.00 se va a empezar a hundir hasta quedar arriba del 100.25 el cual a su vez se hundirá para quedar arriba del 117.55. El vector resultante queda como se muestra en el primer vector de la figura 6.3. En el siguiente paso se hunde el 23.75 hasta quedar arriba del 25.00 el que a su vez se hunde y queda arriba del 97.00 que se hunde y queda arriba del 100.25 como se muestra en el segundo vector de la figura 6.3. El tercer vector de la figura muestra como hunde el 12.50 hasta quedar arriba del 23.75, el cual a su vez se intercambia con el 15.40. Notamos en este tercer vector que ya solamente falta de intercambiar el 4.10 con el 5.60, lo cual al realizarse produce el vector final. Vemos como en todo el algoritmo los números más pequeños fueron subiendo como burbujas.

$$\begin{bmatrix} 12.50 & 12.50 & 5.60 & 4.10 \\ 23.75 & 5.60 & 4.10 & 5.60 \\ 5.60 & 4.10 & 12.50 & 12.50 \\ 4.10 & 23.75 & 15.40 & 15.40 \\ 25.00 & 15.40 & 23.75 & 23.75 \\ 15.40 & 25.00 & 25.00 & 25.00 \\ 97.00 & 75.00 & 75.00 & 75.00 \\ 75.00 & 97.00 & 97.00 & 97.00 \\ 100.25 & 100.25 & 100.25 & 100.25 \\ 117.55 & 117.55 & 117.55 & 117.55 \end{bmatrix}$$

Figura 6.3 Pasos subsecuentes del algoritmo de burbuja.

La complejidad computacional de este algoritmo de burbuja se ve claramente que es mayor que la del algoritmo de selección y es adecuado solamente cuando los elementos NO están muy desordenados. En ese caso la complejidad es del orden de $O(n^2)$ como era el caso del algoritmo de selección.

El algoritmo de burbuja en pseudocódigo es el siguiente:

Algoritmo : Ordenamiento_de_burbuja
#Ordena ascendentemente 100 precios sin usar espacio adicional.

Variables
Entero: precio[100] **#Los precios se almacenan en un vector.**
Entero: paso, artículo, temp
Lógica: hay_cambio **#Variable lógica que indica que hay cambio.**

```
INICIO

"'Primero se leen los precios y se almacenan en el vector precio
El índice del ciclo Para es la variable artículo"'
Para artículo ← 0 a 99 : # Almacena las precios.
        Recibir: precio[artículo]
FinPara
hay_cambio = Verdadero

Mientras hay_cambio == Verdadero :
        hay_cambio ← Falso
        Para artículo ← 0 a 98 :

                #Se comparan dos elementos contiguos.
                Si precio[artículo] > precio[artículo + 1] :
                    #intercambia precios si es necesario.
                    temp ← precio [artículo]
                    precio[artículo] ← precio[artículo + 1]
                    precio [artículo + 1] ← temp

                    hay_cambio ← Verdadero
                FinSi
            FinPara

    #El ciclo Mientras termina cuando ya no hay cambios.
    Fin_Mientras

    Para artículo ← 0 a 99 : # Despliega el vector ordenado.
        Mostrar: precio [artículo]
    FinPara

    FIN
```

En código Python el algoritmo anterior es el siguiente:

```
"' Algoritmo : Ordenamiento_de_burbuja
Ordena ascendentemente 100 calificaciones sin usar espacio adicional."'

for articulo in range(100): # Almacena los precios.
      print(" Dame el precio de un articulo: \n")
      precio [articulo] = float(input( ) )
hay_cambio = True

while (hay_cambio == True): # Principia el ciclo Mientras.

      hay_cambio = False
      for articulo in range( len(precio)-1 ) :

         if (precio [articulo] > precio [articulo+ 1]):
              temp = precio [articulo] # intercambia precios.
              precio [articulo] = precio [articulo+1]
              precio [articulo + 1] = temp
              hay_cambio = True
# Termina el ciclo Mientras.

# Despliega el vector ordenado.
print ("\n El vector ordenado es: \n")
for articulo in range(99) :
     print ( precio [articulo]  "\n")
```

6.7 Búsquedas

Un proceso de búsqueda se lleva a cabo cuando se quiere localizar la posición de un elemento en un vector. Por lo general las técnicas de búsqueda se aplican después de aplicar algún algoritmo de ordenamiento.

6.7.1 Búsqueda binaria

Uno de los algoritmos más comunes, simples y eficientes en la búsqueda es el algoritmo de búsqueda binaria. Consiste en dividir una lista ordenada en dos partes y decidir en qué parte está el número buscado. Dependiendo en que mitad de la lista se encuentra, entonces repetimos la acción y volvemos a partir en dos esa mitad. Así continuamos hasta que llegamos a encontrar el número. Esta es la manera en que buscamos, por ejemplo, un nombre en un directorio. Los siguientes pasos resumen el proceso:

1. Buscar el punto medio para partir la lista en dos partes.
2. Checamos si el punto medio es el valor buscado. En este caso la búsqueda termina.
3. Si el número buscado es mayor que el punto medio, repetir el paso 1, pero sólo para la mitad superior. Si no, entonces hacerlo para la mitad inferior.
4. Repetir los pasos 1, 2 y 3 hasta que se llegue al número buscado.
5. Si no se encuentra en estos pasos, entonces el número no existe en la lista.

Ejemplo 6.19

Búsqueda binaria

Supongamos que tenemos la lista ordenada de 15 números:

17, 23, 62, 140, 193, 203, 312, 439, 780, 793, 812, 827, 901, 916, 974

y que deseamos buscar el 916. Los pasos son para este ejemplo:

1. El punto medio es 439 y es distinto de 916.
2. Como 916 > 439 repetimos el paso 1 con la mitad superior que es:

   780, 793, 812, 827, 901, 916, 974

3. Al repetir el punto 1, la lista se divide en dos partes:

   780, 793, 812 y 827, 901, 916, 974 Si tomamos 812 vemos que 812 < 916, así que volvemos a tomar la mitad superior y la partimos en dos.

4. Las dos partes son: 827, 901 y 916, 974
5. Si tomamos 901 vemos que 901 < 916, así que tomamos la mitad superior.
6. Al partir la lista tenemos 916 y 974
7. Al tomar el 916 llegamos al número deseado.

Un algoritmo en pseudocódigo que realiza la búsqueda binaria es:

```
Nombre: Búsqueda binaria
#Este algoritmo de búsqueda usa la partición de la lista en dos mitades.
INICIO
    Variables:
    Real: Lista[20], número
    Para indice ← 0 hasta 19 :
        Recibir: Lista[indice]
    Fin_para
    Recibir: número
    # Se parte la lista en 2 partes.
    Para índice ← 1 hasta 10 :
            Lista1[índice] ← Lista[índice]
            Lista2[índice] ← Lista[índice + 10]
    Fin_para
    primero ← 1
    último ← número_buscado
    medio ← (primero + último)/2
    Mientras (primero <= último) :
        Si (vector[medio] < númerobuscado) :
            primero ← medio + 1
        Si_No
            Si (vector[medio] == número_buscado) :
              Mostrar: "El número buscado está en la posición ", medio
              primero = 2*último; #Para salir del ciclo mientras.
            Fin_Si
        Si_No :
```

último ← medio
medio ← (primero + último)/2
Fin_Si
Mientras
Si(primero > último)
Mostrar: "No se encontró el número en la lista.";
Fin_Si
Fin

La implementación en lenguaje Python de este algoritmo es:

```
n = int(input( "Dame el número de elementos en la lista") )
n = int(input(  "Dame ", n, " enteros")
for c in range(n+1) :
     vector[c] = int(input( ) )

numero_buscado = int(input( "Dame el valor a buscar: \n" )

primero = 1
ultimo = n
medio = (primero+ultimo)/2
while( primero <= ultimo ):
     if ( vector[medio] < numero_buscado ):
         primero = medio + 1
     elif ( vector[medio] == numero_buscado ):
         print ( numero_buscado , "encontrado en la posición ")
         print ( medio )
         primero = 2*ultimo #Para salir del ciclo.
     else:
         ultimo = medio
         medio = (primero + ultimo)/2
 # Fin del ciclo Mientras.
 if ( primero > ultimo ):
     print ( "Número no encontrado en la lista.\n")
# Fin del algoritmo de búsqueda binaria.
```

6.8 Instrucciones de Python del Capítulo 6

Instrucción	Descripción
append	Permite aumentar elementos a un vector.

6.9 Conclusiones

Los vectores son muy importantes en el estudio de las ciencias, las matemáticas, la computación, la contaduría, finanzas, las artes, y en muchas otras áreas del conocimiento. En este capítulo hemos cubierto la clase de arreglos conocidos como vectores, los cuales son arreglos unidimensionales. A través de ejemplos hemos presentado al lector con la forma de manejar los vectores y usarlos en distintas aplicaciones. También incluimos el uso de vectores en algoritmos de ordenamiento y de búsqueda. Esto incluye definir sus principales características.

6.10 Ejercicios

1. En Python, los vectores se representan por listas y por lo tanto obedecen reglas de listas. Dadas las listas `a = [1, 4, -2, 9]`, `b = [-9, 0, 4, 7, -8]`. Calcule `a + b`, `a - b` y `3*a`.

2. Dado el vector `a = [23, 27, 02, 58, 19, -18, -1, 32, 92]`. Encuentre:

   a) `a [3]`
   b) `a [3:5]`
   c) `a [:2]`
   d) `a [0:2]`

3. La magnitud o norma Euclidiana de un vector $x = [x_1, x_2, ..., x_n]$ se define como:

$$|x| = ||x|| = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$$

   Escriba un programa para calcular la magnitud de un vector. Use el vector $x =$ [2, -4, 6, -3, -9].

4. La norma infinita de un vector se denota por $(\|x\|_\infty)$ se define como el elemento con la mayor magnitud. Para el vector del Ejercicio anterior es $(\|x\|_\infty) = 9$. Escriba un algoritmo que calcule la norma infinita de un vector.

5. La norma p se define por:

$$||x||_p = (x_1^p + x_2^p + ... + x_n^p)^{1/p}$$

Calcular las normas 3 y 4 para los vectores del Ejercicio 1.

6. El producto escalar de dos vectores $x = [x_1, x_2, ..., x_n]$, $y = [y_1, y_2, ..., y_n]$, también llamado producto punto, se define por:

$$x \cdot y = x_1 \cdot y_1 + x_2 \cdot y_2 + ... + x_n \cdot y_n$$

Escriba un programa para calcular el producto escalar de dos vectores.

7. El producto escalar también se define como el producto de las magnitudes de los vectores por el coseno del ángulo que forman. De esta manera se tiene:

$$x \cdot y = |x| \cdot |y| \cos(\theta)$$

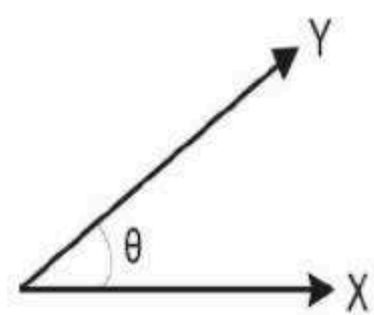


Figura 6.4 Ángulo entre dos vectores.

Dados dos vectores x y podemos visualizar esta ecuación en la figura 6.4. Diseñe un algoritmo que calcule el ángulo entre dos vectores. Para los vectores $x = [3, 4, 5]$, $y = [-2, -1, 6]$ encuentre el ángulo que forman los dos vectores.

8. El trabajo W en joules realizado por una fuerza F (newtons) sobre una partícula que se desplaza d metros está dado por:

$$W = F \cdot d$$

Realice un algoritmo y su implementación en Python que encuentre el trabajo si $F = [5, 2]$, $d = [2, 3]$ y el ángulo entre los vectores F y d.

9. Use el algoritmo de ordenamiento por selección para ordenar un vector aleatorio de 20 elementos.

10. Genere un vector aleatorio de 15 elementos y use el algoritmo de ordenamiento de burbuja para ordenarlo.

11. Genere un vector aleatorio de 10 elementos enteros, seleccione el penúltimo elemento y use el algoritmo de búsqueda binaria para localizarlo.

12. Supongamos que tenemos la lista ordenada

```
[2, 4, 5, 7, 10, 11, 13, 14, 16, 17]
```

Si se realiza una búsqueda binaria, ¿qué secuencia de comparaciones es la correcta para encontrar el 7?

a) 10, 4, 5, 7
b) 11, 5, 10, 7
c) 2, 4, 5, 7
d) 17, 12, 5, 7

Capítulo 7
Arreglos II: Matrices

7.1 Introducción
7.2 Matrices
7.3 Arreglos en Python
7.4 Métodos alternos de escritura de matrices
7.5 Selección de filas y columnas de un arreglo
7.6 Suma y multiplicación de matrices
7.7 Matrices especiales
7.8 Ejemplos
7.9 Conclusiones
7.10 Ejercicios

La técnica es el esfuerzo para ahorrar esfuerzo.

13. José Ortega y Gasset

Objetivos

Las matrices son estructuras de datos que tienen un uso muy importante en las matemáticas. En Python, las matrices se representan como listas de listas. Se define la manera de representar matrices como listas de listas y se definen las matrices básicas y su uso por medio de ejemplos.

7.1 Introducción

Las matrices surgen en matemáticas cuando deseamos manejar datos que se pueden almacenar en forma de vectores pero que además requieren el uso de varios vectores para manejar la información. Por ejemplo, supongamos que deseamos manejar las ventas de un departamento de una tienda departamental. Si solamente se trata de las ventas mensuales de un departamento en un año, entonces podemos usar un vector para representar estos datos donde cada elemento del vector corresponde a las ventas de un mes. Pero si se trata las ventas de varios departamentos, entonces lo más adecuado es el uso de matrices.

Otro ejemplo lo representan las fuerzas que actúan sobre una estructura tridimensional o sobre un puente. Quizás el ejemplo más común es la resolución de un sistema de ecuaciones simultáneas lineales donde los coeficientes de las ecuaciones se arreglan en una matriz.

En este capítulo vemos la manera de representar matrices o arreglos en Python y algunos ejemplos de cómo realizar operaciones sobre matrices en Python.

7.2 Matrices

Las matrices son arreglos con dimensión mayor a 1, es decir, pueden ser de n renglones y m columnas. En este caso decimos que las matrices son de dos dimensiones. Pero pueden ser de tres dimensiones o más. Un ejemplo de una matriz multidimensional

se presenta en la transmisión de imágenes por televisión de alta definición, donde cada pixel de la imagen tiene dos dimensiones que son la posición del pixel, y otras tres dimensiones son los colores asociados al pixel, además de la intensidad. En este capítulo solamente tratamos con matrices de dos dimensiones. Los arreglos de una dimensión se vieron en el capítulo anterior.

Una matriz es una tabla de datos numéricos, alfanuméricos o booleanos. Los datos se encuentran arreglados en renglones y columnas, como se muestra a continuación:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & & & \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

El arreglo o matriz A tiene m renglones o filas y n columnas.

La dimensión de una matriz es el número de filas $\times$ número de columnas. Se dice que la matriz es de dimensión $n \times m$.

Para referirse a la posición de un elemento de la matriz A primero indicamos el número de renglón seguido del número de columna. De esta manera, el elemento del arreglo A que se encuentra en el p-ésimo renglón y en la q-ésima columna lo referenciamos como:

$$a_{p,q}$$

Una matriz cuadrada de orden n es una matriz de dimensión $n \times n$.

Ejemplo 7.1

Ejemplos de arreglos o matrices de dos dimensiones

Consideremos las matrices A, B, y C dadas por:

$$A = \begin{bmatrix} \text{“}a\text{”} & \text{“}b\text{”} \\ \text{“}c\text{”} & \text{“}d\text{”} \\ \text{“}e\text{”} & \text{“}f\text{”} \end{bmatrix}, \quad B = \begin{bmatrix} 77.3 & 3.1416 \\ 10.01 & -201.3 \\ 66.6 & 27.021 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 2 & 3 \\ 9 & 18 & -5 \\ 16 & 4 & 7 \end{bmatrix}$$

Tenemos que A es una matriz de caracteres alfanuméricos consistente en las seis primeras letras del alfabeto con 3 renglones y 2 columnas por lo que A es una matriz de 3×2. El arreglo B tiene tres renglones o filas y dos columnas. y por lo tanto es una matriz de 3×2 y contiene números reales. Finalmente, C está formada por números enteros y es una matriz cuadrada de 3×3 y se designa como matriz cuadrada de orden 3.

7.3 Arreglos en Python

Las matrices en Python se describen como **listas** de **listas**. Una matriz como:

$$A = \begin{bmatrix} 1 & -2.3 & 0 \\ 9.6 & -1.5 & 4 \\ 5.7 & -6.3 & 3.1 \end{bmatrix}$$

se representa como:

```
A = [ [ 1, -2.3, 0], [ 9.6, -1.5, 4 ], [ 5.7, -6.3, 3.1 ] ]
```

La lista es `A` y tiene tres elementos que a su vez son listas. La primera lista `[1, -2.3, 0]` es el primer elemento (elemento 0) de la lista `A`. La siguiente lista `[ 9.6, -1.5, 4 ]` es el elemento 1 de la lista `A`. Finalmente, la lista `[ 5.7, -6.3, 3.1 ]` es el último elemento de la lista `A`. Por otra parte, cada una de estas listas tiene tres elementos.

Para imprimir los datos de este arreglo llamado `A` que es un arreglo de 3×3 podemos usar dos ciclos **for** anidados de la siguiente manera:

```
for i in range(3): # i es para renglones.
   print( ) # Sirve para salto de renglón.

   for j  in range(3): # j es para columnas
```

```
        print(A[i][j], end = ', ')
```

7.3.1 Generación de arreglos por indexación

Para generar un arreglo, que es una lista de listas, podemos generar un arreglo que tenga el tamaño deseado en blanco y posteriormente llenarlo con los datos requeridos. Para esto usamos un ciclo **for** y hacemos variar el índice, de aquí el nombre de método **por indexación**. Una forma de hacerlo para generar una matriz con elementos `None` es:

1. Generar la lista básica vacía, es decir, que esté llena de `None`. Esta es nuestra matriz que está formada por un renglón vacio sin columnas:

   ```
   A = [ ]
   ```

2. Con un ciclo **for** generar las listas que corresponden a cada uno de los renglones. La longitud de estas listas corresponden al número de columnas m:

   ```
   for i in range(m):
       A.append([ ]) # Se genera un renglón.
       for j in range(n):
           A[i].append(None) # Genera cada uno de los elementos.
   ```

Podemos mostrar la matriz con un ciclo **for** anidado en otro ciclo **for**, de la siguiente manera:

```
for k in range(m): # índice de los renglones.
    print( )  # Salto de renglón.
    for j in range(n): # índice de las columnas.
        print( A[k][j] ,", \t")
```

Ejemplo 7.2

Generación de un arreglo

Supongamos que deseamos generar un arreglo de `None` de cuatro renglones y cuatro columnas. El procedimiento es el siguiente:

	Pseudocódigo	Python
1	**Nombre:**Generación de matriz.	"'Generación de matriz.
2	"'Este algoritmo genera una	Este algoritmo genera una
3	matriz de n × m."'	matriz de n × m. "'
4	**INICIO**	# **INICIO**
5	**Variables:**	
6	**Real:** a, b, c	
7	**Entero:** m, n	
8	**Recibir:** m, n	`m = int(input("Dame m: ")` `n = int(input("Dame n: ")`
9	A ← [ ]	`A = [ ]` # Inicializa la matriz A
10	# Genera los renglones.	# Genera los renglones.
11	**Para** i ← 0 hasta i < m:	`for i in range(m):`
12	A←[[ ]]	`A.append([ ])`
14	**Para** j ← 0 hasta j < m:	`for j in range(m):`
15	A[i][j]← None	`A[i].append([None])`
16	# Desplegado de la matriz	# Desplegado de la matriz
17	**Para** k ← 0 hasta k < m :	`for k in range(m) :`
18	**Mostrar:** # Salto de renglón.	`print ( )` # Salto de renglón.
19	**Para** j ← 0 hasta j < n:	`for j in range(n) :`
20	**Mostrar:** A[k][j] , ",\t "	`print( A[k][j] , ",\t")`
21	**FIN**	# **FIN**

La matriz que se obtiene para m = 4 y n = 4 es:

```
[ [ None, None, None, None ]
  [ None, None, None, None ]
  [ None, None, None, None ]
  [ None, None, None, None ] ]
```

7.3.2 Generación de arreglos por comprensión

El método de comprensión se usa cuando los elementos del arreglo están definidos por una función o por un valor. Básicamente consiste de dos ciclos Para anidados precedidos de la función. El primer ciclo externo es para generar las filas o renglones, mientras que el segundo ciclo genera los elementos de las columnas. Para definir un arreglo se tiene el siguiente formato:

```
A = [ [ f(i) for i in range(columnas)] for j in range(renglones) ]
```

que define un arreglo con el número de renglones y columnas deseado. En esta definición todos los renglones son idénticos, pero el objetivo es definir un arreglo con el tamaño deseado al que posteriormente podemos cambiar el valor de sus elementos. Como ejemplo, la matriz de 4×4 del Ejemplo 7.2 se genera con:

```
A = [ [ None for i in range(4)] for j in range(4)]
```

El resultado es el mismo que teniamos antes para este Ejemplo 7.2.

También podemos incluir una función aleatoria. Por ejemplo:

```
from random import *
A = [[ randint(-10, 10) for i in range(4)] for j in range(4)]
```

que genera la matriz de números enteros:

```
[[10, -8, 7, 9],
  [8, 2, 6, 4],
  [7, 0, -1, 9],
  [3, -2, -3, -8]]
```

Esta matriz es aleatoria y cambia para cada corrida.

Ejemplo 7.3

Creación de un arreglo de 5×3

Supongamos que deseamos formar un arreglo de cinco renglones y tres columnas. Los elementos de cada renglón son iguales y deseamos que sean cuadrados de su posición en el renglón. Esto lo hacemos con:

```
A = [[ i**2 for i in range(3)] for j in range(5)]
```

El resultado es:

```
A = [ [0, 1, 4], [0, 1, 4], [0, 1, 4], [0, 1, 4], [0, 1, 4]]
```

que se puede escribir como:

```
[ [0, 1, 4]
  [0, 1, 4]
  [0, 1, 4]
  [0, 1, 4]
  [0, 1, 4] ]
```

Vemos que se creó un arreglo de 5×3 donde los renglones son idénticos.

7.4 Métodos alternos de escritura de matrices

Existen maneras alternas de desplegar y leer matrices. La primera de ellas usa iteradores de listas, como en el siguiente ejemplo:

Ejemplo 7.4

Uso de iteradores de listas

Un iterador de listas se refiere simplemente al manejo de renglones de una matriz. Por ejemplo, si consideramos la matriz:

$$\texttt{matriz} = \begin{bmatrix} -1 & 0 & -10 \\ 3 & -4 & 0.5 \\ 6 & -23 & 8 \\ 7 & 3 & 9 \end{bmatrix}$$

la podemos escribir en Python como:

```
matriz = [[ -1, 0, -10], [ 3, 4, 0.5 ], [ 6, -23, 8], [ 7, 3, 9]]
```

Para desplegarla usamos un iterador de listas como:

```
for renglon in matriz:
    print( renglon)
```

El resultado es:

```
[ -1,    0,    -10 ]
[ 3,     4,     0.5 ]
[ 6,     -23,     2 ]
[ 7,     3,     9 ]
```

Ejemplo 7.5

Otra forma del uso de iteradores de listas

Otra forma de usar un iterador de listas es el siguiente. Consideramos el mismo arreglo:

```
for renglon in matriz: # Selecciona los renglones.
    print( ) # Salto de renglón para separar renglones.
    for c in renglon:# Selecciona las columnas.
        print(c, end = ', ')# Coma y espacio para separar columnas.
```

En este caso el resultado es:

```
-1,    0,    -10,
3,     4,    0.5,
6,    -23,   8,
7,     3,     9,
```

Notamos la ausencia de corchetes.

7.5 Selección de filas y columnas de un arreglo

Para seleccionar filas o columnas de un arreglo usamos los índices de ciclos **Para**. Con ejemplos presentamos las técnicas para hacerlo.

7.5.1 Filas de un arreglo

Para un arreglo `A` en Python, dado que un arreglo es una lista de listas, donde cada lista es un renglón de la matriz, seleccionar una fila del arreglo es equivalente a seleccionar un elemento de la lista `A`. Para esto usamos simplemente:

`A[k]` donde `k` es el renglón que deseamos seleccionar.

Ejemplo 7.6

Selección de las filas de un arreglo

Supongamos que tenemos el arreglo de cuatro renglones y tres columnas dado por

```
A = [ [9, 12, -44], [16, 7, 14], [23, 61, 84], [-2, -54, 60 ]]
```

que se puede escribir como:

```
[ [ 9, 12, -44 ]
  [ 16, 7, 14 ]
  [ 23, 61, 84 ]
  [ -2, -54, 60 ]]
```

Para seleccionar el tercer renglón que corresponde a `k = 2` usamos:

```
>>> A[2]
   [ 23, 61, 84 ]
```

7.5.2 Columnas de un arreglo

Para un arreglo `A` en Python, los elementos de la `k`-ésima columna corresponden a los elementos correspondientes a la posición `k` de cada una de las sublistas de la lista A. Para seleccionar una columna entonces seleccionamos dichos elementos. Esto es equivalente a seleccionar cada k-ésimo elemento de cada una de las listas que están en la lista `A`. Para esto usamos simplemente un ciclo **for**. Si deseamos obtener los elementos de la `k`-ésima columna usamos:

```
1  k = int( input(" Dame la columna que deseas obtener:\n ") )
2  b = [ ] # Aquí se inicializa la lista donde se almacena la columna.
3  print( )
4  for i in range(len(A)): # Para cada renglón.
5        b.append(A[i][k])# Se selecciona el k-ésimo elemento.

6  print(b)
```

Ejemplo 7.7

Selección de las columnas de un arreglo

Supongamos que tenemos el arreglo de cuatro renglones y tres columnas del ejemplo anterior

```
A = [ [9, 12, -1, 0], [6, 7, 2, 14], [2, 3, 1, 8], [2, 5, -6, 0 ]]
```

y que se desea tener en `b` la segunda columna, es decir, la columna para `k = 1`. Para esto usamos el código anterior para obtener:

```
b = [12, 7, 3, 5 ]
```

7.6 Suma, resta y multiplicación de matrices

Para arreglos existen las operaciones de:

- Suma y resta de matrices de la misma dimensión.
- El producto de un escalar por una matriz.
- El producto de una matriz por otra matriz.

7.6.1 Suma y resta de matrices

La suma y resta de matrices se puede efectuar solamente cuando las matrices **son de la misma dimensión**. Se obtiene sumando o restando los elementos correspondientes de la misma posición. De esta manera, si representamos a la matriz resultante como R y a sus elementos por r_{ij} se tiene que:

$$r_{ij} = a_{ij} \pm b_{ij}$$

Ejemplo 7.8

Suma de matrices.

Supongamos las matrices A y B dadas por:

$$A = \begin{bmatrix} 7 & 3 \\ 2 & -6 \\ 6 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 \\ 9 & -5 \\ -4 & 7 \end{bmatrix}$$

Entonces la suma de A y B está dada por:

$$A + B = \begin{bmatrix} 7 & 3 \\ 2 & -6 \\ 6 & 7 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 9 & -5 \\ -4 & 7 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 11 & -11 \\ 2 & 14 \end{bmatrix}$$

y la resta de A y B está dada por:

$$A - B = \begin{bmatrix} 7 & 3 \\ 2 & -6 \\ 6 & 7 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 9 & -5 \\ -4 & 7 \end{bmatrix} = \begin{bmatrix} 6 & 1 \\ -7 & -1 \\ 10 & 0 \end{bmatrix}$$

Vemos que las dos matrices son de 3×2 y el resultado en ambos casos es una matriz de 3×2.

7.6.2 Suma y resta de matrices en Python

En Python realizamos la misma secuencia de instrucciones que en pseudocódigo. Solamente es necesario haber definido la matriz resultado previamente. Un algoritmo en Python que realiza la suma es, para matrices `n` × `m`:

```
# Inicializa la matriz resultado de dimensión n × m.
# La matriz c debe tener n renglones de m ceros.
c = [[0,0,...,0],[0,0,...,0],...,[0,0,...,0]]
for k in range(len(a)):                # Itera los renglones.
    for j in range(len(a[0])):         # Itera las columnas.
            c[k][j] = a[k][j] + b[k][j]
print(c)
```

Ejemplo 7.9

Suma de matrices en Python

Supongamos las matrices `A` y `B` del Ejemplo 7.8:

$$A = \begin{bmatrix} 7 & 3 \\ 2 & -6 \\ 6 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 \\ 9 & -5 \\ -4 & 7 \end{bmatrix}$$

Entonces, `A` y `B` en Python están dadas por:

```
A = [ [7, 3], [2, -6], [6, 7] ]

B = [ [1, 2], [9, -5], [-4, 7] ]
```

```
1  # Inicializa la matriz resultado de dimensión 3 × 2.
2  C = [ [0,0],[0,0],[0,0] ]          # Genera la matriz resultado.
3  for k in range(len(A)):            # Itera los renglones.
4      for j in range(len(A[0])):     # Itera las columnas.
5              C[k][j] = A[k][j] + B[k][j]

6  print(C)                           # Despliega el resultado.
```

El resultado es idéntico al del Ejemplo 7.8:

```
C = [ [8, 5], [11, -11], [2, 14] ]
```

que es equivalente a:

```
C = [ [8, 5],
      [11, -11],
      [2, 14] ]
```

Ejemplo 7.10

Suma de arreglos

Para las matrices:

```
a = [ [3, 2, 1, 9],
      [-1, -3, 0, 8],
      [5, 7, 8, 7] ]

b = [ [9, 0, 1, 3],
      [-1, 7, 1, 4],
      [6, 4, 1, 6] ]
```

La suma se obtiene con:

```
1  a = [ [3, 2, 1, 9], [-1, -3, 0, 8], [5, 7, 8, 7] ]
2  b = [ [9, 0, 1, 3], [-1, 7, 1, 4], [6, 4, 1, 6] ]
3  # Inicializa la matriz resultado de dimensión 3 × 4.
4  c = [ [0,0, 0, 0],[0,0, 0, 0],[0,0, 0,0] ]  # Matriz resultado.
5  for k in range(len(a)):                # Itera los renglones.
6      for j in range(len(a[0])):         # Itera las columnas.
7              c[k][j] = a[k][j] + b[k][j] # Realiza la suma.

8  print(c) # Despliega el resultado.
```

El resultado es:

```
c = [ [12, 2, 2, 12], [-2, 4, 1, 12], [11, 11, 9, 13] ]
```

que se puede escribir como:

```
c = [ [12, 2, 2, 12],
      [-2, 4, 1, 12],
      [11, 11, 9, 13] ]
```

7.6.3 Multiplicación de matriz por un escalar

La multiplicación de una matriz A por un escalar k produce una matriz donde cada elemento se multiplica por el escalar. La matriz resultante es de la misma dimensión. Si representamos a la matriz resultante como R y a sus elementos por r_{ij} se tiene que:

$$r_{ij} = k * a_{ij}$$

Ejemplo 7.11

Multiplicación de matriz por escalar

Supongamos la matriz A dada por:

$$A = \begin{bmatrix} 2 & -1 & 7 & 3 \\ 12 & -26 & 4 & -8 \\ -4 & -5 & 6 & 7 \end{bmatrix}$$

Entonces, si multiplicamos A por el escalar k obtenemos:

$$k * A = \begin{bmatrix} 2k & -k & 7k & 3k \\ 12k & -26k & 4k & -8k \\ -4k & -5k & 6k & 7k \end{bmatrix}$$

Cuando la matriz A la representamos en Python estamos tentados a intentar realizar directamente

$k*A$

pero como la matriz es una lista, la operación $k*A$ indica la repetición de la lista tantas veces como lo indica el factor k. Por lo tanto la multiplicación la tenemos que usar con ciclos anidados de la siguiente manera:

```
R = [[None for i in range(NoColumnas)] for j in range(NoRenglones)]
   for renglones in range(NoRenglones):
        for columnas in range(NoColumnas):
             R[renglones][columnas] = k*A[renglones][columnas]
```

Para `k` = 5 y la matriz `A` tenemos el siguiente algoritmo:

```
1  # Matriz a multiplicar.
2  A = [ [2, -1, 7, 3], [ 12, -26, 4, -8], [ -4, -5, 6 , 7 ] ]
3  k = 5
4  # Matriz resultado.
5  R = [[None for i in range(4)]for j in range(3)]
6  NoRenglones = len(A)
7  NoColumnas = len(A[1])

8  # Se inicia la multiplicación.
9  for columnas in range( NoRenglones ) ) :
10    for columnas in range( NoColumnas ) :
11       R[renglones][columnas] = k*A[renglones][columnas]

12 # Impresión del resultado
13 for r in range(NoRenglones):
14    print( ) # Salto de renglón.
15    for c in range(NoColumnas) :
16       print(R[r][c], end = ' ')
```

El resultado que obtenemos es:

```
10 -5 35 15
60 -130 20 -40
-20 -25 30 35
```

que es el resultado esperado.

7.6.4 Multiplicación de matriz por una matriz

La multiplicación de una matriz A de dimensión $n \times m$ por otra matriz B de dimensión $j \times k$ requiere que las dimensiones de las matrices sean tales que $m = j$. Esto quiere decir que el número de columnas de la matriz A sea igual al números de renglones de la matriz B. **El resultado es una matriz de orden** $n \times k$.

Supongamos que tenemos las matrices:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & & & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix}, \quad B = \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,k-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,k-1} \\ \vdots & & & \\ b_{m-1,0} & b_{m-1,1} & \cdots & b_{m-1,k-1} \end{bmatrix}$$

Los elementos de la matriz $C = A*B = AB$ están dados por:

$$c_{ij} = \sum_{l=0}^{m-1} a_{il} b_{lj} = a_{i0} b_{0j} + a_{i1} b_{1j} + \cdots + a_{i,m-1} b_{m-1,j}$$

donde los índices i, j varían de

$$i = 0, 1, 2, \ldots, n-1 \qquad j = 0, 1, 2, \ldots, k-1$$

Esto es equivalente a decir que el elemento c_{ij} se obtiene multiplicando los elementos de la fila i de la matriz A por los correspondientes elementos de la columna j de la matriz B y sumando los productos. Lo único que debemos saber para poder realizar el producto de dos matrices es que el **número de columnas de la primera matriz sea igual al número de renglones de la segunda matriz**.

Ejemplo 7.12

Multiplicación de matrices

Supongamos las matrices A y B dadas por:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} -7 & -8 \\ 9 & 10 \\ 4 & -11 \end{bmatrix}$$

Vemos que A es de orden 2×3 y B es de orden 3×2. Entonces podemos multiplicar AB para obtener una matriz de 2×2 y también podemos multiplicar BA que resulta en una matriz 3×3. Esto lo hacemos como:

$$AB = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} -7 & -8 \\ 9 & 10 \\ 4 & -11 \end{bmatrix}$$

$$= \begin{bmatrix} 1*(-7)+2*9+3*4 & 1*(-8)+2*10+3*(-11) \\ 4*(-7)+5*9+6*4 & 4*(-8)+5*10+6*(-11) \end{bmatrix}$$

$$= \begin{bmatrix} 23 & -21 \\ 41 & -48 \end{bmatrix}$$

Que es una matriz de 2×2. Por otro lado:

$$BA = \begin{bmatrix} -7 & -8 \\ 9 & 10 \\ 4 & -11 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$= \begin{bmatrix} (-7)*1+(-8)*4 & (-7)*2+(-8)*5 & (-7)*3+(-8)*6 \\ 9(1)+10*4 & 9*2+10*5 & 9*3+10*6 \\ 4*1+(-11)*4 & 4*2+(-11)*5 & 4*3+(-11)*6 \end{bmatrix}$$

$$= \begin{bmatrix} -39 & -54 & -69 \\ 49 & 58 & 87 \\ -40 & -47 & -54 \end{bmatrix}$$

que es una matriz 3×3.

7.6.5 Producto de matrices en Python

El producto de matrices en Python se realiza de la misma manera que se describe en la sección anterior. Por conveniencia repetimos la ecuación que describe los elementos de la matriz producto $C = A*B = AB$ los cuales están dados por:

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{i,m-1} b_{m-1,j}$$

donde los índices i, j varían de

$$i = 0, 1, 2, \ldots, n \qquad j = 0, 1, 2, \ldots, k$$

Un programa que realiza esto para dos matrices a y b de 3×3 es es:

```
1  c = [[0,0,0],[0,0,0],[0,0,0]]# Inicializa la matriz resultado.
2  for k in range(len(a)):
3      for j in range(len(b[0])):
4          for t in range(len(b)): # Obtiene la sumatoria.
5              c[k][j] += a[k][t]*b[t][j]

6  # Despliega el resultado.
7  print(c)
```

Ejemplo 7.13

Producto de arreglos

Para las matrices:

```
a = [[3, 2, 1],[-1, -3, 0],[5, 7, 8]]
b = [[9, 0, 1],[-1, 7, 1], [6, 4, 1]]
```

se usa el siguiente programa:

```
a = [[3, 2, 1],[-1, -3, 0],[5, 7, 8]]
b = [[9, 0, 1],[-1, 7, 1], [6, 4, 1]]
c = [[0,0,0],[0,0,0],[0,0,0]]# Inicializa la matriz resultado.
for k in range(len(a)):
    for j in range(len(b[0])):
        for t in range(len(b)): # Obtiene la sumatoria.
            c[k][j] += a[k][t]*b[t][j]
print(c)
```

El resultado es:

```
c = [ [31, 18, 6], [-6, -21, -4], [86, 81, 20] ]
```

Que se puede escribir como:

```
[ [31, 18, 6],
  [-6, -21, -4],
  [86, 81, 20] ]
```

7.7 Matrices especiales

Existen algunas matrices que tienen características que las hacen distintas de otras matrices y que merecen una atención especial. Entre estas matrices especiales tenemos la matriz identidad, la transpuesta, la inversa, la triangular, la matriz diagonal, entre otras. Estas matrices las presentamos en esta sección.

7.7.1 La matriz identidad

La matriz identidad es aquella matriz que tiene 1's en la diagonal principal y 0's fuera de la diagonal principal. Esta matriz se representa con la letra I:

$$I = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

7.7.2 La matriz identidad en Python

Para generar la matriz identidad generamos una matriz del tamaño deseado pero que solamente tenga ceros y luego hacemos los elementos de la diagonal principal iguales a la unidad.

Los elementos de la diagonal principal son aquellos donde los índices son iguales, es decir, aquellos elementos que son `a[k][k]`.

El siguiente ejemplo obtiene la matriz identidad de dimensión 4×4.

Ejemplo 7.14

Matriz identidad de 4×4

La matriz identidad de 4×4 se genera de la siguiente manera:

```
# Se genera la matriz de ceros.
I = [[0 for k in range(4)] for j in range(4)]

# Los elementos de la diagonal principal se hacen la unidad.
for k in range(4):
      I[k][k] = 1

# Para desplegar la matriz identidad.
for k in range(4):
      print( ) # Para salto de renglón.
      for j in range(4):
            print(I[k][j], end= '', '')
```

El resultado que se obtiene es:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

que es la matriz identidad de dimensión 4×4.

7.7.3 La matriz transpuesta

Una matriz B es la transpuesta de la matriz A si las filas de A se usan como las columnas de B y las columnas de A como filas de B. De esta manera, si la matriz A tiene m filas y n columnas, la matriz transpuesta tiene n filas y m columnas. Para A dada por:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & & & \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

entonces, la matriz transpuesta, que se denota por $A^T = A'$, con n filas y m columnas está dada por:

$$A^T = A' = \begin{bmatrix} a_{0,0} & a_{1,0} & \cdots & a_{m-1,0} \\ a_{0,1} & a_{1,1} & \cdots & a_{m-1,1} \\ \vdots & & & \\ a_{0,n-1} & a_{1,n-1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

Otra manera de expresar la transpuesta de A es expresarla en términos de sus elementos a_{ij} como:

$$A = [a_{ij}] \qquad \text{Matriz original}$$

$$A^T = [a_{ji}] \qquad \text{Matriz transpuesta}$$

Como ejemplo, si tenemos

$$A = \begin{bmatrix} -1 & 4 \\ 8 & 3 \\ 6 & 9 \\ 2 & -7 \end{bmatrix} \qquad \text{Matriz original}$$

La transpuesta es:

$$A^T = \begin{bmatrix} -1 & 8 & 6 & 2 \\ 4 & 3 & 9 & -7 \end{bmatrix} \qquad \text{Matriz transpuesta}$$

7.7.4 Transpuesta de un arreglo en Python

Para obtener la transpuesta A^T de un arreglo A, en otras palabras, transponer un arreglo, usamos la definición de transpuesta que es, en notación de Python:

```
A^T[ j ][ k ] = A[ k ][ j ]
```

Una manera de hacerlo es usando los índices. El procedimiento para obtener la transpuesta de una matriz A de dimensión $n \times m$ se detalla a continuación:

1. Primero se genera un arreglo vacío con las m filas y n columnas requeridas en la matriz transpuesta:

2.
```
A_trans = [ [None, None, ..., None],
            [None, None, ..., None],
                  ⋮
            [None, None, ..., None] ]
```

3. Los valores de `A_trans` están dados por un ciclo como:

```
for j in range (m):
    for k in range(n):
        a_trans[j][k] = a[k][j]
```

4. Finalmente mostramos la matriz transpuesta:

```
for j in range (m):
    print( )
    for k in range(n):
        print(a_trans[j][k] = a [k][j], end = ", ")
```

El programa completo para una matriz de dimensión $n \times m$ es:

```
# Matriz resultado.
A_trans = [ [None, None, ..., None],
            [None, None, ..., None],
            [None, None, ..., None] ]

# Se genera la matriz transpuesta:
for j in range (m):
     for k in range(n):
          a_trans[j][k] = a [k][j]

# Se muestra la matriz transpuesta:
for j in range (m):
     print( )
     for k in range(n):
          print(a_trans[j][k] = a [k][j], end = ", " )
```

Ejemplo 7.15

Transpuesta de un arreglo

Si tenemos el arreglo :

```
A = [ [-1, 0, -10], [3, 4, 0.5], [6, -23, 8], [7, 3, 9] ]
```

La matriz `A` es de 4×3, la matriz transpuesta `A_trans` entonces es de 3×4 y se obtiene con:

```
# Se genera la matriz del tamaño requerido.
A_trans = [ [None, None, None, None],
            [None, None, None, None],
            [None, None, None, None] ]

# Se genera la transpuesta.
for j in range ( len(A) ):
     for k in range( len( A[0] ) ):
          A_trans[k][j] = A [j][k]

# Se despliega la matriz transpuesta.
for j in range (m):
     print( )
     for k in range(n):
          print(A_trans[j][k], end = ", " )
```

El resultado es:

-1, 3, 6, 7,
0, 4, -23, 3,
-10, 0.5, 8, 9,

que se puede reescribir como:

```
[ [-1, 3, 6, 7],
[ 0, 4, -23, 3],
[-10, 0.5, 8, 9] ]
```

que es la transpuesta de `A`.

7.7.5 Obtención de la transpuesta por comprensión

Otra manera de hacerlo es usando la siguiente instrucción por comprensión:

```
[[ renglon[i] for renglon in A] for i in range(No_columnas_de_A)]
```

Ejemplo 7.16

Transpuesta por comprensión de un arreglo

Si tenemos el arreglo :

```
A = [ [-1, 0, -10], [3, 4, 0.5], [6, -23, 8], [7, 3, 9] ]
```

La transpuesta se obtiene con:

```
>>> A_trans = [[renglon[i] for renglon in A] for i in range(len(A[0]))]
[[-1, 3, 6, 7], [0, 4, -23, 3], [-10, 0.5, 8, 9]]
```

que se puede reescribir como:

```
[ [-1, 3, 6, 7],
[ 0, 4, -23, 3],
[-10, 0.5, 8, 9] ]
```

que es la transpuesta de `A`.

7.7.6 La matriz simétrica

La matriz simétrica es aquella matriz que satisface que:

$$a_{jk} = a_{kj}$$

La matriz simétrica debe tener el mismo número de filas que de columnas, entonces una matriz simétrica debe ser una matriz cuadrada.

La matriz A es simétrica si es de la forma

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,n-1} \\ a_{0,1} & a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} \\ a_{0,2} & a_{1,2} & a_{2,2} & \cdots & a_{2,n-1} \\ \vdots & & & & \\ a_{0,n-1} & a_{1,n-1} & a_{2,n-1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

Por ejemplo, A dada por:

$$A = \begin{bmatrix} -1 & 8 & 6 & 2 \\ 8 & 3 & 1 & -7 \\ 6 & 1 & 8 & 9 \\ 2 & -7 & 9 & -6 \end{bmatrix}$$

es una matriz simétrica.

Ejemplo 7.17

Generación de una matriz simétrica 4×4

Podemos generar un arreglo simétrico usando números aleatorios de la siguiente manera:

1. Primero generamos la matriz del tamaño deseado, por ejemplo, por comprensión:

```
A_simetrica = [[None for k in range(4)] for j in range(4)]
```

2. Ahora generamos los elementos aleatorios `a[i][j]` tales que satisfagan `a[i][j] = a[j][i]`.

Un algoritmo que realiza esto, para una matriz de enteros aleatorios es:

```
import random
n = int(input("Dame la dimensión de la matriz: "))

# Se genera una matriz de n x n de enteros aleatorios.
A = [[None for k in range(n)] for j in range(n)]
for k in range(n):
     for j in range(n):
          A[k][j] = random.randint(0, 10)

# La matriz se hace simétrica.
for k in range(n):
     for j in range(k):
          A[j][k] = A[k][j]

# Se despliega la matriz.
for renglon in A:
     print(renglon)
```

El programa produce el siguiente resultado:

```
Dame la dimensión de la matriz: 5

[8, 3, 7, 5, 3]
[3, 10, 10, 10, 5]
[7, 10, 8, 3, 3]
[5, 10, 3, 3, 4]
[3, 5, 3, 4, 3]
```

que corresponde a una matriz simétrica. Dado que es una matriz de números aleatorios, cada corrida de este algoritmo producirá un resultado diferente.

7.8 Ejemplos

Ejemplo 7.18

Matriz de ventas departamentales

Consideremos las ventas por departamento de una tienda durante doces meses. Se tienen cinco departamentos en la tienda. Para almacenar las ventas por departamento necesitamos almacenarlas en un vector de 12 elementos como se muestra a continuación con un ejemplo numérico:

$$\textbf{Ventas por mes} \rightarrow \begin{bmatrix} 123.00 \\ 234.00 \\ 117.00 \\ 439.00 \\ 132.00 \\ 156.00 \\ 341.00 \\ 221.00 \\ 543.00 \\ 698.00 \\ 455.00 \\ 376.00 \end{bmatrix}$$

Ahora para cada uno de los cinco departamentos en esta tienda departamental tenemos un vector. Eso quiere decir que debemos manejar 5 vectores, uno por departamento, para llevar las ventas de cada uno de ellos por cada mes. La desventaja de usar este enfoque es que cada departamento se maneja de manera separada y es muy posible que los administradores quieran ver todas las ventas y compararlas en un algoritmo que despliegue las ventas de los cinco departamentos durante los 12 meses del año. Un ejemplo de estas ventas se muestra a continuación:

Mes	Depto. 1	Depto. 2	Depto. 3	Depto. 4	Depto. 5
enero	123.00	13.00	87.00	217.00	567.00
febrero	234.00	17.00	95.00	199.00	654.00
marzo	117.00	12.00	67.00	321.00	986.00
abril	439.00	23.00	34.00	289.00	677.00
mayo	132.00	18.50	45.00	333.00	456.00
junio	156.00	29.00	39.00	231.00	725.00
julio	341.00	11.00	51.00	111.00	803.00
agosto	221.00	19.00	57.00	201.00	755.00
septiembre	543.00	25.00	88.00	117.00	622.00
octubre	698.00	12.45	63.00	669.00	681.00
noviembre	455.00	4.00	43.00	99.00	349.00
diciembre	376.00	43.00	26.00	387.00	993.00

Estas ventas son en miles de pesos. Si ahora quisieramos ver las ventas de mes de marzo para todos los departamentos, podemos desplegar para cada vector el tercer elemento. Y lo mismo podemos hacer para cualquier otra operación que querramos hacer para cualquiera de los meses. Sin embargo, otra forma de presentar la información de las ventas de todos los meses para todos los departamentos es en forma de una matriz. Los 5 vectores anteriores se pueden arreglar en una matriz. La matriz correspondiente es:

	D. 1	D. 2	D. 3	D. 4	D. 5
enero	123.00	13.00	87.00	217.00	567.00
febrero	234.00	17.00	95.00	199.00	654.00
marzo	117.00	12.00	67.00	321.00	986.00
abril	439.00	23.00	34.00	289.00	677.00
mayo	132.00	18.50	45.00	333.00	456.00
junio	156.00	29.00	39.00	231.00	725.00
julio	341.00	11.00	51.00	111.00	803.00
agosto	221.00	19.00	57.00	201.00	755.00
septiembre	543.00	25.00	88.00	117.00	622.00
octubre	698.00	12.45	63.00	669.00	681.00
noviembre	455.00	4.00	43.00	99.00	349.00
diciembre	376.00	43.00	26.00	387.00	993.00

Esta matriz se representa en pseudocódigo como **ventas[12][5]**. Esta notación significa que la matriz con el nombre `ventas` tiene 12 renglones y 5 columnas. Para leer los elementos de un arreglo de dos dimensiones necesitamos dos ciclos anidados. Por ejemplo para leer los `ventas` de los departamentos podemos usar:

Para depto ← 0 a 4 :
    Para mes ← 0 a 11 :
        Mostrar: "Dame los gastos del mes ", `mes`, "del depto ", `depto`
        Recibe: `ventas[ depto ][ mes ]`
    FinPara
FinPara

El par de ciclos anidados empieza con el primer ciclo **Para** que tiene el valor de `depto` ← 0 y de ahí se pasa al ciclo **Para** anidado donde se indica leer los valores de gastos de cada mes para el depto 0. De ahí se va otra vez al **Para** exterior para cambiar al depto 1 y leer en el **Para** anidado los gastos del depto 1 de los 12 meses del año. Seguimos en esta manera hasta terminar de leer los gastos de los 5 deptos. Notamos que para leer los elementos de la matriz leemos los datos de todos los meses para un departamento y así continuamos hasta terminar todos los departamentos, es decir, hemos leído los elementos de la matriz por renglones. Alternativamente, podemos leer los elementos de la matriz leyendo columna a columna. En cada caso leemos los gastos de cada departamento para el mes de enero (mes = 0), luego leemos los gastos de cada departamento para el mes de febrero (mes = 1) y así continuamos hasta terminar leyendo los gastos de cada departamento para el mes de diciembre (mes = 11). En pseudocódigo podemos ver que esto se hace con:

Para mes ← 0 a 11 :
    Para depto ← 0 a 4 :
        Mostrar: "Dame los gastos del mes ", mes, "del depto ", depto;
        Recibe ventas[depto][mes]
    FinPara
FinPara

Para leer los gastos, que son tipo flotante, de los departamentos usamos en Python:

```
for depto in range(5):
   for mes in range(12):
      gasto[depto][mes] = float(input("Dame los gastos del mes:\n"))
```

Ahora que ya hemos leído los datos de la matriz necesitamos efectuar las operaciones necesarias. En este ejemplo vamos a requerir tres resultados:

- El primero es los gastos anuales de cada departamento.
- El segundo los gastos de cada mes de todos los departamentos.
- Finalmente, podemos calcular el gasto total anual.

Los dos primeros resultados los deseamos almacenar en dos vectores. Para los gastos anuales de cada departamento el vector tiene 5 elementos ya que ese es el número de departamentos. El gasto mensual requiere un vector de 12 elementos, uno para el gasto de cada departamento. El gasto total solamente requiere una variable escalar real. Los vectores y la variable escalar se inicializan a cero antes de empezar a sumar:

```
gasto_mensual = [ 0 ]*12
gasto_depto = [ 0 ]*5
gasto_total = 0
```

La suma mensual la podemos realizar con dos ciclos `for` anidados:

```
for mes in range(12):
   for depto in range(5):
       gasto_depto[depto] = gasto_depto[depto] + gasto[depto][mes]
```

Ahora, para calcular el gasto mensual sumamos de la siguiente manera:

```
for depto in range(5):
     for mes in range(12):
         gasto_mensual[mes] = gasto_mensual[mes] + gasto[depto][mes]
```

Lo que nos falta es obtener el gasto total de todo el año para todos los departamentos. Esto lo podemos hacer de dos maneras. Sumando los elementos del gasto mensual o sumando los elementos del gasto por departamento. Ambas opciones nos deben dar el mismo resultado.

```
for mes in range(12):
   gasto_total = gasto_total + gasto_mensual[mes]
print(gasto_total)
```

```
gasto_total = 0 # Reinicializamos gasto total.
for depto in range(5):
    gasto_total = gasto_total + gasto_depto[depto]
print(gasto_total)
```

En ambos casos el resultado es 16198.

Ejemplo 7.19

Tiros de dados

Tres amigos tienen un dado de 6 caras. El dado se tira 5 veces por cada amigo. Los tiros se almacenan en un arreglo de 3×6 donde la última columna es el total de puntos. Por ejemplo:

Nombre	Tiro 1	Tiro 2	Tiro 3	Tiro 4	Tiro 5	Puntos totales
Juan	6	3	4	4	3	
Rodrigo	4	5	6	4	5	
Manuel	2	4	1	6	2	

Escriba un programa en Python que realice las siguientes acciones:

1. Genere la matriz de 3×6 y llene de manera aleatoria las posiciones de la matriz correspondientes a los tiros de los 3 jugadores.
2. Encuentre los puntos totales de cada jugador (suma de los tiros).
3. Encuentre las veces que cada jugador obtuvo 6.
4. Muestre los resultados obtenidos.

Un posible algoritmo en Python se muestra a continuación:

Dado que se llenará la matriz de manera aleatoria, se debe importar la biblioteca random:

```
import random
```

Se inicializa la lista que se usa como matriz, así como el máximo y el mínimo de cada tiro:

```
tabla = [ ]
min = 1
max = 6
```

Se empieza a llenar la matriz con los tiros:

```
for r in range (3) :
   tabla.append([ ])
   for t in range (5):
      tabla[r].append(random.randint(min, max))
```

Se despliega la tabla de tiros:

```
for k in range(3):
   print(tabla[k])
```

Ahora se inicializa el valor de la suma y se calcula:

```
for r in range (3):
   tabla [r].append(0)
   for t in range (5):
      tabla [r][5] = tabla [r][5] + tabla [r][t]
```

Ahora se tiene la sexta columna de la matriz con la suma de los tiros. El siguiente paso es contar el número de veces que se repite el tiro 6 para cada jugador. Esto lo realizamos checando cada tiro y contando las veces que el 6 ocurre. Para contar usamos el vector **vez** inicializado a 0 para cada jugador.

```
vez = [ 0, 0, 0 ] # Inicialización del vector.
for r in range (3) :
   for t in range(5) :
      if tabla[r][t] == 6 :  # Se compara el tiro con el 6.
         vez[r] = vez[r] + 1  # Si el 6 ocurre se incrementa la cuenta.
```

Se imprime la cuenta del número de tiros con 6 para cada jugador y la suma de puntos.

Hay tres opciones para el caso de obtener un 6, dos o más 6's, o ningún 6.

```
for r in range (3):
    if(vez[r] == 1):
        print("El jugador ", r + 1, " obtuvo 6 ", vez[r], " vez." )
    elif(vez[r] >= 2):
        print("El jugador ", r + 1, " obtuvo 6 ", vez[r], " veces." )
    elif(vez[r] == 0):
        print("El jugador ", r + 1, " NO obtuvo 6. " )
    print("Su suma de puntos fue: ", tabla[r][5], ".")
    print()
```

7.9 Conclusiones

Las matrices encuentran aplicación en muchas áreas de la ciencia, de las matemáticas, de la economía, de las finanzas y de la vida cotidiana. En este capítulo hemos presentado una introducción al uso de matrices usando Python. Varias operaciones con matrices se han implementado en Python. En el capítulo 13 regresaremos a usar matrices pero en dicho capítulo estaremos usando una biblioteca llamada numpy que permite un manejo más sencillo de las matrices. Sin embargo, el conocimiento adquirido en este capítulo nos permite adquirir un conocimiento acerca de como maneja Python los arreglos o matrices.

7.10 Ejercicios

1. Se tiene una lista de datos de jugadores de futbol. Para cada jugador la lista contiene:
    - Cadena con el nombre,
    - Edad,

- Altura,
- Peso,
- Posición.

Escriba un programa que lea los datos de 10 jugadores. El programa debe ser capaz de seleccionar los jugadores que a) Pasen de una edad dada por el usuario, b) Pasen de una cierta altura, c) Tengan un peso mayor a un valor dado, d) Jueguen cierta posición.

Use arreglos para la solución.

2. Para las matrices `A` y `B` mostradas, calcule `AB = A*B` y `BA = B*A`.

$$\mathtt{A} = \begin{bmatrix} 1 & 0 & 2 \\ 2 & 3 & -1 \\ -1 & 0 & 3 \end{bmatrix}$$

$$\mathtt{B} = \begin{bmatrix} 1 & 2 & 3 \\ 7 & 4 & -2 \\ 6 & -10 & -5 \end{bmatrix}$$

3. Para las matrices del Ejercicio anterior calcule `A + B` y `A - B`.

4. Se dice que la matriz `B` es la inversa de la matriz `A` si se cumple que `AB = I`. Para la matriz A dada encuentre cual de las matrices B o C es la inversa de A realizando el producto de AB y BA.

$$\mathtt{A} = \begin{bmatrix} 4 & 8 & 2 \\ 2 & -2 & -1 \\ 6 & 7 & 3 \end{bmatrix}$$

$$\mathtt{B} = \begin{bmatrix} -1 & 10 & 4 \\ 12 & 0 & -8 \\ -26 & 20 & 24 \end{bmatrix}$$

$$\mathtt{C} = \begin{bmatrix} -1 & 10 & 4 \\ 12 & 0 & -8 \\ -26 & -20 & 24 \end{bmatrix}$$

5. Para las matrices del ejercicio anterior demuestre que `BA = I`.

6. En el Ejercicio anterior calcule `AB` y `CA`. Explique sus resultados.

7. Obtenga el producto de `A*b`.

$$\mathtt{A} = \begin{bmatrix} 2 & 1 & 0 \\ 9 & -4 & 11 \\ 16 & 5 & -8 \end{bmatrix}$$

$$\mathtt{b} = \begin{bmatrix} 10 \\ -3 \\ 17 \end{bmatrix}$$

8. Encuentre la transpuesta de `A` y de `b` dados en el Ejercicio anterior.

9. De los resultados del Ejercicio anterior calcule `b'*A`.

10. El juego de memoria consiste en seleccionar dos cartas iguales de un conjunto de cartas volteadas hacia abajo. Una manera de colocar las cartas volteadas es en forma de arreglo. En el juego participan dos jugadores seleccionando dos cartas a la vez. Si las cartas son iguales se asocian con ese jugador. Si son diferentes se regresan. El juego termina cuando se han adivinado todos los pares. Se requiere diseñar un algoritmo que realice el juego de memoria. Use 5 pares de cartas con nombres de animales. Use una matriz de 5×2.

11. Mejore el juego de memoria para que se permita jugar en tres niveles: novato con una matriz 2×5, intermedio con una matriz 4×4, y avanzado con una matriz 4×6.

12. Una matriz diagonal tiene ceros en todos los elementos menos los de la diagonal principal, es decir, es de la forma:

$$D = \begin{bmatrix} d_{0,0} & 0 & 0 & \dots & 0 \\ 0 & d_{1,1} & 0 & \dots & 0 \\ 0 & 0 & d_{2,2} & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & d_{n-1,n-1} \end{bmatrix}$$

Diseñe un algoritmo que genere una matriz diagonal con los elementos de la diagonal principal enteros aleatorios.

13. Una matriz triangular tiene elementos iguales a cero arriba o abajo de la diagonal principal. Todos los demás elementos, incluidos los de la diagonal principal pueden ser diferentes de cero. Si la matriz tiene los elementos abajo de la diagonal principal iguales a cero decimos que es triangular superior, como en:

$$U = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \dots & a_{0,n-1} \\ 0 & a_{1,1} & a_{1,2} & \dots & a_{1,n-1} \\ 0 & 0 & a_{2,2} & \dots & a_{2,n-1} \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & a_{m-1,n-1} \end{bmatrix}$$

Pero si los elementos arriba de la diagonal principal son iguales a cero decimos que la matriz es triangular inferior:

$$L = \begin{bmatrix} a_{0,0} & 0 & 0 & \dots & 0 \\ a_{1,0} & a_{1,1} & 0 & \dots & 0 \\ a_{2,0} & a_{2,1} & a_{2,2} & \dots & 0 \\ \vdots & & & & \\ a_{m-2,0} & a_{m-2,1} & a_{m-2,2} & \dots & 0 \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \dots & a_{m-1,n-1} \end{bmatrix}$$

Escriba un algoritmo para generar matrices triangulares. El algoritmo debe preguntar si la matriz es triangular superior o inferior.

14. La traza de una matriz cuadrada es la suma de los elementos de la diagonal principal. Los elementos de la diagonal principal son aquellos cuyos índices son iguales, es decir, son de la forma:

```
A [k] [k]
```

Entonces la traza está dada por:

$$\mathtt{A} = \sum_{k=0}^{n-1} \mathtt{A[k][k]}$$

Escriba e implemente en Python un algoritmo para calcular la traza de una matriz.

Capítulo 8
Subalgoritmos

8.1 Introducción
8.2 Subalgoritmos
8.3 Funciones
8.4 Funciones en Python
8.5 Procedimientos
8.6 Funciones lambda
8.7 Llamado por valor y llamado por referencia
8.8 Variables locales y globales
8.9 Ejemplos adicionales
8.10 Instrucciones de Python del Capítulo 8
8.11 Conclusiones
8.12 Ejercicios

El todo es más que la suma de las partes.

Aristóteles

Objetivos

Los subalgoritmos nos permiten escribir un algoritmo en módulos que realicen una tarea específica dentro del algoritmo principal. Los subalgoritmos se implementan en Python por medio de funciones y procedimientos.

8.1 Introducción

Hasta este momento hemos desarrollado algoritmos pequeños e ilustrativos de las instrucciones presentadas. Pero en general, un programa o una aplicación de uso comercial o académico puede tener miles de líneas. En este algoritmo pueden existir secciones de código que se repiten varias veces y en otras ocasiones la aplicación es demasiado larga y complicada. En cualquiera de estos casos es conveniente dividir el algoritmo en secciones más pequeñas que permitan simplificar y hacer más clara la operación del algoritmo. Con este fin se usan los subalgoritmos que son porciones del algoritmo que se pueden volver a usar en el desarrollo de una aplicación, o simplemente realizan porciones del algoritmo para hacer más fácil el desarrollo de la aplicación. En este capítulo se presentan dos tipos de subalgoritmos: funciones y procedimientos. Una función recibe datos, los procesa y regresa otros datos. Por otra parte, un procedimiento recibe datos, los procesa y no necesariamente regresa datos. A través de ejemplos mostraremos la manera de implementarlos en Python.

8.2 Subalgoritmos

Los subalgoritmos en general tienen datos de entrada y proporcionan como salida otros datos. Ejemplos de subalgoritmos lo constituyen las funciones trigonométricas como $y = \text{sen } x$, la raíz cuadrada $y = \sqrt{x}$, entre otras, las cuales tienen un dato de entrada que es x y regresan un dato de salida y.

Ejemplo 8.1

Ejemplo de algoritmo para construir una casa

Un algoritmo para construir una casa se puede visualizar de la siguiente manera: El algoritmo principal es el arquitecto que proyecta la construcción y delega a otros la construcción de cimientos y paredes, las ventanas, las puertas, etc. De esta manera el algoritmo se divide en varios subalgoritmos. El primer subalgoritmo construye cimientos, paredes y techos, el segundo subalgoritmo coloca las ventanas, otro subalgoritmo fabrica y coloca puertas, etc. Estos subalgoritmos son "invocados" desde el algoritmo principal que tiene el nombre "Diseño de la casa" y que es ejecutado por el arquitecto. En pseudocódigo el subalgoritmo para fabricar puertas y colocarlas es el siguiente:

Subalgoritmo: Fabricar y colocar puertas .
  # Materiales a utilizar: madera, barniz, bisagras, chapas.
  # Pasos a seguir:
  Para No_de_Puertas de 0 hasta puertas **:**
    Cortar madera para una puerta;
    Barnizar;
    Poner chapas y bisagras;
    Montar puerta;

El subalgoritmo para colocar las ventanas es el siguiente:

Subalgoritmo: Colocar ventanas.
  # Materiales: Aluminio, remaches, pintura, vidrio;
  # Pasos a seguir:

Para k desde 0 hasta No_de_ventanas **:**
    Cortar aluminio y armar ventana;
    Colocar vidrios;
    Colocar ventana;

Estos subalgoritmos se llaman desde el algoritmo principal:

Nombre: Algoritmo construir casa.
  # Pasos a seguir:

1. Construir cimientos, paredes y techos;
2. Fabricar y colocar puertas;
3. Colocar ventanas;

En este ejemplo el subalgoritmo "Fabricar y colocar puertas" requiere que la casa ya tenga cimientos, paredes y techos. Los datos de entrada son los materiales para las puertas. Para el subalgoritmo "Colocar ventanas", el dato de salida es la ventana colocada en la posición deseada. Este es un ejemplo de como un arquitecto (**algoritmo principal**) necesita de otros colaboradores (**subalgoritmos**) para terminar la tarea encomendada.

Como se mencionó anteriormente, los subalgoritmos se pueden presentar de dos tipos, a saber, las funciones y los procedimientos. En la siguiente sección describimos las propiedades de las funciones y en una sección posterior los procedimientos.

8.3 Funciones

Las funciones son un tipo especial de subalgoritmos. Las funciones son subalgoritmos cuyo formato es el siguiente:

función nombre_de_la_función (parámetros)

donde **función** es la palabra clave para indicar que se define una función. Cada función debe tener un nombre dado por **nombre_de_la_función**. Los datos especificados en **parámetros** son los datos que se pasan hacia la función desde el algoritmo principal y que deben ser del mismo tipo tanto en la función como en el algoritmo principal. Los parámetros también reciben el nombre de **argumentos** de la función. Para llamar o invocar una función desde un algoritmo principal simplemente la invocamos de la siguiente manera:

nombre_de_la_función (parámetros)

Por ejemplo, si tenemos una función que calcula la suma de enteros desde 1 hasta n definida en la función entera `sumatoria`, y deseamos obtener la sumatoria de 1 hasta 8, simplemente la llamamos con:

s ← sumatoria(8)

pero su definición debe ser como

función sumatoria(n)

Es importante observar la siguiente regla:

> Las funciones se deben definir antes de ser utilizadas en el algoritmo principal o en otro subalgoritmo o función.

De no hacerlo se marcará un error y el algoritmo no se podrá compilar ni ejecutar. El formato para definir una función es:

función nombre_de_la_función(parámetro 1,..., parámetro M)
  Variables:
  tipo: variable 1, ..., **tipo** variable N
  tipo resultado
  INICIO
    ⋮
    Instrucciones de la función.
    ⋮
    Regresa (resultado)

Nótese que las instrucciones de la función llevan una sangría. A la variable resultado deberá asignársele un valor en las instrucciones que constituyen el cuerpo de la función. La función finaliza con la instrucción **Regresa**.

Ejemplo 8.2

Números primos

El algoritmo para encontrar si un número dado es primo se consideró en un capítulo anterior. En ese ejemplo se implementaba una división y se checaba si el residuo era cero para saber si el número dado era divisible por algún número distinto de él mismo. Estas operaciones las podemos implementar en un subalgoritmo por medio de la siguiente función:

```
función primo (número)
# Dado un número entero positivo, determina si es primo o no.
    Variables:
    Entero: divisor, cociente
    Lógico: primo

    INICIO
     es_primo ← verdadero # Suponemos que sí es primo.
     divisor ← 2 # Se iniciará el recorrido desde 2 y llegará hasta
       # la raíz cuadrada del número a menos que antes se
       # encuentre un divisor exacto.

    Mientras divisor <= sqrt(número) ∧ es_primo :
        cociente ← número // divisor;# División entera.
        Si cociente * divisor = número :
            es_primo ← falso; # No es primo.
      Si_no
            divisor ← divisor + 1;
      Fin_Si
    Fin_Mientras

    Regresa (primo)
```

En este subalgoritmo realizamos la decisión acerca de si el número es primo. En el caso de que sea primo usamos la variable lógica es_primo para que el programa principal sepa si el número dado es primo. Si es_primo es verdadero, entonces el número es primo y si es falso no es primo. Con este subalgoritmo, el algoritmo principal queda como:

Algoritmo: Muestra_primos
Muestra si el número es primo.
Variables
  Entero: n;

INICIO
  Recibir: n;
  es_primo = primo(n); # Aquí se checa si el número es primo.

    Si es_primo **:**
      Mostrar: SÍ es primo; # Si es primo.
    Si_no :
      Mostrar: NO es primo; # No es primo.
    Fin_Si

FIN

Como claramente se ve, el algorimo se ha simplificado sustancialmente gracias al uso de un subalgoritmo que decide si el número dado es primo.

El número de parámetros que se pasan en una función debe ser siempre el mismo tanto en la definición de ella como en el llamado de la función.

8.4 Funciones en Python

Existen dos categorías de funciones en Python: Las funciones incluidas en las bibliotecas que se incluyen con Python y las funciones que se crean por el usuario. Las funciones en Python usan el mismo formato descrito para el pseudocódigo.

8.4.1 Funciones de Python

Python tiene incluidas numerosas funciones agrupadas en distintas bibliotecas. Estas bibliotecas se deben importar previas a su uso. Ya hemos visto en los capítulos previos la biblioteca `math` y la biblioteca `random`, pero existen muchas otras bibliotecas, algunas de las cuales se deben instalar previas a su uso. Ejemplos de estos casos son los conjuntos de bibliotecas que se encuentran en los paquetes `numpy` para cálculos numéricos y en `matplotlib` para graficación y visualización, que se usan en el Capítulo 11. Las funciones que se tienen predeterminadas en la biblioteca `math` son las que se encuentran en una calculadora científica y se muestran en una lista parcial en la Tabla 8.1. Todas las demás bibliotecas incluyen funciones que siguen el mismo formato y se deben importar antes de llamarlas o invocarlas. En el caso de `numpy` y `matplotlib` se deben importar tanto el paquete como la biblioteca, como se verá en el Capítulo 11.

Tabla 8.1 Algunas funciones disponibles en la Biblioteca `math` de Python.

Función	Nombre	Ejemplo
raíz cuadrada	`sqrt(x)`	`sqrt(2.3)= 1.5166`
seno de x	`sin(x)`	`sen(2.3) = 0.7457`
módulo d y residuo de x	`divmod(x, a)`	`divmod(7, 2) = (3,1)`
seno hiperbólico de x	`sinh(x)`	`senh(2.3) = 4.9370`
logaritmo natural de x	`log(x)`	`log(2.3) = 0.8329`
logaritmo decimal de x	`log10(x)`	`log10(2.3) = 0.3617`
exponencial de x	`exp(x)`	`exp(2.3) = 9.9742`
antilogaritmo decimal	`10**x`	`10**2.3 = 199.5262`
ceiling	`ceil(x)`	`ceil(2.3) = 3`
floor	`floor(x)`	`floor(2.3) = 2`
potencia	`pow(x, y)`	`pow(2, 3) = 2**3 = 8`

8.4.2 Funciones definidas por el usuario

Las funciones definidas en las bibliotecas no son siempre todas las que el diseñador necesita, por lo tanto en la mayoría de las ocasiones es necesario diseñar nuestras propias funciones. Los subalgoritmos implementados como funciones deben definirse antes de usarse, por lo tanto se colocan antes del programa principal. La definición de una función en Python debe cumplir las mismas reglas que cumple el programa principal. Estas reglas son:

- El nombre de la función solamente debe usarse para definir la función y sus llamados.
- Los parámetros de una función deben tener el mismo orden en la función y en el llamado de ella en el programa principal.

El formato en Python para definir las funciones es:

```
def nombre_de_la_función(parámetro_1,..., parámetro_n) :
          ⋮
     INSTRUCCIONES
          ⋮
     return variable
```

Notamos lo siguiente:

- La función debe empezar con la palabra clave `def`.
- El nombre de la función se escribe después de la palabra `def`.
- Los parámetros de la función se escriben entre paréntesis separados por comas. También se conocen como *argumentos*.
- Después de los parámetros se escriben dos puntos.
- Las siguientes líneas contienen las instrucciones que forman el cuerpo de la función y llevan una sangría.
- La función termina cuando se usa la instrucción `return` seguida de la variable que se desea calcular en la función.

Consideremos el ejemplo 8.2 que se realizó en pseudocódigo y que ahora implementamos en Python para ilustrar las funciones.

Ejemplo 8.3

Números primos en Python

El algoritmo para encontrar si un número dado es primo se consideró en un capítulo anterior. En ese ejemplo se implementaba una división y se checaba si el residuo era cero para saber si el número dado era divisible por algún número distinto de él mismo. Estas operaciones las podemos implementar en un subalgoritmo por medio de la siguiente función:

```
1  def primo (numero):
2     # Dado un número entero positivo, determina si es primo o no.

3     # INICIO
4     from math import sqrt
5     es_primo = True # Suponemos que sí es primo.
6     divisor = 2 # Se iniciará el recorrido desde 2 y llegará hasta
7        # la raíz cuadrada del número a menos que antes se
8        # encuentre un divisor exacto.
9     while divisor <= sqrt(numero) and es_primo :
10       cociente = numero//divisor # División entera.
11       if cociente * divisor = num :
12           es_primo = False # No es primo.
13       else:
14           divisor = divisor + 1# Se sigue buscando.
15    return (es_primo)
```

En este subalgoritmo realizamos la decisión acerca de si el número es primo. En el caso de que sea primo usamos la variable lógica `es_primo` para que el programa principal sepa si el número dado es primo. Si `es_primo` = 1 entonces el número es primo y si vale cero no es primo. Con este subalgoritmo, el algoritmo principal queda como:

```
1  # Algoritmo: Muestra_primos
2  # Muestra si el número es primo.

3  # INICIO
4  n = int(input("Dame un número entero"))
5  for num in range( n ) :
6        es_primo = primo(n) # Aquí se checa si el número es primo.
7        if (es_primo):            # primo debe ser True.
8           print( "SÍ es primo." )        # SÍ es primo.
9        else:
10          print( "NO es primo." )        # NO es primo.
11 # FIN
```

Ejemplo 8.4

Solución de una ecuación de segundo grado

En la solución de una ecuación de segundo grado $ax^2 + bx + c = 0$ es necesario calcular la raíz cuadrada del discriminante $d = \sqrt{b^2 - 4ac}$ que recibe los valores de a, b y c del programa principal. La función que calcula la raíz cuadrada del discriminante puede escribirse como:

```
1  def discriminante( a1, b1, c1):
2     # Esta función calcula la raíz cuadrada del discriminante.
3     d = math.sqrt(b1*b1 - 4*a1*c1)
4     return d
```

y el *llamado* a esta función desde el programa principal es:

```
1   # Programa principal

2   import math
3   # Se reciben los coeficientes.
4   a = float(input("Dame el coeficiente de x**2: "))
5   b = float(input("Dame el coeficiente de x: "))
6   c = float(input("Dame el término independiente: "))

7   # Se calcula el discriminante en la función.
8   d1 = discriminante(a, b, c)

9   # Se calculan las raíces y se despliegan.
10  x1 = (-b + d1)/2/a
11  x2 = (-b - d1)/2/a
12  print("La primera raíz es: ", x1 )
12  print("La segunda raíz es: ", x2 )
```

Podemos usar otras dos funciones para simplificar el programa principal. Una de ellas es para recibir los coeficientes a, b y c. La otra es para calcular las raíces y desplegarlas. Estas funciones son:

```
def lectura( ) :
    # Función que recibe los coeficientes de una ecuación de segundo grado
    # de la forma a*x**2 + b*x + c = 0.
    # Recibe los coeficientes a, b, c.
    a = float(input("Dame el valor de a: "))
    b = float(input("Dame el valor de b: "))
    c = float(input("Dame el valor de c: "))
    return [a, b, c] # Los coeficientes están en una lista.
```

Notamos que la función `lectura` no recibe argumentos pero debemos escribir los paréntesis vacíos. Además, sí regresa una lista que contiene los tres coeficientes de la ecuación de segundo grado los que se van a pasar a la siguiente función `calcular` junto con el valor de la raíz cuadrada del discriminante. La siguiente función calcula las raíces: `x1` y `x2`:

```
def calcular(a, b, c, d) :
    # Función que recibe los coeficientes y el discriminante de
    # una ecuación de segundo grado de la forma
    # a*x**2 + b*x + c = 0 y calcula las raíces x1 y x2.
    x1 = (-b + d)/2/a
    x2 = (-b - d)/2/a
    return (x1, x2) # Regresa x1 y x2 en una tupla.
```

El programa principal es entonces:

```
1  # Programa principal
2  ''' Este programa calcula las raíces de una ecuación de
3  segundo grado de la forma: a*x**2 + b*x + c = 0.'''

4  import math
5  # Recibe los coeficientes:
6  [a, b, c] = lectura( )

7  # Calcula el discriminante:
8  d1 = discriminante(a, b, c )

9  # Calcula las raíces:
10 (x1, x2) = calcular(a, b, c, d1 )

11 # Despliega las raíces:
12 print("La primera raíz es: ", x1 )
13 print("La segunda raíz es: ", x2 )
14 # Fin del programa principal.
```

Una corrida para la ecuación, primero llama a la función `lectura`, después calcula el discriminante y finalmente calcula las raíces. Si el valor del discriminante es menor que cero, la raíz cuadrada no se puede calcular y se marca un error. Para la ecuación:

$$2x^2 + 3x - 2 = 0$$

nos calcula las raíces:

```
x1 = -2 y x2 = 0.5.
```

8.5 Procedimientos

Los procedimientos, al igual que las funciones, son subalgoritmos que realizan tareas que forman parte de una solución general.

El formato general de un procedimiento es el mismo de las funciones, pero no incluyen una instrucción Regresa (`return`).

A diferencia de las funciones, NO devuelven o regresan una variable. Se usan directamente como una instrucción del algoritmo principal. Entonces el formato es:

procedimiento nombre(parámetro1, . . . , parámetroN)
    Variables
    tipo: variable_1 ,..., variable_N # Variables de uso interno
                # NO disponibles para el programa principal.
    INICIO
        Instrucciones
    Fin del Procedimiento

Unos ejemplos nos ayudarán a entender mejor los procedimientos.

Ejemplo 8.5

Lista de 20 enteros

Se desea elaborar un algoritmo que haga una lista de 20 enteros ordenados de manera vertical. Entre cada entero deseamos escribir una línea formada por tres guiones bajos, como se muestra a continuación:

1
___
2
___
⋮
___
20
___

Notamos que la línea se repite después de escribir cada número y al ser repetitiva podría hacer más difícil el diseño del algoritmo principal. Por lo tanto es adecuado dejar las tres líneas para un subalgoritmo del tipo procedimiento. El procedimiento para escribir estas tres líneas es:

procedimiento dibujar_línea( )
    Variables
    Entero: i # Variable de uso interno.
    INICIO
    Para i ← 0 hasta 2 :
        Muestra " _" ;
    Fin_Para

Como podemos ver este procedimiento solamente escribe los tres guiones bajos que separan los números en la lista y que no pasa ningún parámetro entre él y el algoritmo principal. El algoritmo principal que llama a este procedimiento es:

Algoritmo que despliega una lista de enteros del 1 al 20.

Variables:
    enteros: a, b, k
Para k ← 1 a **20 :**
    Muestra : k
    dibujar_línea( )
FinPara
FIN

En Python el procedimiento es el siguientes:

```
1  def dibujar_linea( ) :

2      for i in range(3) :
3          print( " _ ", end = " " )
4      print( ) # Para salto de renglón.
```

En la instrucción `print` incluimos `end= " "` para omitir el salto de renglón y hacer que se escriban los tres guiones juntos. El algoritmo principal es:

```
1  # Algoritmo que despliega una lista de enteros del 1 al 20.
2  # Programa principal

3  for k in range(1, 21) :
4       print( k )
5       dibujar_linea( )

6  # Fin del algoritmo.
```

Una corrida de este algoritmo produce el resultado deseado.

Ejemplo 8.6

Obtención del módulo y cociente de dos números enteros

El módulo de dos números enteros a y b se define como el residuo de la división a/b. Para obtener el residuo entonces es necesario efectuar la **división entera**, multiplicar el cociente por el divisor y este resultado restárselo al dividendo. Un algoritmo que realiza esto puede realizarse con un procedimiento que usa la función módulo para obtener el residuo y la división para obtener el cociente[1]. Una vez obtenidos estos resultados se regresan al programa principal donde se muestran. El procedimiento es:

procedimiento modulo(a, b)
  INICIO
    Variables:
    Enteros: residuo, cociente
    residuo ← **modulo**(a, b)
    cociente ← a//b
    Mostrar: residuo, cociente
  FIN

[1]La función `divmod(a, b)` realiza estas operaciones.

El algoritmo principal es:

INICIO
Variables:
Enteros: a, b
Recibir: a, b
modulo(a, b)
FIN

La realización en Python de este algoritmo es la siguiente:

```
1  def modulo( a, b ):
2      residuo = a % b
3      cociente = a // b # División entera.
4      print( "residuo", residuo )
5      print( "cociente", cociente )

6  # Programa principal

7  a = float(input( "Dame el valor de a: "))
8  b = float(input( "Dame el valor de b: "))

9  modulo(a, b)
```

Para los valores de `a = 23` y `b = 5` obtenemos `residuo = 3` y `cociente = 4`.
Podemos hacer notar los siguientes puntos importantes:

- El nombre de la función o procedimiento es único y no puede volver a usarse como nombre por ninguna variable, función o procedimiento.
- El número de parámetros en la función o procedimiento debe ser igual en su definición y en su llamado.
- Los nombres de los parámetros pueden cambiar. Lo único que no puede cambiar es el número de parámetros.
- Las variables definidas en la función o procedimiento son independientes de las variables definidas en el programa principal o en cualquier otra función o procedimiento, incluso ocupan distintas localidades de memoria en la computadora. A este tipo de variables se les llama variables locales.

Ejemplo 8.7

Cálculo del factorial de un número

Ahora podemos calcular el factorial de un número por medio del uso de una función. La función factorial y el programa principal son:

```
1  def factorial( n):
2     fact = 1 # Se declaran las variables y se inicializa fact a 1.
3     for k in range(2, n + 1): # Se calcula el factorial en fact.
4        fact = fact*k
5     return fact # Se regresa la variable fact.

6  # El programa principal es:
7  # Mostrar un mensaje para leer un número.
8  print (" Dame un número entero: ")
9  numero = int ( input( ))
10 factor = factorial(numero)
11 # Se despliega el factorial del número.
12 print ("El factorial del número ", numero , " es " , factor)
```

Este programa se prueba para n = 6 y el resultado es: 720.

8.5.1 Recursividad

Las aplicaciones que hemos mostrado hasta este punto tienen instrucciones que se realizan una después de otra. Pero en algunos casos es conveniente usar una técnica que se conoce como **recursividad**. Una función recursiva es aquella que se llama a sí misma de manera directa. El tema de la recursividad es bastante complejo y se enseña en cursos avanzados por lo que en esta sección lo mostraremos con un ejemplo.

Ejemplo 8.8

Ejemplo 8.8 Cálculo del factorial usando recursividad

Como ya dijimos, la recursividad consiste en que una función se llame a sí misma. Por ejemplo, la función factorial se puede redefinir de la siguiente manera, llamándose de manera recursiva:

```
1 def factorial( n ):
2     fact = 1 # Se inicializa fact.
3     if (n > 1):
4         fact = n*factorial(n - 1) # Se calcula el factorial.
5     else:
6         fact = 1
7     return fact # Se regresa la variable fact.
```

Vemos que la función `factorial` se llama a sí misma en el renglón 4 de la función. Con `n = 6` obtenemos que el factorial es 720 como se esperaba.

8.6 Funciones lambda

En ocasiones las funciones solamente requieren dos o tres renglones para su realización, por ejemplo, la función discriminante del Ejemplo 8.4. En esos casos Python tiene otra forma de declarar las funciones en un sólo renglón. Dichas funciones se conocen como funciones anónimas `lambda` y su formato es el siguiente:

```
f = lambda parámetros: expresión de la función f
```

Por ejemplo, la función `f(x) = 3*x` es:

```
f = lambda x: 3*x
```

que se puede usar como:

```
>>> f(8)

24
```

La funcion discriminante del Ejemplo 8.4 se puede escribir como:

```
>>>  d = lambda a, b, c: math.sqrt(b*b - 4*a*c)
```

donde debemos importar la biblioteca `math` para poder usar la función `sqrt`. Entonces se tiene :

```
>>> import math
>>> d(1, 5, 3)
3.605551275463989
```

8.7 Llamado por valor y llamado por referencia

Los ejemplos de funciones que hemos presentado hasta este momento pasan las variables usando lo que en otros lenguajes como Java y C++ se conoce como "paso por valor". En esos lenguajes lo que se hace es crear una nueva localidad de memoria donde se copian los valores de los parámetros de la función. De esta manera, la memoria ocupada aumenta de tamaño. Aunque en Python no se aumenta la memoria, el paso de variables es equivalente a paso por valor.

En el caso de que la variable sea una lista, para pasar este tipo de variables Python, al igual que otros lenguajes de programación, usa lo que se conoce como "paso por referencia". En ese caso la variable se puede modificar en la función o procedimiento y el cambio siempre se realiza usando la referencia a la localidad de memoria donde se almacena la variable.

Como ejemplo consideremos el programa en Python compuesto de un procedimiento y un programa principal:

```
1  def procedimiento(x):
2      x = 3

3  # Programa principal
4  x = 12
5  procedimiento(x)
6  print(x)
```

Este programa despliega `x = 12`, ya que aunque el valor de `x` se modificó en el procedimiento, éste no modificó el valor de `x` en el programa principal.

Ahora bien, si `x` es una lista, la lista se "pasa por referencia". Para ver esto usamos el siguiente programa:

```
1  def procedimiento(x):
2      x.append(5)

3  # Programa principal
4  x = [1, 2]
5  procedimiento(x)
6  print(x)
```

El programa principal genera la lista `[1, 2]` que se pasa al procedimiento. Este procedimiento agrega el elemento `5` al final de la lista. Como la lista se pasa por referencia, automáticamente aparece en el programa principal en el llamado del procedimiento. La instrucción `print` despliega:

```
[1, 2, 5]
```

Todos los arreglos, de cualquier tipo y de cualquier dimensión, se pasan por referencia.

8.8 Variables locales y globales

Las variables que se definen en el programa principal se pueden usar en las funciones o procedimientos que las reciben como argumentos. Por otro lado, las variables que se definen dentro de una función solamente son válidas dentro de esa función. A las variables de una función se les llama **variables locales**.

En las funciones que se han presentado, las variables que corresponden al mismo parámetro en el programa principal y en la función NO se comparten. Por ejemplo, en el ejemplo 8.4, las variables `a`, `b` y `c` en el programa principal no son las mismas que en la función discriminante donde corresponden a `a1`, `b1` y `c1`.

Las variables correspondientes a la función solamente son accesibles por la función, es decir, se pueden usar solamente por ella. Si deseamos usarlas afuera de la función se marcará un error. Esto quiere decir que una variable del algoritmo principal que se pasa a una función donde se modifica y es variable local, cambia su valor en la función pero NO en el algoritmo principal. Consideremos el siguiente ejemplo en Python:

Ejemplo 8.9

Variables locales

Se desea ver el valor de la variable `x` en el programa principal y en la función. La función `f(x)` es:

```
1  def f(x):
2      x = x + 1
3      print ( 'Valor de x dentro de f =', x)
4      return x

5  # Algoritmo principal
6  x = 3
7  print( 'Valor inicial de x = ', x)
8  z = f(x)
9  print ( 'Valor de x después de llamar a f(x). x = ', x)
```

Una corrida produce:

```
Valor inicial de x = 3
Valor de x dentro de f = 4
Valor de x después de llamar a f(x): x = 3
```

El valor de `x` antes de llamar a la función vale lo mismo y no se afecta por el cambio que se le hace dentro de la función. Esto es porque la variable `x` en la función `f(x)` es una **variable local.**

Las **variables globales** son aquellas que no importa donde se usen o modifiquen siempre conservan los valores asignados, ya sea en el algoritmo principal o en las funciones o procedimientos. Para hacer que una variable sea **global** hay que definirla como tal por medio de la instrucción:

```
global variables
```

donde `variables` se refiere a las variables que se desea sean globales y puedan mantener su valor en las distintas partes del algoritmo.

Ejemplo 8.10

Variables locales - Continuación

Para hacer que la variable `x` sea global insertamos en la función la instrucción:

```
global x
```

y omitimos el parametro `x` en el llamado de la función ya que la variable es global. De esta manera la función y el algoritmo principal quedan como:

```
1  def f( ):
2      global x
3      x = x + 1
4      print ( ' Valor de x dentro de f = ', x)
5      return x

6  # Algoritmo principal
7  x = 3
8  print( ' Valor inicial de x = ', x)
9  z = f( )
10 print( 'Valor de x después de llamar a f(x). x = ', x)
```

El resultado es:

```
Valor inicial de x = 3
Valor de x dentro de f = 4
Valor de x después de llamar a f(x). x = 4
```

Como vemos ahora el valor modificado de x se ha pasado al algoritmo principal ya que la variable **x** se ha declarado como **variable global**.

En el caso de listas, para una lista generada o creada en el programa principal y que se pasa a la función, si dicha lista solamente se modifica dentro de la función, los cambios se reflejan de manera global, por ejemplo en:

```
a = [3, 5, 6, 7]
a.append(9)
```

En este caso la lista a solamente se modifica pero sigue siendo la misma lista ahora dada por:

```
a = [3, 5, 6, 7, 9]
```

En este caso la lista **sigue siendo una variable global**. Pero si se reasigna su valor como en:

```
a = a + [9]
```

Entonces se trata de una **nueva variable que se trata como variable local**. Por lo tanto, no se modifica en el programa principal, solamente en la función.

8.9 Ejemplos adicionales

Como se explicó antes, un procedimiento es un subalgoritmo que no regresa ninguna variable. Por lo general se usan para realizar alguna acción directamente sobre las variables o para una acción como desplegar variables, entre otras. Como ejemplo, supongamos que el procedimiento escritura solamente va a escribir un mensaje si una variable **x** es la unidad y va a escribir otro mensaje si la variable es cero. La variable que se pasa es **x**. Como el procedimiento no regresa ningún parámetro se tiene entonces:

```
1  def escritura( x ):
2     if (x == 1):
3        print ("La variable es la unidad.")
4     else:
5        print ("La variable es cero.")
```

Vemos que este procedimiento no incluye `return` ya que no regresa ninguna variable o parámetro. Sin embargo, al ser llamado se ejecuta escribiendo la frase que corresponda. Otra variante de los procedimientos es cuando no regresan ningún parámetro y además no se les pasa ningún parámetro. El mismo procedimiento anterior puede servir para ilustrar esto. Por ejemplo, si solamente deseamos escribir un mensaje, entonces no le pasamos el parámetro `x`. El procedimiento queda como:

```
1  def escribir_mensaje( ):
2     print("Solamente se escribe este mensaje.")
```

Un programa principal que llama a este procedimiento es:

```
escribir_mensaje( )
```

Notamos que el procedimiento de escritura lleva sus paréntesis vacíos ya que no se pasa ningún parámetro.

Ejemplo 8.11

Tiro de un dado

Cuando se tira un dado el resultado es aleatorio. Para poder simular un tiro aleatorio usamos la función `random` que sirve para generar números aleatorios. Esta función está incluida en la biblioteca `random`. El formato de esta función es

```
import random
x = random.random( )
```

lo cual va a generar un número aleatorio real entre 0 y 1. Otra función que genera números enteros aleatorios es:

```
x = random.randint(a, b)
```

que genera un número aleatorio entero entre `a` y `b`. Podemos usar un ciclo `Para` y

operaciones básicas para obtener tantos números aleatorios como se necesiten en el rango deseado. Por ejemplo para obtener diez números aleatorios reales entre 10 y 76 usamos:

```
import random
for k in range(10):
    x = random.random( )*(76 - 10) + 10
    print (x)
```

y para obtener 10 enteros entre 10 y 76 usamos:

```
import random
for i in range(10):
    x = random.randint(10, 76)
    print(x)
```

Para nuestro ejemplo necesitamos un número entero entre 1 y 6 lo que se obtiene en un procedimiento:

```
def dado( ):
    from random import randint
    global numero
    numero = randint(1, 6)
    print ('El dado sale: ', numero)
```

Este procedimiento se corre y se obtienen números del 1 al 6 en cada corrida. Por ejemplo:

```
>>> dado( )
5
>>> dado()
2
```

No importa cuantas veces ejecutemos el procedimiento dado siempre nos da un número entero del 1 y 6.

Ejemplo 8.12

Serie de Fibonacci

La serie de Fibonacci es una serie donde a partir del tercer término cada elemento se puede obtener mediante la suma de los dos anteriores términos de la serie. El primer elemento de la serie es 0, el segundo término es 1, y a partir del tercer término se pueden calcular con

$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

Un procedimiento que calcula la serie de Fibonacci es:

```
1  def fibonacci( m ):
2      f0 = 0
3      f1 = 1
4      lista_fibo = [ 0, 1]

5      # Ahora se calculan los restantes:
6      for k in range(2, m)
7          f = f0 + f1
8          f0 = f1
9          f1 = f
10         lista_fibo.append(f)

11     # Se despliega la lista:
12     print( 'La serie de Fibonacci es:', lista_fibo)
```

Este procedimiento nos produce la serie de Fibonacci, para los 10 primeros de la serie con:

```
>>> fibonacci( 10 )

[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ]
```

Ejemplo 8.13

Regreso de varias variables de un subalgoritmo

Consideremos un subalgoritmo que reciba tres variables del algoritmo principal y que a cada variable le sume el número 3, regrese una variable y la suma para finalmente escribirlas desde otro subalgoritmo que las reciba del programa principal. El subalgoritmo que modifica a, b y c es:

```
1  def modificar(a, b, c ):
2      a = a + 3
3      b = b + 3
4      c = c + 3
5      suma = a + b + c # Realiza la suma.

6       # Regresa la suma después de modificar los datos.
7      return a, suma # Regresa a y la suma después de modificar los datos.
```

El subalgoritmo que despliega el resultado es:

```
1  def escribir( suma ):
2      print( " La suma es : ", suma )
```

El programa principal es entonces:

```
1  # Programa principal.
2  a = 1; b = 2.3; c = 7.1
3  [ suma, a ] = modificar (a, b, c )
4  escribir (suma)
5  print("El valor de a es:", a)
```

El resultado que vemos es:

```
La suma es: 19.4
El valor de a es: 4
```

Ejemplo 8.14

Suma y producto de los dígitos de un número

Deseamos encontrar los números enteros que satisfagan la condición de que la suma de sus dígitos sea igual al producto, por ejemplo, el número 123 cumple la condición ya que $1 + 2 + 3 = 1 \times 2 \times 3$. Para obtener estos números se necesitan tres funciones:

- La primera función debe regresar la suma de los dígitos de un número que se pasa como argumento, por ejemplo, debe regresar 3 cuando el número dado es 12 y 7 cuando el número dado es 223.
- La segunda función debe regresar el producto de los dígitos. Para el número 12 debe regresar $1 \times 2 = 2$ y para 223 debe regresar $2 \times 2 \times 3 = 12$.
- La última función debe comparar los resultados de las dos funciones y si son iguales indicar que el número satisface la condición deseada.

Para hacer esto debemos dividir el número entre 10 para encontrar las unidades, decenas y centenas, donde la parte de interés es el residuo. Como sabemos el residuo se obtiene con la función módulo. Por ejemplo, si el número es 239, podemos dividir entre 10 y el residuo es 9 con lo que obtenemos el dígito más a la derecha también llamado dígito menos significativo ya que solamente contribuye a las unidades. Si dividimos el cociente, que en este caso es 23, entre 10 el residuo es 3 y ahora ya tenemos el segundo dígito del número. Finalmente, el cociente que es 2 es el último residuo y que es el dígito más significativo. De esta manera tenemos los tres dígitos. Esta tarea de obtener los dígitos lo podemos realizar en una cuarta función usando la función módulo que se realiza con el operador %. Por ejemplo, para el número 178 realizamos:

`178%10` que da el residuo **8** y el cociente de `178/10` es `17`,
`17%10` que da el residuo **7** y el cociente de `17/10` es `1`,
`1%10` que da el residuo **1** y el cociente de `1/10` es `0`,
y se termina el proceso.

Los dígitos de 178 son 1, 7 y 8.

Después de obtener los dígitos en una función realizamos la suma de los dígitos, en otra función realizamos la multiplicación de los dígitos, y en una última función

realizamos la comparación de la suma y la multiplicación. Finalmente, si el resultado de la comparación es verdadero el número dado satisface la condición.

La función que realiza la suma de los dígitos usa un ciclo `for` y se muestra a continuación:

```
1  def adicion( caso ):
2      suma = 0     # Inicializa la suma a cero.
3      for k in range(caso):
4          suma = suma + lista_digitos[k]
5      return suma
```

La función que realiza la multiplicación de los dígitos es:

```
1  def multiplicacion( caso ):
2      producto = 1 # Inicializa el producto a 1.
3      for k in range(caso):
4          producto = producto*lista_digitos[k]
5      return producto
```

La función que **obtiene los dígitos** realiza la obtención de los residuos y cocientes para obtener los dígitos del número.

Restringimos que el número esté entre 1 y 1,000,000.

Para la realización de esta función se usa un equivalente del `switch-case` (ver Capítulo 3) realizado con instrucciones `if` para determinar el orden de magnitud del número. La función definida en Python es:

```
def digitos( numero ):
   global lista_digitos
  if (numero <= 10):
     caso = 1
   elif (numero <= 100):
      caso = 2
   elif (numero <= 1000):
      caso = 3
   elif (numero <= 10000):
      caso = 4
  elif (numero <= 100000):
      caso = 5
  else:
      caso = 6
  for k in range(caso):
     # Obtención del residuo.
     lista_digitos.append(numero%10) # Obtención del residuo.
      numero = numero//10 # Obtención del cociente.
  # Los dígitos están almacenados en el vector lista_digitos[k].
  # El número de dígitos es el valor de caso.
  return caso
```

Finalmente, la función que compara los resultados de nuestras dos funciones `adicion` y `multiplicacion` es:

```
def comparacion( suma, producto):
   resultado = False
   if (suma == producto):
       resultado = True
   return resultado
```

El algoritmo principal recibe el número, llama a las funciones `digitos`, `adicion`, `multiplicacion` y `comparacion` para finalmente escribir un mensaje si es el caso de que el número dado satisface o no las condiciones. Este programa principal es:

```
1  #Programa principal
2  lista_digitos = [ ]
3  print  ("Dame un numero: ")
4  numero = int( input( ) )
5  caso = digitos( numero )
6  print ("Lista de dígitos", lista_digitos )
7  suma = adicion( caso )
8  producto = multiplicacion( caso )
9  resultado = comparacion(suma, producto)
10 if (resultado):
11    print (  " El número SI cumple la condición.")
12 else:
13     print ( " El número NO cumple la condición." )
```

Los siguientes números cumplen la condición: 22, 123, 132, 213, 231. Otros números menores a 231 no satisfacen esta condición.

Ejemplo 8.15

Obtención del perímetro de un triángulo

Dadas las coordenadas de tres puntos, debe construirse un programa que calcule el perímetro del triángulo que forman. El programa debe incluir una función para leer las coordenadas de los puntos y una función que calcule el perímetro. Para leer las coordenadas de los vértices del triángulo usamos una función que se llama `leer_coordenadas`. Esta función recibe las coordenadas `x`, `y` de un punto y las guarda en una lista. Para calcular la distancia usamos el teorema de Pitágoras en una función llamada `distancia` que recibe las coordenadas de dos puntos. Finalmente, en el programa principal leemos las coordenadas de los tres puntos llamando tres veces a la función `leer_coordenadas`, calculando las distancias entre pares de puntos y desplegando el valor del perímetro. A continuación se listan las dos funciones y el programa principal. La función para calcular la distancia es:

```
1  def dist ( p1, p2 ):
2     distancia = sqrt((p1[1]-p2[1])**2+(p1[2]-p2[2])**2)
3     return (distancia) # Se calculó la distancia.
```

La función para leer las coordenadas es:

```
from math import sqrt # Se importa la función sqrt.

def leer_coordenadas ( i ):
   p = [ ]
   print ( 'Dame la coordenada x del punto', i )
   x = float( input( ) )
   p.append( x )
   print ( 'Dame la coordenada y del punto', i )
   y = float( input( ) )
   p.append( y )
   return( p ) # Se recibió un punto.
```

El programa principal se muestra a continuación:

```
# Programa principal
A = leer_coordenadas (1)
B = leer_coordenadas (2)
C = leer_coordenadas (3)
perimetro = dist(A,B) + dist(B,C) + dist(C,A)
print( "El perímetro es: ", perimetro )
```

Ejemplo 8.16

Serie de Taylor

Escribir un programa que realice una serie de Taylor de la función seno, sabiendo que la función seno desarrollada en serie de Taylor tiene la forma:

$$\sin x = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

La solución requiere que se realicen las operaciones de exponenciación y el cálculo del factorial. Este último se puede realizar en una función de la siguiente manera:

```
1  def factorial ( n ):
1     fact = 1 # Se inicializa el valor del factorial
1     for i in range( n ):
1        fact = fact*i
1     return fact
```

El programa principal requiere recibir el valor de **x**, realizar la potencia y sumar los términos. Entonces el programa principal es:

```
1  # Programa principal
2  # Se recibe el valor de x.
3  x = float ( input( 'Dame el valor de x: ' ))
4  S = 0.0 # Se inicializa la suma.
5  for i in range(10):
6     S = S + pow(-1.0, i)*pow(x,(2*i + 1))/fact(2*i+1)
7  # Se despliega el resultado.
8  print( " El valor de la serie es :", S)
```

Una corrida de este programa produce para `x = 2`:

```
Dame el valor de x: 2
El valor de la serie es : 0.909297426825641
```

Si comparamos con el valor exacto de `sen(2) = 0.9092974268256817` vemos que nuestro resultado es bastante exacto con solamente 10 términos en la serie del seno.

Ejemplo 8.17

Máximo común divisor

Se desea desarrollar un algoritmo que calcule el máximo común divisor de dos números. El máximo común divisor de dos números enteros es el mayor entero que los divide exactamente. Usando la función módulo podemos encontrar el entero mayor que simultáneamente satisfaga que el residuo es cero. Nuestro algoritmo programado en Python es:

```
1  a = int ( input( "Número mayor: " ) )
2  b = int ( input( "Número menor: " ) )
3  for k in range ( b, 1, -1):
4     if (a%k == 0 & b%k == 0):
5        print(" El máximo común divisor es: " , k )
6        break
```

Para probar el algoritmo usamos como número mayor 27 y como número menor 7. El resultado es:

```
El máximo común divisor es: 1
```

Ejemplo 8.18

Conversión de decimal a binario

Escribir un programa que dado un número entero decimal introducido por el usuario, lo convierta en un número binario y lo almacene en una de lista de 32 bits. Este problema se puede resolver usando funciones internas de Python y es conveniente ver cómo se puede realizar para poder extenderlo y realizar conversiones a otras bases. El algoritmo debe realizar los siguientes pasos:

1. Leer el número entero decimal.

2. Realizar divisiones entre 2 rescatando el residuo como el bit del resultado y repitiendo la división con el cociente resultante hasta que se termine la posibilidad de dividir.
3. Presentar el resultado.

Como ejemplo consideremos el número entero decimal 23. Al dividir entre dos

$$\frac{23}{2} = 11 \text{ con residuo } 1 \qquad \text{Bit menos significativo.}$$

$$\frac{11}{2} = 5 \text{ con residuo } 1$$

$$\frac{5}{2} = 2 \text{ con residuo } 1$$

$$\frac{2}{2} = 1 \text{ con residuo } 0$$

$$\frac{1}{2} = 0 \text{ con residuo } 1 \qquad \text{Bit más significativo.}$$

El número 23 en binario es entonces:

$$10111$$

En comparación, usando la función `bin` de Python obtenemos:

```
>>> bin(23)
```

'0b10111'

que es el mismo resultado. Para realizar este algoritmo, podemos leer el número en una función, realizar la división en otra función y el resultado en una tercera función. La función en pseudocódigo para recibir el número es:

función leer(número)

Mostrar: "Dame el número a convertir"

Recibir: número

La función para hacer la división es:

```
función dividir( número ) :
    binario = [ ]
    cociente = número
    Mientras ( cociente > 0 ) :
        residuo = cociente % 2
        cociente = cociente // 2
        binario = binario.append(residuo)
    regresa binario
```

La función para mostrar el resultado es simplemente:

```
función desplegar( binario) :
        Mostrar: "El resultado es: ", binario
```

El algoritmo principal es entonces:

```
leer( número)
binario = dividir( número )
desplegar( binario )
```

En Python, las funciones son:

```
def leer( ): # Lee el número.
    print( "Dame el número a convertir")
    numero = int( input ( ) )
    return numero

def dividir( numero ): # Realiza las divisiones.
   binario = [ ]
   cociente = numero
   while (cociente > 0 ):
       residuo = cociente % 2
       cociente = cociente // 2
       binario.append(residuo)
   return binario

def desplegar( binario):
     binario = list( reversed( binario ) )
     print( "El resultado es: ", binario )
```

El programa principal es entonces:

```
# Programa principal
numero = leer( )
binario = dividir( numero )
desplegar( binario )
```

La ventaja de implementar nuestro propio algoritmo es que podemos realizar conversiones de decimal a cualquier base con solamente cambiar el 2 (base binaria) en la función `dividir` por cualquier base.

Ejemplo 8.19

Identificación de palíndromos

Una frase es un palíndromo si se lee lo mismo de derecha a izquierda o de izquierda a derecha, sin contar los espacios entre palabras. Ejemplos de palíndromos son:

Anita lava la tina
Asa
Anula la luz azul a la luna

Para checar si una frase es palíndromo generamos otra frase con los caracteres en orden inverso y luego comparamos caracter por caracter las dos frases. Para esto sólo es necesario comparar la mitad de la frase ya que se compara la primera mitad en la primera frase con la segunda mitad que se encuentra ahora como primera mitad en la segunda frase. Para leer la frase usamos la siguiente función, donde la frase se almacena en una lista:

```
1  def lectura( ):
2     frase = input( 'Dame la frase sin espacios:\n' )
3     lista = [ ]
4      for k in range(len(frase)):
5        lista.append(frase[k])
6     print(lista)
7     return lista
```

Esta función regresa la variable `lista` donde están almacenados los caracteres de la frase. En otra función realizamos la inversión de la frase y la comparación. Para invertir usamos:

```
frase = list( reversed( lista ) )
```

Para realizar la comparación usamos un ciclo Mientras (`while`) y solamente comparamos la mitad de los caracteres de cada lista. Inicializamos un contador para checar si se cumple la condición de igualdad. En el caso de encontrar un caracter que no sea igual NO se incrementa el contador, después se compara el contador con la cuenta de la mitad de la longitud de la lista. Si la longitud de la lista sobre dos es igual al valor del contador se tiene un palíndromo. La función queda como se muestra a continuación:

```
1  def checar(lista):
2      max = len(lista)
3      frase = list(reversed(lista))

4      # Inicializa el índice del ciclo y el contador.
5      numero = 0
6      contador = 0
7      while(numero + 1 <= max/2):
8          if ( frase[ numero ] == lista[ numero ] ):
9              contador += 1 # Se incrementa el contador.
10         numero += 1 # Se incrementa el ciclo del índice.

11     if ( contador == max//2 ):
12         print( " La frase SÍ es palíndromo.")
13     else:
14         print( " La frase NO es palíndromo.")
```

El algoritmo principal es simplemente:

```
1  # Programa principal.
2  lista = lectura( )
3  checar(lista)
```

Si ejecutamos este algoritmo completo tenemos:

```
Dame la frase sin espacios:
anitalavalatina
La frase Sí es palíndromo.
```

Ejemplo 8.20

Cotizaciones de dolar y euro

Diseñe un algoritmo e implemente un programa en Python que use funciones para calcular el promedio de las cotizaciones de la semana y el mayor valor de la semana del dólar y del euro. Las cotizaciones son:

Moneda	lunes	martes	miércol	jueves	viernes	prom.	máximo
dolar	12.23	12.24	12.40	12.45	11.99		
euro	15.60	15.55	16.00	15.90	15.34		

El programa debe ser capaz de:

1. En una función denominada `promedio` : recibir una lista de valores de tamaño `n`, calcular el promedio y regresar el valor promedio.
2. En un procedimiento denominado `mayor` : recibir una lista de valores de tamaño `n`, encontrar el valor máximo, almacenarlo en la posición `n` e imprimirlo.
3. En el programa principal :
   a) Solicitar al usuario las cotizaciones de la semana de cada tipo de moneda y almacenarlas en la lista correspondiente.
   b) Obtener el promedio de las cotizaciones para cada tipo de moneda llamando la función `promedio` y colocarlo en la columna correspondiente.
   c) Obtener el valor máximo de la semana para cada tipo de moneda, usando la función `mayor`.
   d) Mostrar las cotizaciones de la semana, el promedio y el valor máximo tanto para los US dólares como para los euros.

Para la función que calcula el máximo usamos la siguiente función donde los datos de entrada son el vector de cotizaciones x y el tamaño del vector n:

```
1  def mayor(n, x) :
2        mayor = 0 # Se inicializa el mayor.
3        for k in range(n) :
4              if x[k] > mayor # Se checa si x[k] es mayor.
5                    mayor = x[k]
6        return mayor
```

Para calcular el promedio se deben sumar todos los elementos del vector y después dividirse la suma entre n para calcular el promedio. La función promedio realiza esto de la siguiente manera:

```
def promedio(n, x) :
      suma = 0 # Se inicializa la suma.
   for k in range(n) :
      suma = suma + x[k] # Se suman todos los elementos del vector.
      promedio = suma/n # Se calcula el promedio.

   return promedio
```

El programa principal consiste en leer los vectores para la cotización y el euro y almacenarlos en una matriz de 2×5 llamada `moneda[2][5]`:

```
# Programa Principal

# Lista o vector con los días de la semana.
semana = ['lunes', 'martes', 'miércoles','jueves', 'viernes']

# Inicialización de las cotizaciones del dolar y euro.
moneda = [ [ ], [ ] ]

for k in range(2) :
   for dia in range(5) :
      if k == 0 :
         print("Cotización del dolar del día ", dia)
      else :
         print("Cotización del euro del día ", dia)

      moneda[k] += [ float(input( )) ]
```

Cada renglón de la matriz se debe pasar primero a la función `promedio` y luego a la función `mayor`:

```
n = 5
prom_dolar = promedio(n, moneda[0])
prom_euro = promedio(n, moneda[1])
mayor_dolar = mayor(n, moneda[0])
mayor_euro = mayor(n, moneda[1])
```

Los resultados se almacenan en las posiciones 6 y 7 de cada renglón, esto lo hacemos con:

```
# Se agregan los resultados a cada renglón de la matriz.

moneda[0].append(prom_dolar)
moneda[0].append(mayor_dolar)
moneda[1].append(prom_euro)
moneda[1].append(mayor_euro)
```

Ahora los renglones de la matriz son de longitud 7 ya que a las cinco cotizaciones de la semana les hemos agregado el promedio semanal y la cotización mayor. Para desplegarlos usamos:

```
# Despliega el encabezado de la salida.
letrero = ['L', 'Ma', 'Mi', 'J', 'V', 'Prom', 'Mayor' ]
for k in range(7) :
   print(letrero[k], '\t', end= ' ')

# Despliega las cotizaciones, los promedios y los valores mayores.
for k in range(2) :
   print( )
   for j in range(7):
      print(moneda[k][j], "\t", end = " ")
```

Una corrida con los datos mostrados en la tabla produce el siguiente resultado:

```
L       Ma      Mi      J       V       Prom    Mayor
12.23   12.24   12.4    12.45   11.99   12.26   12.45
15.6    15.55   16.0    15.9    15.34   15.678  16.0
```

8.10 Instrucciones de Python del Capítulo 8

Instrucción	Descripción
`def`	Se usa para definir una función.
`return`	Indica las variables que se regresan de la función.

8.11 Conclusiones

Las funciones y procedimientos son implementaciones de subalgoritmos que nos permiten particionar y simplificar un algoritmo que de otra manera sería muy grande y posiblemente difícil de entender y con una posibilidad de cometer errores en su diseño e implementación. Aunado al uso de funciones y procedimientos tenemos la forma de pasar las variables entre el programa principal y las funciones y procedimientos. Finalmente, vimos el concepto de variables locales y globales. Numerosos ejemplos nos permiten aprender a utilizar las funciones y procedimientos en la implementación de subalgoritmos.

8.12 Ejercicios

1. Considere el siguiente programa en Python. Muestre qué resultados produce y explique en cada etapa, cómo los va obteniendo.

```
def misterio ( n ) :
    t = 0
    for i in range (1, 4):
        m = n * i
        print ( n, " X ", i, " son ", m)
        t = t + m
    return (t)
```

Programa Principal.

```
num = int (input ("Número ")
print (" El resultado es: ", misterio (num) )
```

2. Se desea escribir un algoritmo que genere una tabla de multiplicar. Se debe leer en una función el número entero n que indica la tabla de multiplicar que se desea. Otra función debe generar la tabla correspondiente de `n × 1` hasta `n × 10`. Finalmente, una tercera función imprime el resultado.

3. Usando una función lambda y un ciclo Para escriba un algoritmo para desplegar las tablas de multiplicar del 2 y del 4.

4. Se desea generar una matriz o arreglo que tenga tres renglones y dos columnas. Esta matriz debe estar inicializada con ceros. Se debe realizar los siguiente:

   a) Dentro de una función inicializar la matriz con 0's.

   b) En otra función, usando el generador de números aleatorios enteros, cambiar los ceros por un entero aleatorio.

   c) En otra función obtener la transpuesta de la matriz de enteros aleatorios.

   d) Finalmente en otra función, desplegar la matriz transpuesta.

5. Repetir el ejercicio anterior usando variables globales.

6. La conjetura de Goldback afirma que cualquier número par es la suma de dos números primos. Diseñe un algoritmo y prográmelo en Python para que evalúe si un número cumple esta conjetura. Para esto debe cumplir con lo siguiente:

   a) Recibir un número en una función llamada `recibir`.

   b) En otra función checar que el número es par.

   c) Si el paso *b*) se cumple, una tercera función debe buscar dos números primos tales que la suma de ellos es el número dado.

   d) Finalmente, una cuarta función debe escribir los números primos si existen. Si no se cumple el paso *b*), debe indicar que el número dado no es par.

   Ejemplos son $n = 16 = 3 + 13$, $n = 38 = 1 + 37$, $n = 120 = 3 + 117$.

7. La tabla de pago de impuestos mostrada se va a usar para calcular el impuesto sobre la renta (ISR) que un profesionista debe pagar. Realice un algoritmo y su implementación en Python que reciba un sueldo y calcule el impuesto que debe pagar. El impuesto se calcula sumando a la cuota fija el porcentaje correspondiente. Use una función para este cálculo.

Límite inferior	Límite superior	Cuota fija	Tasa de impuesto
0	999	0	5 %
1,000	9,999	50	10 %
10,000	99,999	100	15 %
100,000	999,999	10,000	25 %
1,000,000		100,000	35 %

Por ejemplo, si alguien tiene un sueldo de 234,000 entonces su impuesto es de:

100,000 + 0.25*234,000 = 68,500

8. En un supermercado la promoción del día es la siguiente: Después de haber pasado a la caja, el cliente selecciona un número entre 1 y 5 al azar. Si coincide con el último dígito que registra su número de ticket tiene un descuento del 50 %. Realice un algoritmo y su implementación en Python para que una función obtenga el número aleatorio entre 1 y 5. En otra función se hace la comparación con el número dado por el cliente. Los datos de entrada son la compra total y el número dado por el cliente.

9. Un procedimiento para calcular el domingo de Resurrección en la religión cristiana en el rango de años que va de 1982 a 2048 es el siguiente:

   a) Se calculan a, b, c y d como:

   $a = año\,\%19$
   $b = año\,\%4$
   $c = año\,\%7$
   $d = (19 * a + 24)\,\%30$
   $e = (2 * b + 4 * c + 6 * d + 5)\,\%7$

   b) La fecha requerida se calcula con: marzo $22 + d + e$

   Diseñe un algoritmo que calcule esta fecha. El algoritmo debe recibir el año y debe escribir la fecha requerida. Dado que la fecha puede corresponder al mes de abril, calcule la fecha correcta si es así. Use una función para los cálculos la fecha y otra función anidada que determine si la fecha calculada corresponde al mes de abril.

Capítulo 9
Entrada y salida y de datos con archivos

9.1 Introducción
9.2 Escritura de datos en un archivo
9.3 Escritura de datos numéricos
9.4 Lectura de datos de un archivo
9.5 Lectura y escritura de datos en Excel
9.6 Instrucciones de Python del Capítulo 9
9.7 Conclusiones
9.8 Ejercicios

La memoria es el centinela del cerebro.

William Shakespeare

Objetivos

Los datos de entrada y salida de un algoritmo necesitan ser almacenados en archivos, por lo general. En este capítulo se cubre la manera de almacenar información de distintas maneras.

9.1 Introducción

Hasta ahora los datos de entrada los hemos dado por medio del teclado y los datos de salida los hemos desplegado en la pantalla de la computadora, pero también los podemos escribir en un archivo para uso posterior en otra aplicación o para transportarlos o enviarlos por Internet a otro usuario. En este capítulo cubrimos las distintas maneras de crear, escribir y leer datos en un archivo.

9.2 Escritura de datos en un archivo

En una computadora los datos se almacenan usando la unidad básica de almacenamiento que es el byte y que consiste de 8 bits. Un archivo (file) es una secuencia de bytes. Un archivo siempre tiene un principio y un final. Para escribir a un archivo o para leer datos que estén en él, nuestra aplicación primero debe abrir (open) el archivo. Al finalizar de usar el archivo debemos cerrarlo (close). Para escribir datos en un archivo usamos una variable que se llama identificador (*handle* en inglés). Esta variable permite a Python hacer referencia al archivo. El valor que Python le asigna a la variable no es tan importante como su nombre, que es el que usamos para referenciar el archivo donde vamos a escribir o del que vamos a leer datos. Para abrir un archivo lo hacemos de la siguiente manera:

```
identificador = open( nombre_del_archivo, modo)
```

El significado de las variables es el siguiente:

- `open` es la palabra clave para abrir un archivo.
- `nombre_del_archivo` se refiere al archivo que deseamos abrir si ya existe o crear y abrir.
- `modo` es una variable que indica que acciones deseamos ejecutar al archivo.
- El identificador es una variable con la que nos vamos a referir al archivo.

La variable `modo` tiene alguna de las siguientes opciones:

Modo	Descripción
'r'	Abre el archivo para lectura.
'w'	Abre el archivo para escribir. Si ya existe borra los datos existentes.
'a'	Abre el archivo para escribir. Si ya existe escribe después de los datos existentes.
'r+'	Abre el archivo para leer y escribir.

9.2.1 Escritura de datos alfanuméricos

Una vez abierto el archivo, el siguiente paso es escribir en él. Esto lo hacemos con la instrucción `write` con el siguiente formato:

```
identificador.write('frase')
```

donde `frase` es lo que deseamos escribir en el archivo. Una vez que se escribe lo deseado, lo siguiente es cerrar el archivo con:

```
identificador.close( )
```

Con la instrucción `write` solamente podemos escribir cadenas.

Con un ejemplo ilustramos la manera de escribir un archivo de datos.

Ejemplo 9.1

Escritura de texto en un archivo

Lo primero que tenemos que hacer es seleccionar un nombre de archivo. En este caso seleccionamos `perros.txt` y vamos a escribir nombres de razas de perros:

```
iden = open( 'perros.txt', 'w' )
```

y queremos escribir tres nombres de razas de perros: `collie`, `cocker`, `poodle`. Para que cada nombre quede en un renglón debemos incluir el salto de línea "`\n`". Esto lo hacemos con:

```
iden.write( "collie \n" )
iden.write( "cocker \n" )
iden.write( "poodle \n" )
```

Finalmente, cerramos el archivo con:

```
iden.close( )
```

Una vez terminado esto podemos abrir el archivo `perros.txt` con un editor de textos y ver su contenido. Estas instrucciones las podemos escribir directamente en el IDLE de Python o en un archivo de Python. El programa se guarda con el nombre `archivo_perros.py`.

Observamos que debemos incluir el salto de línea "`\n`", ya que la instrucción `write` NO lo genera automáticamente como la instrucción `print` que sí lo genera cada vez que se usa.

La instrucción `write` **NO** genera salto de línea.

Ejemplo 9.2

Escritura en un archivo usando la instrucción `writelines`

El ejemplo anterior lo podemos escribir de manera más compacta usando la instrucción `writelines` de la siguiente manera:

Después de crear y abrir el archivo `perros2.txt` con:

```
iden = open( 'perros2.txt', 'w' )
```

usamos la instrucción `writelines` de la siguiente manera:

```
iden.writelines( [ "collie \n", "cocker \n", "poodle \n" ] )
```

Finalmente, cerramos el archivo:

```
iden.close( )
```

El archivo completo es:

```
iden = open( 'perros2.txt', 'w' )
iden.writelines( [ "collie \n", "cocker \n", "poodle \n" ] )
iden.close( )
```

Este archivo se guarda como `razas2.py`. El resultado es el mismo que el anterior pero es más compacto.

Nótese que dentro de la instrucción `writelines` los renglones están dentro de una lista de Python que se escribe entre corchetes.

9.2.2 La instrucción `with`

La instrucción `with` proporciona otra manera de abrir un archivo para escritura o lectura. La ventaja de usar `with` es que el archivo se cierra después de usarlo. El formato es:

```
with open( "nombre_del_archivo.txt", "modo" ) as identificador:
    identificador.write( " frase" )
```

La primera línea termina con dos puntos y las siguientes líneas deben estar con sangría.

Ejemplo 9.3

Escritura de datos alfanuméricos usando `with`

Supongamos que deseamos crear el archivo `jugadores.txt` y queremos escribir el nombre de un primer jugador que se llama Pérez y el nombre de un segundo jugador que se llama Mario. Usamos como identificador la palabra `fut`. Esto lo hacemos con:

```
with open( "jugadores.txt", "w" ) as fut:
    fut.writelines( [" Pérez \n", "Mario \n" ] )
```

9.3 Escritura de datos numéricos

Con la instrucción `write`, hasta ahora solamente hemos escrito datos alfanuméricos, por lo que si deseamos escribir datos numéricos los tenemos que convertir a datos alfanuméricos, es decir, convertirlos a cadenas con la instrucción `str( )` o incluirlos dentro de una cadena.

Ejemplo 9.4

Escritura de datos numéricos

Siguiendo el ejemplo anterior, ahora deseamos crear el archivo: `jugadores.txt`, pero

además deseamos escribir los goles que anotaron que son 12 y 13, respectivamente, para Pérez y Mario. Para este fin creamos una lista con los datos de estos jugadores incluyendo un espaciamiento con "\t" de la siguiente manera:

```
datos = [ "Pérez \t 12 \n", "Mario \t 13 \n" ]
```

Ahora podemos escribir la lista datos de la manera que sabemos hasta ahora:

```
archivo = open( "jugadores2.text", "w" )
datos = [ "Pérez \t 12 \n", "Mario \t 13 \n" ]
archivo.writelines(datos)
archivo.close( )
```

La instrucción write también permite escribir valores numéricos con formato. Si tenemos una variable a = 3.2316 usamos lo siguiente:

```
archivo.write( '%s'  %a )
```

Ejemplo 9.5

Forma alterna de escritura de datos numéricos

Escribir datos alfanuméricos y numéricos se puede hacer de la siguiente manera: por ejemplo, para el jugador Pérez, si la variable goles_Perez es 12, entonces:

```
goles_Perez = 12
archivo = open("nuevo.dat", "w")
archivo.write("Pérez \t %s"  %goles_Perez )
archivo.close( )
```

En este caso estamos incorporando el nombre del jugador, el tabulador \t y %s en la misma cadena, seguida del símbolo %goles_Perez. Después de escribir datos en el archivo lo cerramos.

Las técnicas anteriores se pueden combinar para escribir datos en la forma deseada por el usuario. El siguiente ejemplo muestra como crear una tabla en un archivo.

Ejemplo 9.6

Creación de una tabla en un archivo

Supongamos que deseamos escribir una tabla de datos en un archivo. Si la tabla deseada es la siguiente:

1.2	3.0	-7.45
-8.3	6	17.563
98.78	-12.5	-46.2332
2	-567.2	8.43
1	0	1.2013

Vemos que nuestra tabla consiste de 5 renglones y 3 columnas. Para Python, esta tabla la podemos dar en una matriz la cual es una lista donde cada elemento de la lista es otra lista que llamamos `datos` de la siguiente manera:

```
datos = [ [ 1.2, 3.0, -7.45] ,
          [ -8.3, 6, 17.563] ,
          [ 98.78, -12.5, -46.2332] ,
          [ 2.001, -567.2, 8.43],
          [ 1.000, 0, 1.2013 ] ]
```

para escribir estos datos en un archivo, después de crearlo y abrirlo con:

```
tabla_de_datos = open ( "tabla.dat", "w" )
```

usamos dos ciclos anidados para escribir de renglón en renglón y para cada renglón de columna en columna todos los elementos al archivo `tabla_de_datos.dat`:

```
for renglon in datos:
   for columna in renglon:
      tabla_de_datos.write( "%14.8f " % columna)
   tabla_de_datos.write("\n")
```

Finalmente, cerramos el archivo con:

```
tabla_de_datos.close( )
```

El archivo completo es:

```
datos = [ [ 1.2, 3.0, -7.45] ,
          [ -8.3, 6, 17.563] ,
          [ 98.78, -12.5, -46.2332] ,
          [ 2, -567.2, 8.43],
          [ 1, 0, 1.2013] ]
tabla_de_datos = open ( "tabla.dat", "w" )
for renglon in datos:
   for columna in renglon:
      tabla_de_datos.write( "%14.8f" %columna)
   tabla_de_datos.write("\n")
tabla_de_datos.close( )
```

Los resultados que obtenemos se encuentran en el archivo `datos.dat` que se muestra a continuación al abrirlo con el WordPad:

```
   1.20000000     3.00000000    -7.45000000
  -8.30000000     6.00000000    17.56300000
  98.78000000   -12.50000000   -46.23320000
   2.00000000   567.20000000     8.43000000
   1.00000000     0.00000000     1.20130000
```

9.4 Lectura de datos de un archivo

Para leer datos de un archivo necesitamos saber el formato de los datos. De la sección anterior sabemos que los datos se almacenan como cadenas, de esta manera, si son numéricos tenemos que convertirlos de cadena al formato adecuado. La primera acción que tenemos que realizar es abrir el archivo para lectura. Esto lo hacemos con:

```
identificador = open("nombre_del_archivo", "r" )
```

Posteriormente, procedemos a realizar la lectura de los datos de acuerdo al formato en que se guardaron en el archivo. Con ejemplos mostramos los distintos casos que se pueden presentar.

9.4.1 Lectura de datos de un archivo

Cuando leemos datos de entrada de un archivo, los datos pueden ser de dos tipos: alfanumérico y numérico. Además, en un mismo archivo se pueden combinar de los dos tipos de datos. Primero cubrimos la manera de leer datos numéricos.

Ejemplo 9.7

Lectura de datos numéricos

Supongamos que tenemos el archivo `enteros.dat` con los siguientes datos y de los cuales queremos calcular su promedio:

```
 10
 23
-34
 66
 88
-23
```

El primer paso es abrir el archivo con:

```
mi_archivo = open("enteros.dat")
```

El siguiente paso es leer el dato renglón por renglón, convertirlos a enteros, y sumarlos para finalmente dividir entre el número de elementos que se leyeron. La suma de estos números es 130 y el número total de elementos leídos es 6, por lo que el promedio es 21.6666666666. El último paso es cerrar el archivo. Primero inicializamos la `suma` a 0 y el número de renglones, que es la palabra `cuenta`, leídos a 0.

```
suma = 0
cuenta = 0
```

Después de abrir el archivo tenemos que leer los números renglón por renglón, lo que hacemos con un ciclo `for`:

```
for renglon in mi_archivo:
```

Luego convertimos a entero y lo sumamos a la suma e incrementamos el contador cuenta de cuantos renglones se han leido con:

```
dato = int(renglon)
suma = suma + dato
cuenta += cuenta
```

Salimos del ciclo y dividimos el valor de suma entre cuenta para obtener el promedio que es un número real:

```
promedio = float(suma/cuenta)
```

Terminamos cerrando el archivo y desplegando el promedio:

```
mi_archivo.close( )
print( "Promedio", promedio)
```

El archivo completo es:

```
mi_archivo = open("enteros.txt")
suma = 0
cuenta = 0
for renglon in mi_archivo:
    dato = int(renglon)
    suma = suma + dato
    cuenta += 1
promedio = float(suma/cuenta)
mi_archivo.close( )
print( "Promedio", promedio)
```

El resultado que se obtiene en el IDLE de Python es 21.666666666666668.

Ejemplo 9.8

Lectura de datos alfanuméricos y numéricos

Vamos a suponer que tenemos un archivo llamado `temperatura.txt` que contiene información de los promedio de las temperaturas Celsius mensuales en el año 2014 de El Cairo, Egipto:

Ene	13.8
Feb	15.2
Mar	17.4
Abr	21.4
May	24.7
Jun	27.3
Jul	27.9
Ago	27.9
Sep	26.3
Oct	23.7
Nov	19.1
Dic	15

Deseamos leer este archivo y desplegarlo en la pantalla del IDLE. Claramente vemos que esta información se puede representar como un diccionario en Python. La ciudad corresponde a la `clave` y la temperatura promedio al `valor`. Supongamos que la información se encuentra en el archivo `temperaturas.txt`. Como siempre primero abrimos el archivo. Para leer cada uno de los datos de un renglón necesitamos partir el renglón en sus componentes que son el mes y el valor de la temperatura y asignarlos a la clave y al valor del diccionario. Para esto usamos la instrucción `renglon.split` donde la primera parte es el mes y la última es la temperatura. Para leer todos los renglones en el archivo usamos un ciclo `Para`. El archivo se forma de la siguiente manera:

Abrimos el archivo e inicializamos la suma de temperaturas con:

```
archivo = open("temperatura.txt", "r")
sumaT = 0
```

Iniciamos el ciclo `Para`:

```
for renglon in archivo:
```

Partimos el renglón en sus componentes, que en este caso son solamente dos:

```
datos = renglon.split( )
```

La primera palabra la asignamos al mes y la última al valor de la temperatura:

```
mes = datos[0]
temperatura = datos[-1]
```

Formamos la palabra para desplegarla:

```
palabra = mes + " " + temperatura
print (palabra)
```

Sumamos la temperatura leída a la suma acumulada pero antes la convertimos a real:

```
sumaT += float(temperatura)
```

Cerramos el archivo:

```
archivo.close( )
```

Calculamos el promedio y lo desplegamos:

```
promedio = sumaT/12
print("El promedio es: " promedio)
```

El archivo completo es:

```
archivo = open( "temperatura.txt", "r")
sumaT = 0
for renglon in archivo:
    datos = renglon.split( )
    mes = datos[0]
    temperatura = datos[-1]
    palabra = mes + " " + temperatura
    sumaT += float(temperatura)
archivo.close( )
promedio = sumaT/12
print("El promedio es: ", promedio)
```

9.5 Lectura y escritura de datos en Excel

Desde Python también podemos leer datos creados en Excel. Uno de los formatos para guardar un archivo de Excel es **csv** que quiere decir **comma-separated-values** (valores separados por comas). Para que Python pueda leer datos en este formato se debe importar la biblioteca `csv` que contiene la función `csv.reader` para leer de archivos y la función `csv.writer` para escribir en archivos en este formato.

Ejemplo 9.9

Lectura de datos en Excel

Vamos a suponer que tenemos un archivo llamado `datosExcel.csv` que contiene información como se muestra en la figura 9.1.

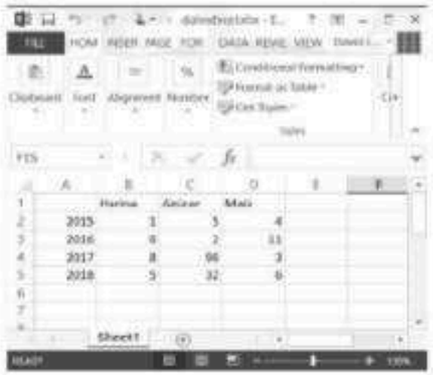


Figura 9.1 Archivo datosExcel.csv

En este archivo de Excel se encuentran en el primer renglón los nombres de los productos, y en los demás renglones abajo de estos nombres se encuentran los precios promedio de ellos en los cuatro trimestres del año. Lo que deseamos hacer es leer los datos del archivo y realizar el promedio de cada producto. Al terminar deseamos escribir en otro archivo Excel, con nombre `promedio_anual.csv`, el nombre de cada producto y el promedio de precios de cada producto.

Para leer este archivo tenemos que realizar lo siguiente:

1. Importar la biblioteca `csv`. Esta biblioteca sirve para abrir archivos con la extensión `csv`. Esto lo hacemos con la instrucción:

```
import csv
```

2. Abrir el archivo con:

```
archivo = open( "datosExcel.csv", "r")
```

3. Inicializar la lista donde vamos a almacenar cada uno de los renglones. El resultado será una lista de listas. Le damos el nombre de tabla y la inicializamos con:

```
tabla = [ ]
```

4. Usando un ciclo `Para` leer los renglones. Cada renglón leído se añade a la lista que le hemos llamado `tabla`. Esto lo hacemos con la instrucción `append` de la siguiente manera:

```
for renglon in csv.reader(archivo):
   tabla.append(renglon)
```

5. Al terminar de leer el archivo lo cerramos con:

```
archivo.close( )
```

6. Para desplegar los datos leídos, ya que se trata de una lista de listas que tiene la forma de una tabla usamos la instrucción `pprint` que significa pretty print y que se usa para tablas como es nuestro ejemplo. Se tiene que importar la biblioteca `pprint` que puede ir como encabezado del programa. Entonces usamos:

```
import pprint
pprint.pprint(tabla)
```

El algoritmo completo es el siguiente:

```
import csv
```

```
import pprint
archivo = open( "datosExcel.csv", "r")
tabla = [ ]
for renglon in csv.reader(archivo):
   table.append(renglon)
archivo.close( )
pprint.pprint(tabla)
```

Ahora lo que nos falta es realizar la suma de las cantidades de cada año. Para realizar esto vemos que el primer renglón (renglón 0) corresponde a los nombres de las materias y la primera columna (columna 0) a los años. Por lo tanto, para realizar la suma de cada una de las cantidades por año, necesitamos empezar por el renglón 1 y la columna 1. Los pasos a seguir son los siguientes:

1. Crear el renglón donde vamos a escribir los resultados:

   ```
   renglon = [0]*len(tabla[0])
   ```

2. El primer elemento de la columna de resultados debe indicar la palabra `Promedios` lo que hacemos con:

   ```
   renglon[0] = 'Promedios'
   ```

3. Con un ciclo `for` recorremos las columnas de 1 a longitud del renglón y con otro ciclo `for` anidado realizamos la suma de los elementos de cada columna. Antes de sumar tenemos que convertir los datos de cadena a flotante. Para obtener el promedio dividimos el resultado de la suma entre el número de renglones -1. El promedio se almacena en la columna correspondiente del vector renglón. Todo esto lo hacemos con:

   ```
   for c in range(1, len(tabla[0])):
      s = 0
      for r in range(1, len(tabla)):
         s += float(tabla[r][c])
      renglon[c] = s/(len(tabla)-1)
   ```

4. Finalmente desplegamos el renglón (ya fuera de los ciclos anidados):

```
print(renglon)
```

Esta parte es entonces:

```
renglon = [0.0]*len(tabla[0])
renglon[0] = 'Promedios'
for c in range(1, len(renglon)):
   s = 0
   for r in range(1, len(tabla)):
      s += float(tabla[r][c])
   renglon[c] = s/3
print(renglon)
```

Finalmente, escribimos `tabla[0]` y `renglón` en otro archivo llamado `promedio_anual.csv`. Esto lo hacemos con:

```
salida open("promedio_anual.csv", "w")
for k in range(0, len(tabla[0])):
salida.write(str(tabla[0][k]))
salida.write(",")
salida.write(" \n")
salida.write(str(renglon))
salida.close( )
```

Ejemplo 9.10

Escritura de datos en Excel

Para escribir en un archivo con el formato csv usamos la instrucción

```
csv.writer
```

Supongamos que deseamos escribir el arreglo:

```
a = [[1, 2, 3],
     ["café", "azúcar", "crema"],
     [12.50, 18.01, 16.54]]
```

Tenemos que seguir los siguientes pasos:

1. Importar la biblioteca `csv`:

   ```
   import csv
   ```

2. Crear el archivo para escribir `ejemplo2.csv` y definir un identificador:

   ```
   iden = open("ejemplo2.csv", "w")
   ```

3. Crear un identificador para escribir en nuestro archivo con la instrucción `writer` de la biblioteca `csv`:

   ```
   escribir = csv.writer(iden)
   ```

4. Escribir los renglones con un ciclo Para y con la instrucción `writerow`:

   ```
   for renglon in a:
        escribir.writerow(renglon)
   ```

5. Finalmente cerrar el archivo:

   ```
   iden.close()
   ```

El programa completo es:

```
import csv
a = [[1, 2, 3],
["café", "azúcar", "crema"],
[12.50, 18.01, 16.54] ]
iden = open("ejemplo2.csv", "w")
escribir = csv.writer(iden)
for renglon in a:
     escribir.writerow(renglon)
iden.close()
```

El resultado que se obtiene se muestra en la figura 9.2.

Figura 9.2 Contenidos del archivo `ejemplo2.csv`.

9.6 Instrucciones de Python del Capítulo 9

Instrucción	Descripción
`close`	Cierra un archivo.
`csv.reader`	Lee archivos csv.
`csv.writer`	Escribe archivos csv .
`import csv`	Importa la biblioteca de `csv`.
`import pprint`	Importa la biblioteca de `pprint`.
`open`	Abre un archivo para lectura o escritura.
`pprint`	Escribe tablas.
`with`	Abre archivos y los cierra al terminar.
`write`	Escribe en un archivo.
`writerow`	Escribe un renglón completo en un archivo.
`writelines`	Escribe renglones.

9.7 Conclusiones

En este capítulo hemos cubierto la manera de almacenar tanto datos de entrada como de salida en un archivo que se puede leer o escribir desde un programa en Python. Los datos del archivo pueden haber sido creados en otro programa. Así mismo hemos cubierto su realización en lenguaje Python. A través de ejemplos desarrollamos y se muestran las distintas técnicas de lectura y escritura de archivos de datos, de texto y archivos de Excel.

9.8 Ejercicios

1. Genere una lista de números enteros aleatorios entre 0 y 10 y guárdelos en una lista que a su vez se guarda en un archivo nuevo llamado `aleatorio.dat`. Después abra el archivo desde un editor de textos como el WordPad o Word para comprobar.
2. Guarde en un archivo los nombres de los países de América del Sur junto con su población. Consulte una enciclopedia para la población. Guarde los datos en un archivo `paises.dat`. Cheque su resultado en Word.
3. Diseñe un algoritmo que lea un archivo llamado `reales.txt` de números reales y los sume. El resultado debe almacenarse en un archivo llamado `suma.dat` que contenga una lista con los números reales originales y la suma calculada debe ser el último número de la lista.
4. La lista del ejemplo anterior almacenada en el archivo `suma.dat` debe convertirse a `suma.csv` para poder ser leída desde Excel.
5. Resuelva el problema de la ecuación de segundo grado del Capítulo 2 calculando el discriminante en una función que recibe los coeficientes a, b, c de un archivo llamado `coeficientes`.dat. Dependiendo del valor del discriminante, diseñe tres funciones que calculen las raíces de la ecuación dependiendo si el discriminante es positivo, cero o negativo. Una cuarta función debe imprimir las raíces en la pantalla de la computadora y almacenarlas en un archivo llamado `raices.dat`. Este archivo debe llevar el encabezado: "Raíces de la ecuación de segundo grado con coeficientes a, b, c."

Capítulo 10
Programación orientada a objetos

10.1 Introducción
10.2 Conceptos asociados a la POO
10.3 Primera clase en Python
10.4 Creación de la clase NumeroComplejo
10.5 Declaración y uso de Setters y Getters
10.6 Sobreescritura de operadores
10.7 Herencia
10.8 Sobreescritura de métodos
10.9 Ejemplos
10.10 Instrucciones de Python del Capítulo 10
10.11 Conclusiones
10.12 Ejercicios

Un buen diseño agrega valor más rápido de lo que agrega costo.

Thomas C. Gale

Objetivos

En este capítulo damos una breve descripción de la manera de escribir programas usando el paradigma orientado a objetos en Python, en particular se incluyen ejemplos que ilustran la creación y uso de clases a través de objetos así como la implementación de la encapsulación, herencia y polimorfismo a distintos problemas.

10.1 Introducción

En este capítulo damos una descripción inicial del paradigma de programación orientada a objetos **POO** (Object Oriented Programming: OOP) como se encuentra implementado en Python. Una gran cantidad de libros de texto han sido dedicados a la enseñanza de la programación orientada a objetos y sus posibles aplicaciones. En esta sección solamente nos enfocamos en la presentación de las características fundamentales de la POO en Python, así como se proporcionan los detalles para poder empezar a programar utilizando este paradigma.

10.2 Conceptos asociados a la POO

La programación orientada a objetos tiene sus inicios en la década de los 60 con la aparición del lenguaje de programación Simula 67, pero este paradigma se hizo popular hasta la década de los 80 debido a la influencia del lenguaje de programación C++. Actualmente, la POO es de las más utilizadas a nivel mundial. Podemos encontrar lenguajes de programación que están basados completamente en éste, como Java o lenguajes que tienen un amplio soporte a las características de POO como Python, Ruby, PHP, etc. Entre las principales características de este paradigma se encuentra la representación y el uso de unidades mínimas de código llamados objetos o entidades (abstracciones de elementos del mundo real), los cuales, se comunican entre sí con el propósito de resolver uno o varios problemas de una manera más clara y

sencilla. Tomando en cuenta lo anterior, la programación orientada a objetos requiere un entendimiento claro y preciso de los siguientes términos:

1. Clase
2. Objeto
3. Atributo
4. Método
5. Encapsulación
6. Herencia
7. Polimorfismo

Una **clase o instancia** permite que la creación y el manejo de los objetos sea sencillo. En términos más simples, una clase es una plantilla o plano (blueprint) para la creación de objetos. Esta nos dice cuales deben de ser las características y las acciones que deben hacer cada objeto que sea creado a partir de ésta.

Un **objeto** es el elemento fundamental en la programación orientada a objetos, puede representar un objeto real como sería una cuenta bancaria, un cliente o una transacción o puede representar un elemento abstracto como una ecuación, una fracción o un número complejo. Cada objeto tiene un conjunto de características que lo definen así como un conjunto de acciones que determinan su comportamiento con respecto a otros objetos.

Un **atributo** es el valor que almacena internamente el objeto (las características de un objeto) y que pueden ser datos primitivos o incluso otros objetos. Es importante destacar que a pesar de que cada objeto creado a partir de una clase comparte los mismos atributos y acciones que otros objetos, tiene un estado diferente determinado por los valores que tengan en un instante de tiempo determinado.

Un **método** es un grupo de instrucciones asociadas al objeto a las que se les ha dado un nombre específico. Cuando un método es invocado se ejecutan las instrucciones del programa. Los métodos de un objeto definen como se comporta y cuáles son las acciones que se pueden realizar sobre él o hacia otro objeto. La principal diferencia entre una función utilizada en la programación procedural y un método es que los métodos pueden acceder y modificar a los atributos que contiene el objeto, mientras que una función no lo puede hacer.

Encapsular un objeto quiere decir que puede almacenar información y trabajar con ella de tal manera que los cambios de estado solamente puedan ser realizados por los métodos que pertenecen al objeto y que ningún otro objeto o función externa puedan hacerlo. La principal característica de la encapsulación es la de abstraer al usuario del objeto de los cambios internos que puedan tener los atributos, así como proteger la integridad del objeto no permitiendo que nadie pueda alterar el flujo predeterminado en los métodos.

La **herencia** es una técnica de la POO en la cual, se puede crear una clase nueva a través de una clase existente, por medio de la declaración de nuevos métodos y atributos que extienden la funcionalidad de la clase de la que se hereda. Los objetos de la nueva clase heredan las propiedades y el comportamiento de todas las clases a las que pertenecen, es decir, pueden usar las características y el comportamiento heredado sin tener que volver a implementar la funcionalidad.

El **polimorfismo** es la característica de la POO que permite a los valores de diferentes tipos de datos, ser manejados usando una interfaz uniforme. El mismo método puede ser usado por varias clases, pero de forma diferente.

En la figura 10.1 se muestran todas las definiciones anteriores utilizando un ejemplo basado en la creación de una clase `Computadora`, de la cual se crean o instancian 2 objetos llamados `Laptop` y `Servidor`, cada uno con su conjunto de atributos y métodos.

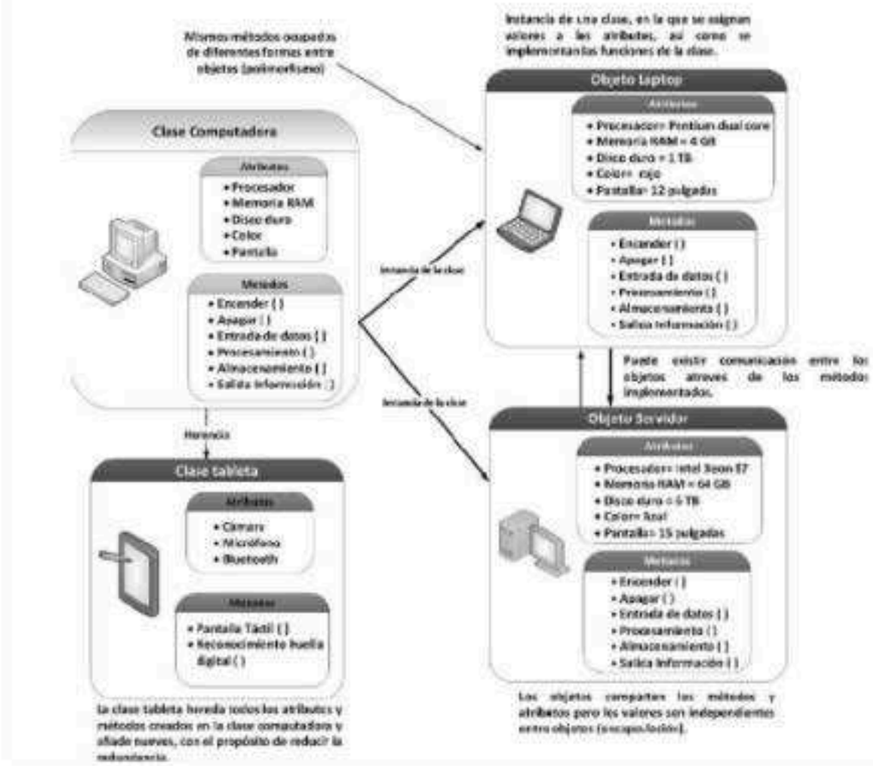


Figura 10.1 Conceptos de la Programación Orientada a Objetos.

Los objetos creados pueden interactuar entre sí por medio de sus métodos, los cuales funcionan como mensajes que se pueden enviar o recibir. Por otra parte, se crea una nueva clase llamada `Tableta`, la cual hereda todos los atributos y métodos creados en la clase Computadora y además añade nuevos atributos y métodos propios del dispositivo (y que no comparte con una computadora común). En las siguientes secciones se mostrará como implementar este tipo de clases y relaciones utilizando el lenguaje de programación Python.

10.3 Primera clase en Python

Crear una clase en Python es muy sencillo y práctico, lo único que se debe hacer es usar la palabra reservada `class` seguida del nombre de una clase, tomando en cuenta que por convención el nombre debe empezar con una letra mayúscula (no debe empezar con un número o un caracter especial). Como ejemplo crearemos una clase llamada `Hola` cuyo único propósito es el de mostrar un mensaje de saludo personalizado:

```
# Inicio de la clase
class Hola:
    # Atributos de la clase
    Nombre = " "
    Apellido = " "
    # Métodos de la clase
    # Constructor de la clase
    def __init__(self, nombre, apellido):
        # Referencia a un atributo de la clase (self.Nombre)
        print ("Constructor de la clase")
        self.Nombre = nombre
        self.Apellido = apellido
    def saludar(self):
        print ("Hola " + self.Nombre + " " + self.Apellido)
# Fin de la clase
```

Del ejemplo anterior podemos observar que los métodos (`saludar` e `init`) y los atributos (`Nombre` y `Apellido`) se combinan en una clase llamada `Hola`, de la cual se pueden crear múltiples objetos utilizando la siguiente instrucción:

```
# Creación de un objeto asociado a la clase
primerObjeto = Hola("Juan", "Perez")
```

Cuando se declara un objeto en Python implícitamente se hace uso de un método especial llamado `init`, el cual es el constructor de la clase que tiene como tarea la inicialización de los atributos y es llamado cada vez que creamos un nuevo objeto. En el caso de nuestra primera clase, el constructor espera que le enviemos tres atributos: el nombre (tipo string), el apellido (tipo string) y una referencia al mismo objeto llamada `self`, la cual es la forma en Python de poder acceder a los atributos del objeto creado por medio de la clase. Dentro de una clase cada método debe tener como primer parámetro una referencia al objeto que lo está llamando, de esta manera aseguramos que los cambios que hagamos dentro de los atributos sólo afecten al objeto que se está ocupando y no a otros objetos creados a partir de la misma clase. Para poder utilizar los métodos definidos dentro de la clase se hace uso del operador punto (.) de la forma siguiente:

```
# Uso de un método asociado al objeto
primerObjeto.saludar( )
```

Como se puede ver, el atributo `self` es auto referenciado por el objeto, por lo que no es necesario enviar ningún atributo adicional. En la figura 10.2 se muestra la ejecución del programa completo.

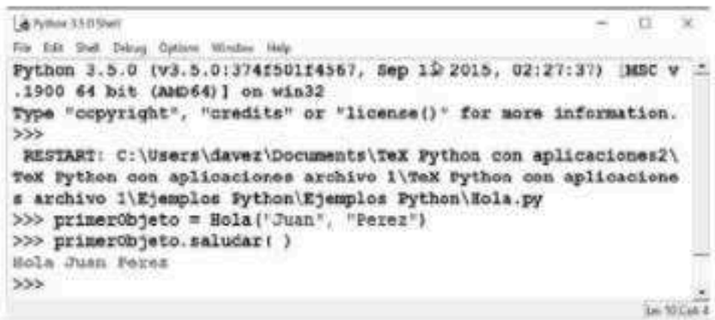


Figura 10.2 Ejecución de la primera clase en Python.

10.4 Creación de la clase NumeroComplejo

Con el propósito de profundizar a fondo acerca de las características especiales de la POO en Python, crearemos un ejemplo que permitirá el manejo de números complejos, donde mostraremos el uso de la encapsulación, el polimorfismo y los métodos de acceso a una clase, con tal propósito iniciamos la definición de una nueva clase llamada NumeroComplejo, en la cual declararemos la parte real e imaginaria del número así como el constructor que inicializará los valores y una función que lo muestre en pantalla de la siguiente manera:

```
# Inicio de la clase
class NumeroComplejo:
    # Atributos de la clase
    ParteReal = 0
    ParteImaginaria = 0
    # Métodos de la clase
    # Constructor de la clase
    def __init__(self, real, imaginaria):
      self.ParteReal = real
      self.ParteImaginaria = imaginaria
    # Imprimir número complejo
    def imprimirNumero(self):
      print(str(self.ParteReal) + " + i*" str(self.ParteImaginaria) )
# Fin de la clase
```

Para poder acceder a esta clase creamos un objeto, el cual inicializamos con los valores del número complejo e imprimiremos su valor en pantalla de la manera siguiente:

```
# Creación de un objeto asociado a la clase
primerNumero = NumeroComplejo(12.0, 4.0)
print("número complejo")
# Uso de un método asociado al objeto
primerNumero.imprimirNumero( )
```

Antes de continuar podemos observar que al igual que con la clase `Hola`, es necesario crear un constructor para inicializar la clase `NumeroComplejo`. En Python no siempre se debe inicializar el constructor de manera explícita ya que muchos de los elementos presentes se inicializan de manera implícita. Aunque no nos demos cuenta, cuando usamos una lista o declaramos una cadena de texto internamente estamos creando un objeto, el cual inicializamos con valores predeterminados por el lenguaje. Lo anterior es uno de los elementos más importantes de Python, porque implícitamente todos los elementos son objetos y por ende tienen asignados atributos y metodos que hacen que la programación sea mucho más dinámica y fácil de implementar.

10.5 Declaración y uso de Setters y Getters

Como mencionamos anteriormente, la encapsulación se refiere a la delimitación del acceso a determinados métodos y atributos de los objetos en una clase. Esta propiedad es una de las más importantes del paradigma, ya que permite crear una interfaz segura entre el usuario y el código interno de la clase, para ello, se crean un conjunto de funciones especiales llamadas **Setters y Getters**, las cuales su único propósito es el de establecer (set) u obtener (get) los valores de los atributos en una clase. Para la clase `NumeroComplejo` se crean las siguientes funciones:

```
# Cambiar la parte real (setter)
def cambiarParteReal(self, real):
    self.ParteReal = real

# Obtener la parte real (getter)
def obtenerParteReal(self):
    return self.ParteReal

# Cambiar la parte imaginaria (setter)
def cambiarParteImaginaria(self, imaginaria):
    self.ParteImaginaria = imaginaria

# Obtener la parte imaginaria (setter)
def obtenerParteImaginaria(self):
    return self.ParteImaginaria
```

A las funciones `cambiarParteReal` y `cambiarParteImaginaria` se les conoce como **setters**, ya que permiten inicializar las variables después de que se ha utilizado el constructor. A las funciones `obtenerParteReal` y `obtenerParteImaginaria` se les conoce como **getters**, ya que permiten obtener el estado de la variable en un momento determinado, lo cual puede ser útil para realizar otros cálculos. Como ejemplo de uso de las funciones podemos cambiar los valores de un objeto de tipo `NumeroComplejo` como se muestra en el siguiente código y en la figura 10.3:

```
# Creación y uso de un objeto asociado a la clase NumeroComplejo.
primerNumero = NumeroComplejo(12.0, 4.0)
print("número complejo versión inicial")
primerNumero.imprimirNumero( )
print("número complejo modificado (cambiar valor de la parteReal)")
primerNumero.cambiarParteReal(25.0)
primerNumero.imprimirNumero( )
print("Parte imaginaria en otro cálculo (parteImaginaria + 5.0)")
print( (primerNumero.obtenerParteImaginaria( ) ) + 5.0 )
```

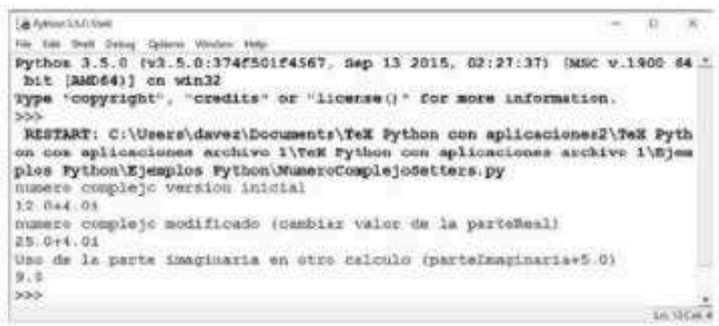


Figura 10.3 Ejecución de la clase NumeroComplejo con Setters y Getters

Tomando en cuenta el uso de setters y getters, se puede ver que son la manera de abstraer a la clase de los usuarios finales, pero hablando estrictamente de la visibilidad de los objetos, se puede decir que en Python a diferencia de otros lenguajes todos los elementos son públicos, por lo que si deseamos asegurar un encapsulamiento total, debemos de hacer uso de los módulos y paquetes en Python, los cuales dividen en espacios de nombres las clases y ayudan a crear una jerarquía de uso de clases.

10.6 Sobreescritura de operadores

Una característica clave del paradigma de orientación a objetos es la redefinición de métodos existentes con el propósito de extender la funcionalidad de un método (herencia) o simplificar las operaciones (sobrecarga de operadores) de los objetos en una clase. En esta sección, en específico usamos la clase `NumeroComplejo` a la cual añadimos las dos operaciones básicas de números complejos, la suma y la resta como se muestra en el siguiente código:

```
#Suma de 2 números complejos
def numeroComplejoSuma(self, numero):
    self.ParteReal = self.ParteReal + numero.ParteReal
    self.ParteImaginaria = self.ParteImaginaria + numero.ParteImaginaria

#Resta de 2 números complejos
def numeroComplejoResta(self, numero):
    self.ParteReal = self.ParteReal - numero.ParteReal
    self.ParteImaginaria = self.ParteImaginaria - numero.ParteImaginaria
```

Para poder realizar alguna de las dos operaciones basta con la creación de dos objetos de tipo `NumeroComplejo`, para posteriormente añadir a un objeto la suma de otro por medio del método `numeroComplejoSuma` como se muestra en el siguiente código y en la figura 10.4:

```
# Creación de un objeto asociado a la clase
primerNumero = NumeroComplejo(12.0, 4.0)
print("Primer número complejo")
# Uso de los métodos asociados al objeto
primerNumero.imprimirNumero( )
print("Segundo número complejo")
segundoNumero = NumeroComplejo(8.0, 4.5)
segundoNumero.imprimirNumero( )
print("Suma de ambos números complejos")
primerNumero.numeroComplejoSuma(segundoNumero)
primerNumero.imprimirNumero( )
```

Figura 10.4 Ejecución de la clase `NumeroComplejo` con suma y resta de números.

Ahora como se puede notar, las operaciones no son tan naturales como si hiciésemos la suma de la forma `numero1 + numero2`. Para poder resolver esto, utilizamos la sobrecarga de operadores, donde creamos dos funciones especiales que redefinan el comportamiento de la operación suma y la operación resta de la siguiente manera:

```
# Suma de 2 números complejos sobrescribiendo el operador +.
 def __add__(self, numero):
   self.ParteReal = self.ParteReal + numero.ParteReal
   self.ParteImaginaria = self.ParteImaginaria + numero.ParteImaginaria
# Suma de 2 números complejos sobrescribiendo el operador -.
def __sub__(self, numero):
   self.ParteReal = self.ParteReal - numero.ParteReal
   self.ParteImaginaria = self.ParteImaginaria - numero.ParteImaginaria
```

Como se puede notar al igual que el constructor, los dos métodos anteriores son tratados de forma especial, ya que redefinen el comportamiento de las operaciones establecidas en Python (la suma y la resta), en el caso del constructor redefine al método implementado en la clase padre objeto (véase la sección de herencia), estos métodos tienen como distintivo el uso de dos guiones bajos, los cuales indican que son métodos especiales y que deben ser tratados como tal[1]. Para realizar la suma de los objetos creados previamente, ahora solamente basta con implementar el siguiente código con el resultado mostrado en la figura 10.5:

[1]Estos métodos son idóneos para implementarse sólo en clases que crean nuevos tipos de datos, ya sean numéricos o alfanuméricos

```
print("Suma de ambos números complejos: sobrecarga")
# Suma de dos números complejos de manera natural
primerNumero + segundoNumero
primerNumero.imprimirNumero( )
```

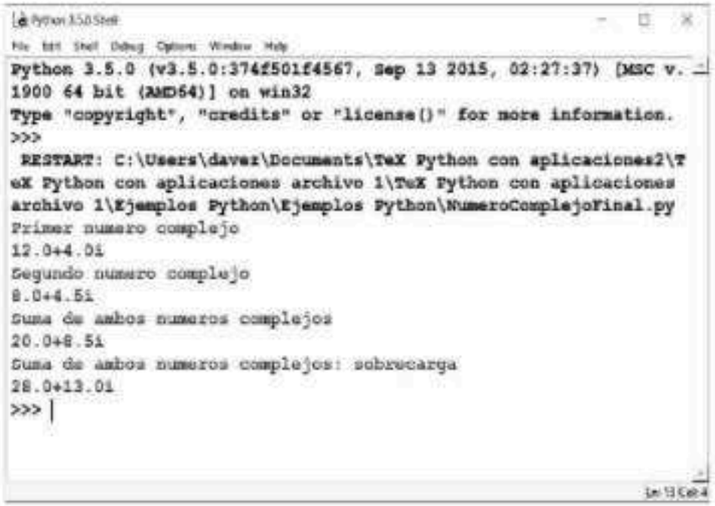


Figura 10.5 Ejecución de la clase NumeroComplejo con suma y resta de n úmeros, utilizando sobrecarga.

Es importante hacer notar del ejemplo anterior que no se están almacenando los resultados de las sumas en nuevos objetos, por lo que el resultado final es consecuencia de las dos sumas anteriores (con y sin sobrecarga).

10.7 Herencia

La herencia es la propiedad del paradigma de orientación a objetos que permite reutilizar y extender clases previamente creadas, lo cual permite la reutilización de código, en el sentido de que ya no es necesario escribir métodos o atributos que son creados en las clases de las cuales se hereda. En Python existen dos tipos de herencia, la simple y la múltiple. En el caso de esta sección abordaremos la herencia simple, ya que es la más fácil de entender, así como la que más se utiliza en la mayoría de aplicaciones. La

herencia simple en Python consiste en que una nueva clase hereda u obtiene métodos y atributos previamente creados de otra clase a la que llamaremos clase padre. La herencia se puede ver como una jerarquía de clases donde, las clases hijo heredan de una clase padre, pero puede ser heredada a su vez por múltiples clases hijo. Es importante notar que todas las clases heredan de una clase base a la cual llamaremos clase objeto, la que cuenta con los métodos y atributos necesarios para crear objetos. Esta clase está implícita en la creación de todas las clases y se pueden acceder a todos los atributos que ella tiene.

Para mostrar la herencia creamos una nueva clase llamada `Operacion`, la cual implementa las características que comparten las operaciones aritméticas de suma y multiplicación. La clase `Operacion` implementa el siguiente código:

```
# Inicio de la clase padre Operación
class Operacion:

    # Atributos de la clase
    valor1 = 0
    valor2 = 0

    # Métodos de la clase
    # Constructor de la clase
    def __init__(self, numero1, numero2):
        self.valor1 = numero1
        self.valor2 = numero2

    # Obtener valor1 (getter)
    def obtenerValor1(self):
        return self.valor1

    # Obtener valor2 (getter)
    def obtenerValor2(self):
        return self.valor2

    # Cambiar valor1 (setter)
    def cambiarValor1(self, nuevoValor):
        self.Valor1 = nuevoValor
```

```
    # Cambiar valor2 (setter)
    def cambiarValor2(self, nuevoValor):
        self.Valor2 = nuevoValor

    # Imprimir número
    def imprimirValor(self, numero):
        print(numero)
# Fin de la clase padre Operación
```

En la implementación de la clase se crean los setters y getters necesarios para acceder a los atributos de la operación, así como se declara el constructor, que como hemos revisado antes, sirve para inicializar los atributos la primera vez que se crea un objeto. Finalmente se crea un método que imprime a cualquiera de los números implícitos en la operación. Para crear la clase `Suma`, sin tener que volver a definir los elementos de la clase `Operacion` especificamos que la clase `Suma` hereda de la clase padre `Operacion` de la siguiente forma:

```
# Inicio de la clase hijo Suma
class Suma(Operacion):

    '''Se llama al constructor de la clase
    padre para inicializar los números.'''

    def __init__(self, numero1, numero2):
        super().__init__(numero1, numero2)

    '''Se agrega un método nuevo a la clase
    suma(que no está en la clase padre)'''
    def sumar(self):
        self.imprimirValor(self.valor1 + self.valor2)
# Fin de la clase hijo Suma
```

Para especificar que una clase hereda de otra escribimos el nombre de la clase padre entre paréntesis después del nombre de la clase hijo. Python automáticamente buscará esa clase ya sea en el mismo archivo o en el espacio de nombres asignado por Python. A diferencia de otras clases de tipo constructor, en la clase `Suma` se pasan los atributos a la clase padre a través de la palabra reservada `super`, la cual sirve de vínculo entre la clase padre y la clase hija. Es importante hacer notar que si la clase hija no necesitara

pasar ningún parámetro a la clase padre, no sería necesario incluir la palabra reservada `super`, ya que cuando se crea la clase `Suma` automáticamente llama al constructor de la clase padre. Finalmente en el método `sumar` podemos notar que es posible acceder al método `imprimirValor` de la clase padre, lo cual ayuda a reducir el número de líneas de código. A continuación presentamos el código para crear el objeto de la clase `Suma` y su ejecución en la figura 10.6:

```
# Creación de objeto de la clase hijo
suma1 = Suma(10, 5)
print("Operando 1 de la suma")
suma1.imprimirValor(suma1.obtenerValor1( ) )
print("Operando 2 de la suma")
suma1.imprimirValor(suma1.obtenerValor2( ) )
print("Resultado de la suma")
suma1.sumar( )
```

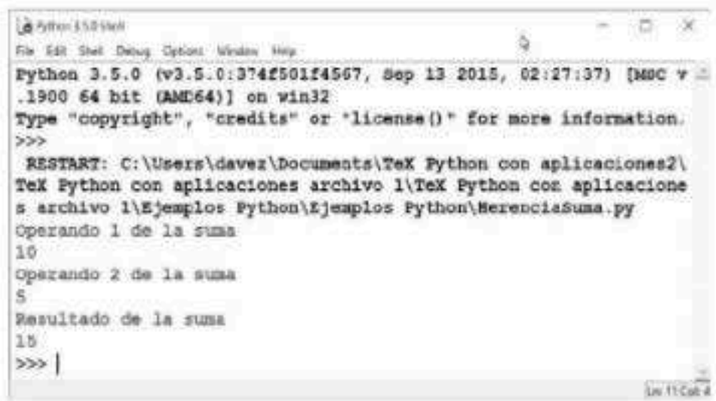


Figura 10.6 Ejecución de la clase `Suma`.

10.8 Sobreescritura de métodos

Los métodos que una clase hereda de otra clase se pueden usar por ella ya que hereda todos los métodos definidos en la clase padre, junto con los definidos en ella misma. Además, la clase puede redefinir un método de la clase padre, creando un nuevo método con la misma firma (nombre, argumentos de entrada y salida), de manera que los métodos de la clase padre son redefinidos o remplazados. A esto se le conoce como sobreescritura de métodos. En el caso de nuestro ejemplo de operaciones aritméticas crearemos una nueva clase llamada `Multiplicacion` la cual heredará de la clase `Operacion` y redefinirá el método `imprimirValor` de la siguiente manera:

```
#Inicio de la clase hijo Multiplicación
class multiplicacion(Operacion):
    "'Se llama al constructor de la clase
    padre para inicializar los números."'

    def __init__(self, numero1, numero2):
        super().__init__(numero1,numero2)

    def multiplicar(self):
        self.imprimirValor(self.valor1*self.valor2)

    # Se sobrescribe un método de la clase padre
    def imprimirValor(self,numero):
        print("el número es: " + str(numero))

#Fin de la clase hijo Multiplicación
```

Como se puede observar el método `imprimirValor` ahora envía a pantalla un mensaje personalizado de la operación de la multiplicación. A continuación presentamos el código para crear el objeto de la clase `Multiplicacion` y su ejecución en la figura 10.7:

```
multiplicacion1 = multiplicacion(10, 5)
multiplicacion1.imprimirValor(multiplicacion1.obtenerValor1( ))
multiplicacion1.imprimirValor(multiplicacion1.obtenerValor2( ))
print ("Resultado de la multiplicación:")
multiplicacion1.multiplicar( )
```

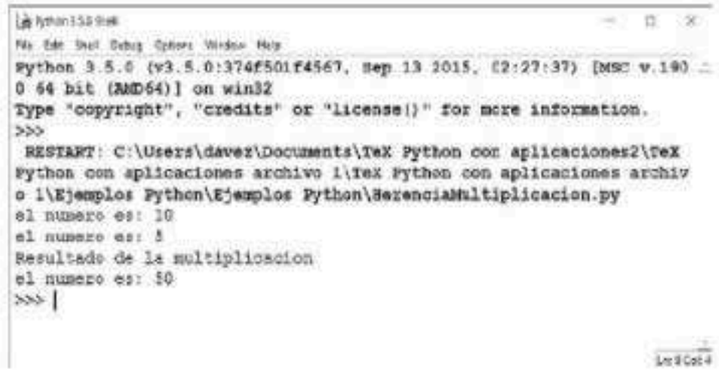


Figura 10.7 Ejecución de la clase Multiplicación.

Tomando en cuenta las clases `Multiplicacion` y `Suma`, se puede ver que ambas clases pueden acceder y modificar al método `imprimirValor`, lo cual muestra claramente la propiedad de **polimorfismo** en las clases, ya que los objetos de ambas clases responden a métodos con el mismo nombre, pero con implementaciones diferentes.

10.9 Ejemplos

En esta sección se muestran varios ejemplos de clases que ilustran las principales características de Python acerca del paradigma de programación orientado a objetos. En cada uno de los ejemplos se describe como crear la clase que implemente la funcionalidad, así como posteriormente cómo crear un objeto de esa clase, el cual llame a los atributos y métodos de la clase.

10.9.1 Creación y movimientos de una cuenta bancaria

En este ejemplo se muestra la manera de implementar los movimientos asociados con una cuenta bancaria, utilizando las herramientas de orientación a objetos ofrecidas por Python. Los movimientos que se tienen que realizar son creación de la cuenta, depósitos, retiros y la visualización del saldo. Tomando en cuenta lo anterior, lo primero que haremos es la creación de una clase `cuentaBancaria`, la cual tendrá 2 atributos asociados a cada uno de los clientes:

```
# Inicio de clase
class CuentaBancaria:

    #Variables de la clase
    saldo = None
    nombre = None
```

El primer método que debemos crear en la clase es el constructor, que es el encargado de crear la cuenta y asignarle un valor al saldo inicial y al nombre del dueño:

```
# Constructor de la clase
def __init__(self,saldo,nombre):
    # Revisar si las variables de entrada son del tipo especifico
    if ((type(saldo) == int or type(saldo) == float) and
        isinstance(nombre,str)):

        self.saldo = saldo
        self.nombre = nombre
        self.mostrarInformacion( )

    # Si no son del tipo específico, se ponen valores por default
    else:
        self.saldo = 0
        self.nombre = "Desconocido"
        self.mostrarInformacion( )
```

Ahora se implementa un método que muestre el nombre del cliente y su saldo, el cual después es utilizado por varios métodos (incluyendo al constructor) para saber el estado de la cuenta:

```
"'Despliega la representación de la cuenta
bancaria como una cadena"'
def mostrarInformacion(self):
    print(self.nombre + " cuenta con: " + str(self.saldo) +
        " de saldo")
```

El paso siguiente es la creación de los métodos para depositar y retirar el saldo. En el primer caso se suma al saldo anterior la cantidad percibida y en la segunda se resta. Debemos considerar también, el caso en que el retiro sea mayor que el saldo:

```
def deposito(self, cantidad):
    if type(cantidad) == int or type(cantidad) == float:
        print("Depósito de " + self.nombre + " por: " + str(cantidad))
        # Incrementar la cantidad de saldo
        self._calcularSaldo(cantidad)
        self.mostrarInformacion( )
    else:
        print("Depósito de " + self.nombre + " por: " + str(cantidad))
        print("las cantidades deben ser números")

def retiro(self, cantidad):
    if type(cantidad) == int or type(cantidad) == float:
        print("Retiro de " + self.nombre + " por: " + str(cantidad))
        cantidadRetirar = cantidad
        if cantidadRetirar > self.saldo:
            print(self.nombre + " No hay saldo suficiente.")
        else:
            # Reducir la cantidad del saldo de la cuenta
            self._calcularSaldo(self.saldo - cantidadRetirar)
            self.mostrarInformacion()
    else:
        print("Retiro de " + self.nombre + " por: " + str(cantidad))
        print("Las cantidades deben de ser números.")
```

Finalmente creamos un método para calcular (getter) el saldo asociado a una cuenta. Es importante hacer notar, que se utiliza un guion bajo en el nombre del método, lo cual indica una convención utilizada por Python para denotar que un método (o atributo) solamente es de uso interno de la clase y no debe ser accedido a través del operador punto por un objeto. A diferencia de otros lenguajes de programación, en Python el acceso a atributos o métodos en una clase no está restringido, es decir, no existen delimitadores por lo que se utilizan distintas convenciones como la mostrada, para señalar que el uso de un método o una clase a través de un objeto puede ser peligroso y llevar a resultados indeseables.

```
def _calcularSaldo(self,cantidad):
    # Obtener el saldo asociado a la cuenta
    self.saldo = cantidad
# Fin de clase
```

A continuación presentamos el código para crear dos objetos de la clase `CuentaBancaria` y su ejecución en la figura 10.8.

```
# Creación de objetos
objeto1 = CuentaBancaria(12, "Pedro")
objeto2 = CuentaBancaria(125, "Juan")
# Manipulación de las funciones de los objetos
objeto2.retiro(89)
objeto1.retiro(12000)
objeto1.deposito("mil doscientos")
```

Figura 10.8 Ejecución de la clase CuentaBancaria.

10.9.2 Cálculo de la media aritmética y la desviación estándar

En este ejemplo se muestra una clase que implementa dos de las técnicas más usadas en la estadística para resumir una colección de datos, la media aritmética y la desviación

estándar. La media aritmética es la relación de la suma de todos los datos con respecto a la cantidad o dicho de otra manera, es el valor que resulta de repartir equitativamente el total entre todos los datos. Por su parte la desviación estándar nos proporciona información sobre cómo están distribuidos los datos alrededor de la media: lo alejados (dispersos) o cercanos que estén de la misma. Las fórmulas asociadas a la media y la desviación se definen a continuacion:

- Media aritmética

$$\overline{X} = \frac{\sum_{i=1}^{n} X_i}{N} \tag{10.1}$$

- Desviación estándar:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (X_i - \overline{X})^2}{N}} \tag{10.2}$$

Donde X_i representa cada uno de los datos y N representa el número total de datos en una colección. Para crear el programa en Python lo primero que se hace es crear la clase `Estadistica` sin constructor (debido a que para este programa no son necesarias variables instancia), así como un método para la media y otro para la desviación de la siguiente manera:

```
# Inicio de clase
class Estadistica:
   def media(self, lista):
    '''Se verifica que cada elemento de la
   lista sea de tipo int o float'''
         if (all((isinstance(x, int) or
                 isinstance(x, float))for x in lista) == True):
            acumulado = 0
            for x in lista:
               acumulado = acumulado + x
            return acumulado/len(lista)
         else:
            print("La función solamente admite números")
            return 0
      def desviacionEstandar(self,lista):
```

```
        if (all((isinstance(x, int) or
                isinstance(x, float)) for x in lista) == True):
          acumulado = 0
          media = self.media(lista)
          for x in lista:
             acumulado = acumulado + ((x - media)**2)
          return math.sqrt(acumulado/len(lista))
        else:
          print("La función solamente admite números.")
          return 0
# Fin de clase
```

Teniendo la clase construida se crea un objeto de la clase y se aplican los métodos de la siguiente manera:

```
# Creación de objetos
operacion = Estadistica( )
# Manipulación de las funciones de los objetos
lista = [ 9, 3, 8, 8, 5.6, 9, 8, 9, 18, 1.442677 ]
print("Lista: " + " , " .join(str(x) for x in lista))
print("Media: " + str(operacion.media(lista)))
print("Desviación estandar: " + str(operacion.desviacion(lista)))
```

La ejecución del programa se muestra en la figura 10.9:

Figura 10.9 Ejecución de la clase Estadística.

10.9.3 Operaciones con matrices

En el siguiente ejemplo se crea una clase que implementa dos de las operaciones más populares al momento de trabajar con matrices, la suma y multiplicación de matrices. Para ambas se crean métodos de uso interno para verificar que se puedan realizar las operaciones. El primer paso para la creación del programa es la definición de la clase de la siguiente manera:

```
# Inicio de clase
class OperacionMatrices:
```

Después se define un método interno para revisar que las variables pasadas a través del objeto sean matrices, es decir, que las variables cumplan con distintas propiedades como el uso de listas de dimensión mayor a la unidad o el uso de valores alfanuméricos de la siguiente manera:

```
def _verificarMatriz(self, matriz):
    '''Se verifica que cada fila de la
    matriz sea una lista'''
    if type(matriz) == list and len(matriz) >0:
        '''Se verifica que exista la misma cantidad de
        elementos por fila en la matriz'''
        if len(set(len(x) for x in matriz)) != 1:
            return False
        for x in matriz:
            if type(x) == list and len(x) >0:
                '''Se verifica que todos los elementos sean de
                tipo entero o flotante en la matriz'''
                if (all((isinstance(valor, int) or
                    isinstance(valor, float))for valor in x) != True):
                        return False
            else:
                return False
        return True
    else:
        return False
```

El siguiente método interno revisa que dos variables sean matrices (apoyándose en el método previamente creado) y que además sean de la misma dimensión, con el propósito de ver que las matrices introducidas estén en el formato deseado para la operación suma:

```
def _verificarSuma(self, matriz1, matriz2):
    # Se verifica que ambas estructuras sean matrices
    if ((self._verificarMatriz(matriz1) != True) or
       (self._verificarMatriz(matriz2) != True)):
          return False
    else:
       '''Se verifica que las matrices
       sean de la misma dimensión'''
       if len(matriz1) != len(matriz2):
          return False
       else:
          '''Se verifica que ambas matrices tengan
          la misma cantidad de elementos por fila'''
          for x, y in zip(matriz1, matriz2):
             if len(x) != len(y):
                return False
    return True
```

De manera análoga se define un último método interno que revisa que ambas variables introducidas sean matrices, que tengan la misma dimensión y además el número de columnas de la primera matriz introducida sea igual al número de renglones de la segunda matriz introducida:

```
def _verificarMultiplicacion(self, matriz1, matriz2):
    # Se verifica que ambas estructuras sean matrices
    if ((self._verificarMatriz(matriz1) != True) or
       (self._verificarMatriz(matriz2) != True)):
          return False
    else:
       '''Se verifica que el número de columnas de la matriz 1
       sea igual al número de renglones de la matriz 2'''
```

```
        if len(matriz1[0]) == len(matriz2):
            return True
        else:
            return False
```

El siguiente paso es la implementación de los métodos principales de suma y multiplicación de matrices, los cuales hacen uso de los métodos internos de verificación para comprobar si pueden realizar las operaciones:

```
    # Método que implementa la suma de matrices
    def sumaMatrices(self, matriz1, matriz2):
        # Se verifica que ambas estructuras sean matrices
        if self._verificarSuma(matriz1, matriz2) == False:
            print("No se puede realizar la operación con matrices")
            return [ ]
        else:
            resultado = [ ]
            for x in range(len(matriz1)):
                resultado.append([ ])
                for y in range(len(matriz2[0])):
                    resultado[x].append(matriz1[x][y] + matriz2[x][y])
            return resultado

    # Método que implementa la multiplicación de matrices
    def MultiplicacionMatrices(self, matriz1, matriz2):
        if self._verificarMultiplicacion(matriz1, matriz2) == False:
            print("No se puede realizar la operación con matrices")
            return [ ]
        else:
            resultado = [ ]
            for x in range(len(matriz1)):
                resultado.append( [ ] )
                for y in range(len(matriz2[0])):
                    res = 0
                    for z in range(len(matriz2)):
                        res = res + matriz1[x][z]*matriz2[z][y]
```

```
            resultado[x].append(res)
        return resultado
# Fin de clase
```

Teniendo la clase construida se crea un objeto de la clase y las matrices con las que se probarán los métodos de suma y multiplicación de la siguiente manera:

```
# Creación de un objeto asociado a la clase
operaciones = OperacionMatrices( )

'''Se crean las matrices con las que
se probarán los métodos propuestos'''

lista1 = [ [3,2,1,9],[-1,-3,0,8],[5,7,8,7] ]
print(" Matriz 1:")
print(lista1)

lista2 = [ [9,0,1,3], [-1, 7, 1, 4], [6, 4, 1, 6] ]
print(" Matriz 2:")
print(lista2)

lista3 = [ [3, 2, 1], [-1, -3, 0], [5, 7, 8] ]
print(" Matriz 3:")
print(lista3)

lista4 = [ [9, 0, 1], [-1, 7, 1], [6, 4, 1] ]
print(" Matriz 4:")
print(lista4)
```

Finalmente se llaman a los métodos de suma y multiplicación asociados a la clase:

```
# Se hace uso de los métodos de suma y multiplicación
print(" Suma de la matriz 1 y 2:")
print(operaciones.sumaMatrices(lista1, lista2))
```

```
print(" Multiplicación de la matriz 3 y 4:")
print(operaciones.MultiplicacionMatrices(lista3, lista4))
print(" Multiplicación de la matriz 1 y 4:")
print(operaciones.MultiplicacionMatrices(lista1, lista4))
```

La ejecución del programa anterior se muestra en la figura 10.10.

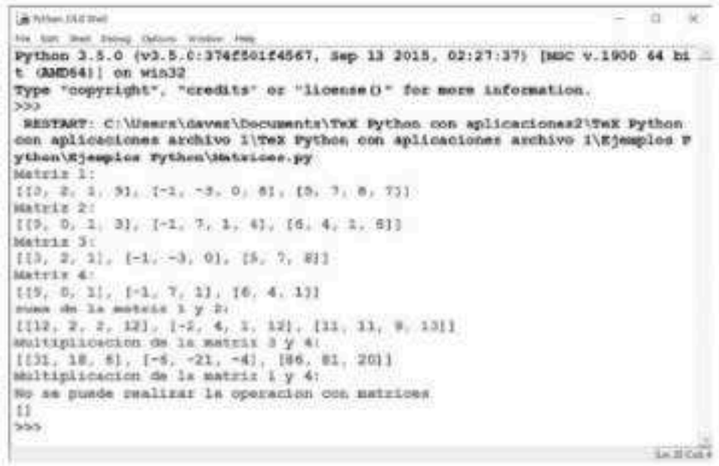


Figura 10.10 Ejecución de la clase OperacionMatrices.

10.9.4 Figuras geométricas

En este ejemplo se crea una clase padre llamada `Figura` de la que heredan dos clases hijo llamadas `Rectangulo` y `Triangulo`, las cuales utilizan las variables de la clase padre para calcular el área y perímetro. Es importante destacar que el comportamiento de los métodos de área y perímetro son redefinidos para las clases hijo. Para crear este programa lo primero que se debe hacer es crear la clase padre, la cual tendrá las variables y métodos que son accesibles por las clases hijo:

```
# Inicio de la clase padre
class Figura(object):

    valor1 = None
    valor2 = None
```

```
    def __init__(self, valor1, valor2):
        #Se inicializan las variables que son comunes a las clases hijo
        self.valor1 = valor1
        self.valor2 = valor2

    def area(self):
        # Se le asigna un comportamiento genérico a la clase padre
        print(" El área de la figura no está definida.")

    def perimetro(self):
        print(" El perímetro de la figura no está definida.")

# Fin de clase padre
```

De la clase anterior se puede ver que los métodos de área y perímetro tienen un comportamiento genérico, así como las variables asignadas a los atributos de la clase. La idea central subyace en que las clases hijo utilicen las variables que son comunes independientemente de si se trata la clase Rectángulo o Triangulo y redefinan los métodos área y perímetro de la clase padre, ya que el comportamiento es diferente para ambas figuras. Teniendo en cuenta lo anterior, ahora definimos la clase Rectángulo de la siguiente manera:

```
# Inicio de clase hijo
class Rectangulo(Figura):

    '''Se usa la definición del constructor de la clase
    padre para inicializar a la clase hijo'''
    def __init__(self, valor1, valor2):
        super(Rectangulo,self).__init__(valor1, valor2)

    # Se obtiene el área del rectángulo
    def area(self):
        areaRectangulo = self.valor1 * self.valor2
        print(" El área del rectángulo es: " + str(areaRectangulo) )
        return areaRectangulo
```

```
    # Se obtiene el perímetro del rectángulo
    def perimetro(self):
      perimetroRectangulo = 2 * self.valor1 + 2 * self.valor2
      print(" El perímetro del rectángulo es: " +
            str(perimetroRectangulo))
      return perimetroRectangulo

  #Fin de clase hijo
```

En el constructor de la clase `Rectangulo` se llama al constructor de la clase padre con el objetivo de que inicialice las variables `valor1` y `valor2`, las cuales son ocupadas en los métodos redefinidos `area` y `perimetro`. De manera análoga se implementa la clase `Triangulo`:

```
  # Inicio de clase hijo
  class Triangulo(Figura):

    base = None
    altura = None

    def __init__(self, valor1, valor2, base, altura):
      super(Triangulo,self).__init__(valor1, valor2)
      self.base = base
      self.altura = altura

    #Se obtiene el área del triángulo
    def area(self):
      areaTriangulo = (self.base * self.altura) / 2
            print(" El área del triángulo es: " + str(areaTriangulo))
      return areaTriangulo

    # Se obtiene el perímetro del triángulo
    def perimetro(self):
      perimetroTriangulo = (self.valor1 + self.valor2 + self.base)
      print(" El perímetro del triángulo es: " +
            str(perimetroTriangulo))
      return perimetroTriangulo
```

```
#Fin de clase hijo
```

A diferencia de la clase `Rectangulo`, la clase `Triangulo` tiene dos atributos propios de la clase (base y altura) que no comparte con la clase `Rectangulo`. Teniendo las clases construídas se crean objetos con los que se probarán los métodos `area` y `perimetro`:

```
# Creación de objetos
objeto1 = Figura(31, 12)
objeto2 = Rectangulo(9, 5)
objeto3 = Triangulo(10, 8, 5, 6)
# Manipulación de las funciones de los objetos
print(" Clase padre, objeto1: ")
objeto1.perimetro( )
objeto1.area( )
print(" Clase hijo, objeto2:")
objeto2.perimetro( )
objeto2.area( )
print(" Clase hijo, objeto3: ")
objeto3.perimetro( )
objeto3.area( )
```

La ejecución del programa anterior se muestra en la figura 10.11.

10.9.5 Lista enteros

En este ejemplo se crea una clase personalizada en una lista clásica de Python para que solamente acepte números enteros, así como se crean distintos métodos para recuperar y eliminar elementos de la lista. Para crear el programa lo primero que se debe de hacer es crear la clase llamada `ListaEnteros`, la cual contendrá un atributo, el cual es una lista común de Python:

```
#Inicio de clase
class ListaEnteros:
    Lista = None
```

El siguiente paso es la creación de un constructor en el que se limite el uso de una lista solamente con números enteros de la siguiente manera:

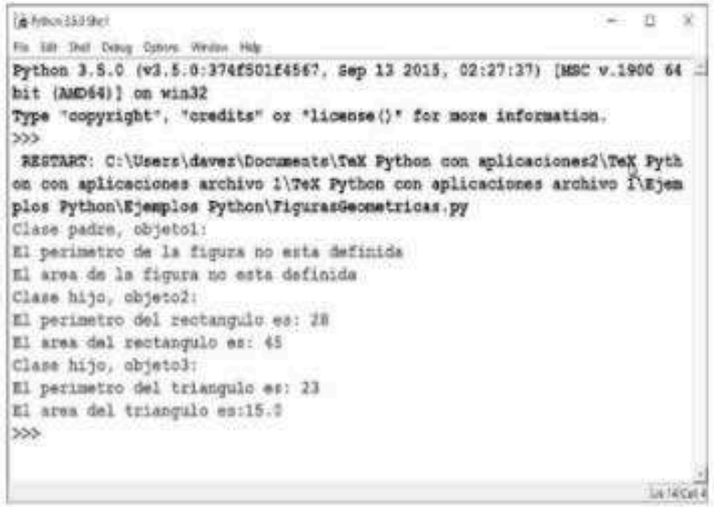


Figura 10.11 Ejecución de las del programa de figuras geométricas.

```
# Constructor de la clase
def __init__(self, listaNumeros):
    # Se verifica que la lista solo tenga valores enteros
    if type(listaNumeros) == list:
        if all(isinstance(x, int) for x in listaNumeros) == True:
            self.Lista = listaNumeros
        else:
            print ("La lista contiene elementos que no son enteros.")
            self.Lista = [ ]
```

Después se crean los métodos para obtener la lista a través de la clase y el método que sobreescribe el comportamiento de la impresión común para que imprima la lista de manera personalizada:

```
# Se sobrescribe la forma de imprimir la lista
def __repr__(self):
    return ("[" + ", ".join(str(x) for x in self.Lista) + "]")
```

```
# Se obtiene la lista
def obtenerLista(self):
    return self.Lista
```

Los siguientes métodos a implementar son la obtención del número de elementos y el de agregar números, tomando en cuenta que solamente se pueden agregar números enteros:

```
# Se obtiene el número de elementos en la lista
def numeroElementos(self):
   print (" Número de elementos en la lista: " +
        str(len(self.Lista)))

# Se agrega un número a la lista
def agregarNumero(self,numero):
   if type(numero) == int:
      self.Lista.append(numero)
   else:
      print(" Solamente se pueden agregar enteros.")
```

El siguiente paso consiste en la creación de métodos para la eliminación de elementos de la lista:

```
# Se elimina la primera ocurrencia de un número
def eliminarPrimeraOcurrencia(self, numero):
   if type(numero) == int:
      if numero in self.Lista:
         self.Lista.remove(numero)
      else:
         print(" El número no está en la lista.")
   else:
      print(" Solamente se pueden eliminar enteros.")

# Se eliminan todas las ocurrencias de un número
def eliminarOcurrencias(self, numero):
   if type(numero) == int:
```

```
        if numero in self.Lista:
          self.Lista = [x for x in self.Lista if x != numero]
        else:
          print(" El número no está en la lista.")
      else:
        print(" Solamente se pueden eliminar enteros.")
```

Finalmente se crean los métodos para buscar las ocurrencias de un número en la lista personalizada:

```
# Se busca la primera ocurrencia de un número
def buscarNumero(self, numero):
  if type(numero) == int:
    if numero in self.Lista:
      print(" Existe al menos una ocurrencia del número.")
    else:
      print(" No existen ocurrencias de ese número en la lista.")
  else:
      print(" Solamente se pueden buscar enteros.")


# Se buscan todas las ocurrencias de un número
def obtenerPosiciones(self, numero):
  if type(numero) == int:
    if numero in self.Lista:
      return [x + 1 for x in range(len(self.Lista))
        if self.Lista[x] == numero]
    else:
      return [ ]
  else:
      print("Sólo se puede obtener la posición de números enteros.")
      return [ ]
# Fin de clase
```

Teniendo la clase construida se crea un objeto de la clase y se utilizan los distintos métodos implementados:

```
# Creación del objeto asociado a la clase

lista = ListaEnteros([1, 2, 5, 3, 5, 5])

# Manipulación de las funciones del objeto
print(" Se imprime la lista en pantalla.")
print (lista)
print("\n")

print(" Contamos el número de elementos en la lista.")
lista.numeroElementos( )
print("\n")

print(" Agregamos el número 15 a la lista.")
lista.agregarNumero(15)
print (lista)
print("\n")

print(" Buscamos si existe en la lista el número 5.")
lista.buscarNumero(5)
print("\n")

print(" Buscamos todas las posiciones del número 5.")
print(lista.obtenerPosiciones(5))
print("\n")

print(" Eliminamos la primera ocurrencia del número 5.")
lista.eliminarPrimeraOcurrencia(5)
print(lista)
print("\n")

rint(" Eliminamos todas las ocurrencias del número 5.")
lista.eliminarOcurrencias(5)
```

```
print(lista)
```

La ejecución del programa anterior se muestra en la figura 10.12.

Figura 10.12 Ejecución del programa lista de enteros.

10.10 Instrucciones de Python del Capítulo 10

Instrucción	Descripción
`add`	Sobrescribir el operador + .
`class`	Palabra clave para definir una clase.
`getter`	Se usa para obtener los atributos de una clase.
`init`	Inicializar los atributos de una clase.
`operador punto .`	Referencia a un atributo o método de una clase.
`self`	Referencia al mismo objeto.
`setter`	Se usa para establecer los atributos de una clase.
`sub`	Sobrescribir el operador - .
`super`	Inicializar los atributos de la clase padre (herencia).
`zip`	Se usa para reorganizar listas.

10.11 Conclusiones

En este capítulo hemos dado una introducción al uso del paradigma de programación con objetos en Python. Solamente se han dado las definiciones más elementales. Sin embargo, con este capítulo ya se pueden crear clases de la POO en Python. Los ejercicios al final del capítulo ayudarán a profundizar en el tema.

10.12 Ejercicios

1. Sabiendo que la gravedad de la luna es un sexto de la gravedad de la tierra, escriba una clase que realice la conversión del peso en la tierra al peso en la luna.
2. Escriba una clase que calcule la conversión de temperaturas entre grados Fahrenheit, Centígrados y Kelvin.
3. Escriba una clase que implemente las operaciones básicas entre fracciones (suma, resta y multiplicación).
4. Escriba una clase con el nombre Baraja. Debe usar un mazo de 52 cartas y dar manos de 5 cartas a tres jugadores. Decidir cuál es el ganador de la partida.

Capítulo 11
Graficación en Python

11.1 Introducción
11.2 Visualización de datos
11.3 Gráficas en 2 dimensiones
11.4 Figuras múltiples
11.5 Subgráficas
11.6 Otros tipos de gráficas bidimensionales
11.7 Opciones de gráficas
11.8 Gráficas tridimensionales
11.9 Instrucciones de Python del Capítulo 11
11.10 Conclusiones
11.11 Ejercicios

Una imagen vale más que mil palabras.

Proverbio chino

Objetivos

Una de las mejores maneras de presentar datos es por medio de una gráfica o una imagen. Python permite visualizar información usando un paquete adicional. En este capítulo presentamos una introducción a la visualización de datos con Python.

11.1 Introducción

Una de las características más importantes de Python es la visualización. En este capítulo se muestra cómo realizar gráficas simples de funciones de una o dos variables, lo que produce gráficas en dos y tres dimensiones. La manera de realizar gráficas muy complicadas se vuelve simple cuando se usa `matplotlib` ya que se cuenta con una gran cantidad de instrucciones para graficar. En este capítulo se examina el potencial de `matplotlib` para graficar datos y funciones.

11.2 Visualización de datos

Visualización de datos es una característica importante de un programa, ya que permite tener un panorama de como se comportan los datos. Por ejemplo, una gráfica de las cotizaciones de las acciones de una compañía permite rapidamente saber la tendencia de la compañía. Graficar una función trigonométrica nos permite apreciar que estamos trabajando con una función periódica. Los programas diseñados en Python también pueden aprovechar esta ventaja de graficación de datos. Python puede emplear bibliotecas de graficación como las que tiene `matplotlib`, el cual fue diseñado por John D. Hunter (1968-2012) y que está disponible sin costo para poder realizar gráficas de muy alta calidad a través de Python. Las características y opciones de graficado son muy variadas con `matplotlib` y van desde generar gráficas en dos y tres dimensiones hasta la posibilidad de cambiar las propiedades de ellas. También se

verá cómo identifica `matplotlib` las gráficas y cómo se pueden realizar cambios en ellas.

Dentro del paquete `matplotlib` existe una biblioteca con instrucciones para graficar. Esta biblioteca es `pyplot`. De esta manera, siempre que vayamos a graficar tenemos que importar esta biblioteca. En el transcurso del capítulo iremos describiendo las instrucciones que se deben importar de esta biblioteca. Las instrucciones para descargar e instalar `matplotlib` se encuentran en el Apéndice A y y para estudiar y usar este capítulo, ya debe estar instalado.

11.3 Gráficas en 2 dimensiones

La instrucción básica para graficar es `plot(y)` donde `y` es una lista de datos. Con el uso de `plot` se abre una nueva ventana con la gráfica generada. La instrucción `plot` se encuentra en la biblioteca `pyplot` del paquete `matplotlib`. Se carga con:

```
import matplotlib.pyplot
```

Para desplegar la gráfica debemos incluir la instrucción `show( )`.

De manera alterna, podemos importar solamente las instrucciones `plot` y `show` con:

```
from matplotlib.pyplot import plot, show
```

Ejemplo 11.1

Gráfica de una lista

Se desea graficar la lista de puntos `y = [ 1, 2, 5, 3 ]`. Para realizar la gráfica de esta lista hacemos lo siguiente:

- Importamos las instrucciones `plot` y `show` de la biblioteca `matplotlib.numpy` con

  ```
  from matplotlib.pyplot import plot, show
  ```

- Realizamos la gráfica con:

  ```
  plot( [ 1, 2, 5, 3 ] )
  ```

- Finalmente, desplegamos la gráfica con:

```
show( )
```

El archivo completo es:

```
from matplotlib.pyplot import plot, show
plot( [ 1, 2, 5, 3 ] )
show( )
```

El resultado se muestra en la figura 11.1. Vemos que la gráfica ha unido los puntos de la lista con segmentos de rectas. Si solamente deseamos que se grafiquen los puntos con `x`, a la instrucción `plot` añadimos 'x' como en:

```
plot( [ 1, 2, 5, 3 ] , 'x')
```

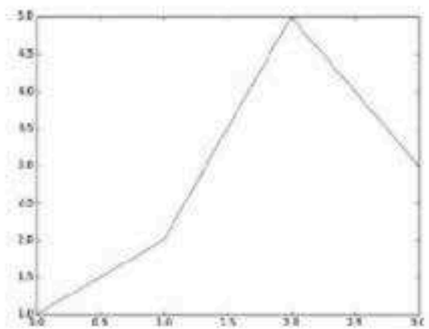


Figura 11.1 Gráfica de la lista `[ 1, 2, 5, 3 ]`.

Notamos que la escala del eje `x` se ajusta automáticamente a valores que se asignan por `matplotlib`.

En el ejemplo anterior dimos solamente los valores de `y`. Si deseamos dar los valores de `x` también en una lista los damos en una lista que antecede a la lista de valores de `y`, como se muestra en el siguiente ejemplo:

Ejemplo 11.2

Gráfica de una lista (x, y)

Se desea graficar la lista de puntos `y = [1, 2, 5, 3]` contra `x = [5, 6, 7, 8]`. Para realizar la gráfica el archivo es el siguiente:

```
from matplotlib.pyplot import plot, show
plot( [ 5, 6, 7, 8 ], [ 1, 2, 5, 3 ] )
show( )
```

El resultado se muestra en la figura 11.2. Nótese el cambio de valores del eje **x** que ahora va de 5 a 8 como se especifica en los valores de **x**.

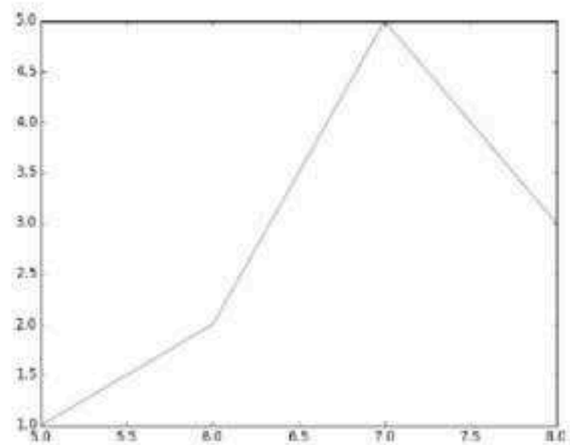


Figura 11.2 Gráfica de la lista `y = [ 1, 2, 5, 3 ]` vs. `x = [ 5, 6, 7, 8 ]`.

Para cambiar los límites de los ejes usamos la instrucción `xlim` y `ylim` de la siguiente manera:

```
xlim(límite inferior de x, límite superior de x)
ylim(límite inferior de y, límite superior de y)
```

Ejemplo 11.3

Gráfica con límites

Se desea realizar la gráfica del ejemplo 11.2 con límites de x de -1 a 12 y límites de y de 0 a 8. Las instrucciones `xlim` `ylim` se deben importar. El archivo es:

```
from matplotlib.pyplot import plot, show, xlim, ylim
plot( [ 5, 6, 7, 8 ], [ 1, 2, 5, 3 ] )
xlim( -1, 12)
ylim( 0, 8)
show( )
```

El resultado se muestra en la figura 11.3. Nótense los límites de los ejes x, y.

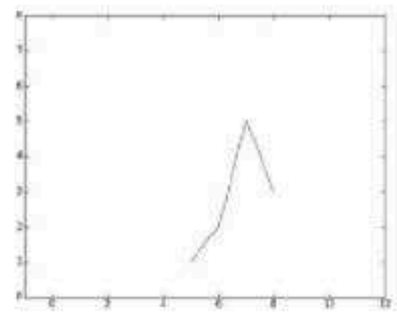


Figura 11.3 Gráfica de la lista `y = [ 1, 2, 5, 3 ]` vs. `x = [ 5, 6, 7, 8 ]` con límites de los ejes.

Podemos generar la lista de valores para los ejes x, y usando la instrucción `linspace` de la biblioteca `numpy`. El formato de `linspace` es:

```
linspace(límite inferior, límite superior, número de puntos)
```

El siguiente ejemplo muestra su uso.

Ejemplo 11.4

Gráfica del `sen(x)` usando `linspace`

Se desea realizar la gráfica del sen(x) con límites de **x** de $-\pi$ a $+\pi$ y limites de **y** de -2 a +2. Para poder generar los valores de **x** usamos:

```
x = linspace( -pi, pi, 256)
```

Para generar los valores de y = sen(x) usamos:

```
y = sin(x)
```

Finalmente graficamos con:

```
plot(x, y)
show( )
```

La instruccion `linspace`, el valor de `pi`, y la función `sin` las importamos de `numpy` con:

```
from numpy import linspace, pi, sin
```

El archivo completo es:

```
from numpy import linspace, pi, sin
from matplotlib.pyplot import plot, show, xlim, ylim
x = linspace( -pi, pi, 256)
y = sin(x)
plot(x, y)
ylim(-2, 2)
show( )
```

La gráfica resultante se muestra en la figura 11.4. Los valores del eje **x** no están dados en términos de π. Para cambiar los ejes usamos "ticks". Para el eje **x** usamos los `xticks`. Para nuestro ejemplo queremos que aparezca en los valores $-\pi$, $-\pi/2$, 0, $+\pi/2$, $+\pi$. Esto lo hacemos con:

```
from matplotlib.pyplot import xticks
xticks( [ -pi, -pi/2, 0, pi/2, pi ],
            [ r'$-\pi$', r'$-\pi/2$', r'$0$', r'$\pi/2$', r'$\pi$' ] )
```

La primera lista indica el valor donde se coloca el tick y la segunda lista es el tick. De este modo el tick π se coloca en el valor de `x = pi`. Esto lo agregamos al archivo anterior y la gráfica resultante aparece en la figura 11.5.

Nótese que tenemos que importar de la biblioteca `matplotlib.pyplot`la instrucción `xticks`.

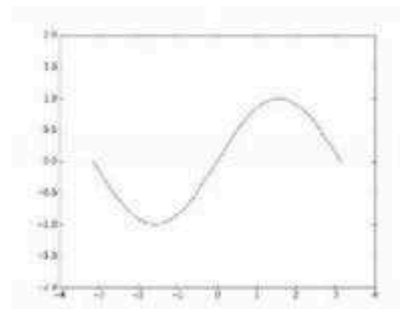


Figura 11.4 Gr àfica de sin(x).

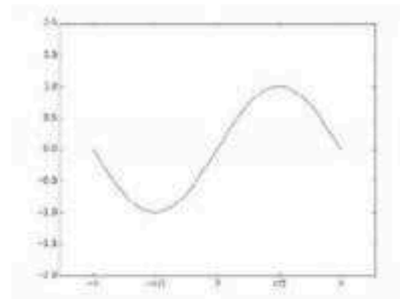


Figura 11.5 Gráfica de `sin(x)` con ticks en el eje x.

También podemos añadir leyendas a una gráfica usando las instrucciones `label` y `legend`. La instrucción `label` se añade dentro de la instrucción `plot`.

Ejemplo 11.5

Gráfica de `sen(x)` y `cos(x)` con `label` y `legend`

Ahora se desea realizar las gráficas de `sen(x)` y `cos(x)` añadiendo la leyenda sen(x) y cos(x). El archivo es similar al del ejemplo anterior. La diferencia está en que se debe calcular el coseno con:

```
z = cos(x)
```

Y la instrucción `plot` debe modificarse con:

```
plot(x, y, label = 'seno')
```

Una instrucción similar se escribe para el coseno:

```
plot(x, z, label = 'coseno')
```

Adicionalmente se debe colocar la leyenda en algún lugar de la gráfica. En nuestro caso seleccionamos que lo coloque en la parte superior izquierda con:

```
legend( loc = 'upper left' )
```

Finalmente, debemos importar en `pyplot` la instrucción `legend`. El archivo final es:

```
from numpy import linspace, pi, sin
from matplotlib.pyplot import plot, show, xlim, ylim
from matplotlib.pyplot import legend, xticks
x = linspace( -pi, pi, 256)
y = sin(x)
z = cos(x)
plot(x, y, label = 'seno')
plot(x, z, label = 'coseno')
legend( loc = 'upper left' )
xticks( [ -pi, -pi/2, 0, pi/2, pi ],
          [ r'$-\pi$', r'$-\pi/2$', r'$0$', r'$\pi/2$', r'$\pi$'] )
```

```
ylim(-2, 2)
show( )
```

La gráfica resultante se muestra en la figura 11.6.

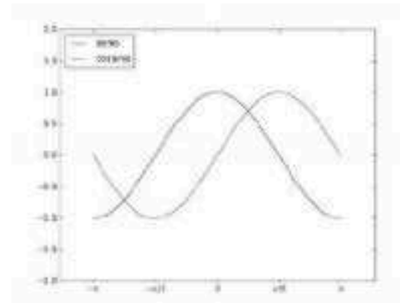


Figura 11.6 Gráfica de seno y coseno con `legend`.

11.4 Figuras múltiples

Para poder realizar varias figuras es necesario numerarlas. Esto lo hacemos con la instrucción figure. Es necesario asociarle un número de figura de la siguiente manera:

```
a = figure(2)
```

lo que hace que la figura activa sea la figura 2. Se debe importar la instrucción `figure` de `matplotlib`.

Ejemplo 11.6

Ejemplo con dos figuras

El ejemplo 11.5 lo realizamos ahora con cada gráfica en una ventana diferente. Se debe importar la instrucción `figure` de `matplotlib`. El archivo es:

```
from numpy import linspace, pi, sin, cos
from matplotlib.pyplot import plot, show, xlim, ylim
from matplotlib.pyplot import legend, xticks, figure
x = linspace( -pi, pi, 256)
y = sin(x)
z = cos(x)
a = figure(2)
plot(x, y, label = 'seno')
legend( loc = 'upper left' )
b = figure(20)
plot(x, z, label = 'coseno')
legend( loc = 'upper left' )
xticks( [ -pi, -pi/2, 0, pi/2, pi ],
          [ r'$-\pi$', r'$-\pi/2$', r'$0$', r'$\pi/2$', r'$\pi$'])
ylim(-2, 2)
show( )
```

Las gráficas resultantes se muestran en la figura 11.7.

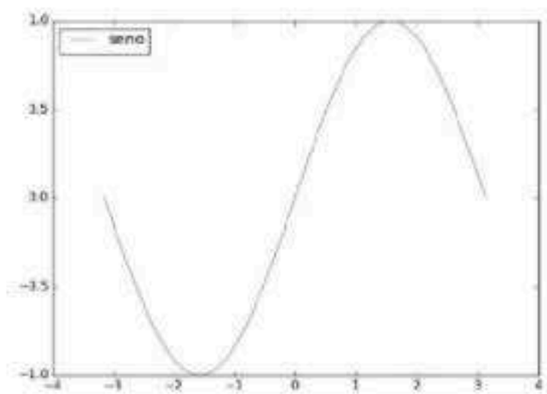


Figura 11.7a Ventana para la figura 2.

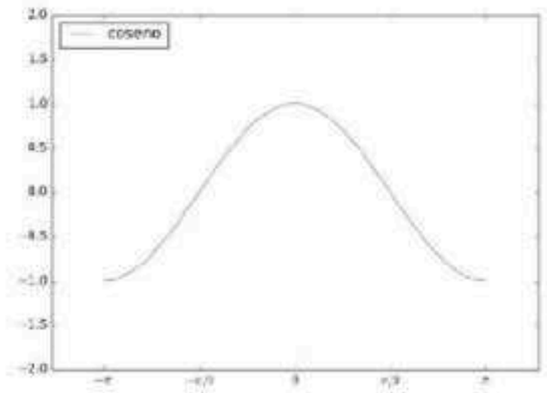


Figura 11.7b Ventana para la figura 20.

11.5 Subgráficas

Para realizar varias gráficas en la misma figura usamos subgráficas que se generan con la instrucción `subplot`, la cual se debe importar de `matplotlib`. El formato es:

```
subplot (m, n, k)
```

Esta instrucción divide la figura en `m` × `n` subgráficas arregladas en forma de matriz de m renglones y `n` columnas. La variable `k` numera las subgráficas de izquierda a derecha y de arriba hacia abajo en numeración consecutiva y hace que se active la subgráfica correspondiente.

- `m` es el número de renglones.
- `n` es el número de columnas.
- `k` es la k-ésima subgráfica en numeración consecutiva o posición en la gráfica.

Mostramos con un ejemplo la colocación de las subgráficas usando el ejemplo 11.6.

Ejemplo 11.7

Ejemplo con dos figuras

En este caso usamos dos renglones de gráficas. Las correspondientes instrucciones son:

```
subplot(2, 1, 1)
```

y

```
subplot(2, 1, 2)
```

El archivo completo es:

```
from numpy import linspace, pi, sin, cos
from matplotlib.pyplot import plot, show, xlim, ylim
from matplotlib.pyplot import legend, xticks, subplot
x = linspace( -pi, pi, 256)
y = sin(x)
z = cos(x)

subplot(2, 1, 1) # n = 2 renglones
# m = k = 1 , una gráfica en el primer renglón.
plot(x, y, label = 'seno')
legend( loc = 'upper right' )

subplot(2, 1, 2) # m = 1, k = 2 una gráfica en el segundo renglón.
plot(x, z, label = 'coseno')
legend( loc = 'upper left' )
xticks( [ -pi, -pi/2, 0, pi/2, pi ],
          [ r'$-\pi$', r'$-\pi/2$', r'$0$', r'$\pi/2$', r'$\pi$'] )

ylim(-2, 2)
show( )
```

El resultado se muestra en la figura 11.8. Ahí vemos dos renglones de gráficas cada renglón con una gráfica. Ahora sustituimos:

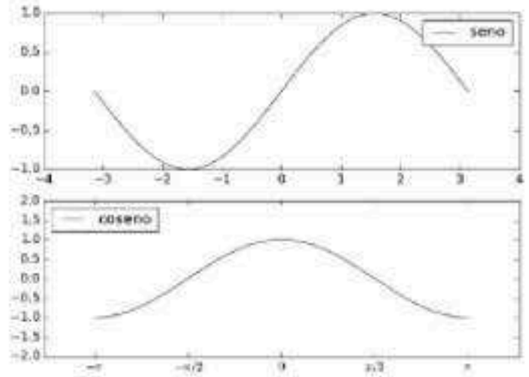


Figura 11.8 Subgráficas en dos renglones.

```
subplot(2, 1, 1)        por        subplot(1, 2, 1)
```

y

```
subplot(2, 1, 2)        por        subplot(1, 2, 2)
```

con lo que obtenemos la figura 11.9.

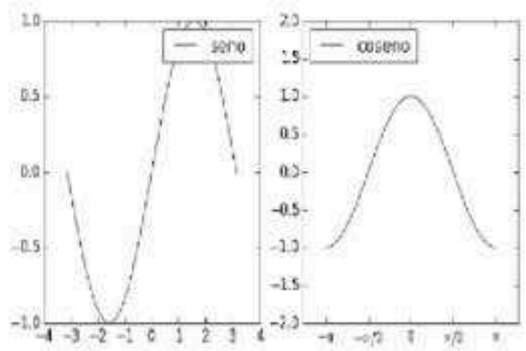


Figura 11.9 Subgráficas en el mismo renglón.

Si ahora deseamos 6 gráficas en tres renglones tendremos que usar:

`subplot(3, 2, 5)` para la quinta subgráfica o primera del tercer renglón.
`subplot(3, 2, 4)` para la cuarta subgráfica o segunda del segundo renglón.

11.6 Otros tipos de gráficas bidimensionales

En esta sección describimos otros tipos de gráficas en dos dimensiones que son muy usadas.

11.6.1 Gráfica polar

La gráfica polar se obtiene usando la instrucción `plot` pero usando las componentes `r` (radio vector) y θ (ángulo o argumento). Cualquiera de los dos `r` y θ o los dos pueden variar en un rango determinado. Por ejemplo, en la ecuación

$$r = \theta$$

podemos variar uno de los dos parámetros o los dos.

Ejemplo 11.8

Gráfica polar

La gráfica polar de

$$r = 2\pi\theta$$

se puede graficar con el siguiente archivo:

```
from numpy import pi, arange, linspace
from matplotlib.pyplot import subplot, plot,show, grid, title
teta = arange(0, 3, 0.01) # Se generan los valores de θ.
r = 2*pi * teta  # Se calcula r para cada valor de θ.
subplot(111, polar = True) # Se crea la subgráfica polar
plot(r, teta, color = 'r', linewidth = 3) # Se crea la gráfica.
```

```
grid(True) # Se añade la retícula.
title(" Gráfica polar de r = 2$\pi\\theta$ ")
show( )
```

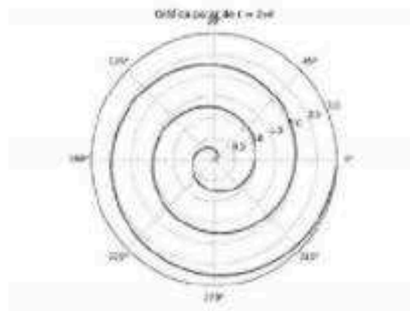


Figura 11.10 Gráfica polar.

11.6.2 Gráfica de pie

Ejemplo 11.9

Gráfica de pie

La gráfica de pie se usa cuando queremos representar en porcentaje los datos. La gráfica de pie se forma con la instrucción `pie`. Por ejemplo para graficar con una gráfica de pie los datos [15, 30, 40, 5, 10] usamos el siguiente archivo:

```
from matplotlib.pyplot import figure,subplot, title, pie, show
# make a square figure
figure(1, figsize = (6, 6)) # Hace figura cuadrada.
fracs = [15, 30, 40, 5, 10]
pie(fracs, autopct = '%2.1i%%') # Escribir los porcentajes en la gráfica.
title('Gráfica de pie')
show( )
```

El resultado se muestra en la figura 11.11.

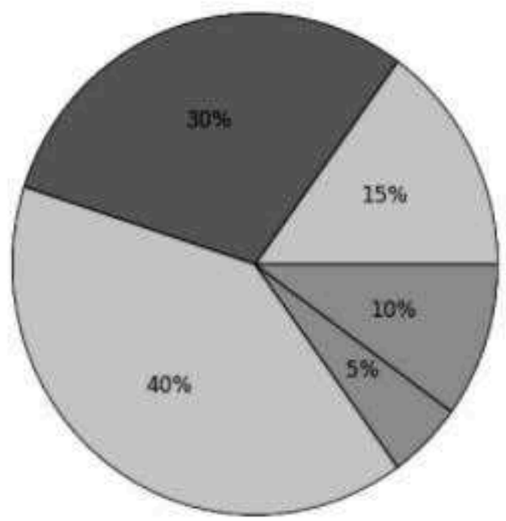


Figura 11.11 Gráfica de pie.

11.6.3 Gráfica de histograma

La gráfica de histograma se usa para representar distribuciones de datos. Es una gráfica de barras.

Ejemplo 11.10

Gráfica de histograma

La gráfica de histograma se forma con la instrucción `hist`. Por ejemplo, para graficar con una gráfica de histograma los números aleatorios generados con una distribución normal gaussiana usamos el siguiente archivo:

```
from matplotlib.pyplot import title, hist, show
from numpy.random import normal
numeros_gaussianos = normal(size = 1000)
```

```
hist(numeros_gaussianos)
title('Gráfica de histograma')
show( )
```

La gráfica del histograma que se obtiene se despliega en la figura 11.12.

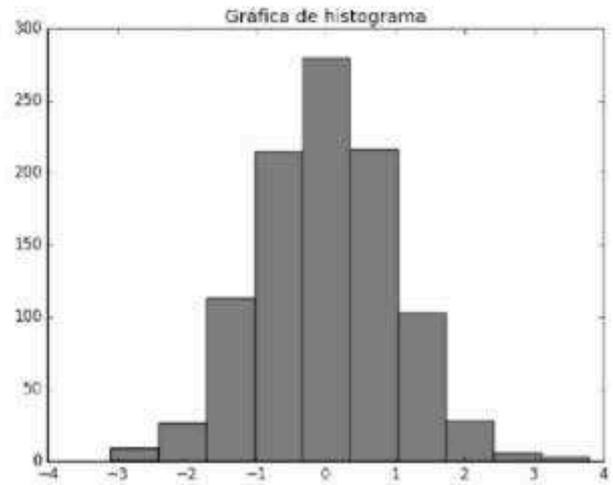


Figura 11.12 Gráfica de histograma.

11.6.4 Gráfica de `stem` o de puntos

La gráfica de stem o de puntos se usa para graficar señales en forma de puntos. Se usa en procesado de señales digitales o de tiempo discreto.

Ejemplo 11.11

Gráfica de stem

La gráfica de `stem` es una gráfica de puntos unidos al eje horizontal por medio de una línea recta. Se forma con la instrucción `stem`. Por ejemplo para graficar con una gráfica de `stem` una lista de números usamos el siguiente archivo:

```
from matplotlib.pyplot import stem, show
from numpy import pi, arange
```

```
stem( arange(-pi, pi) )
show( )
```

La gráfica de `stem` se muestra en la figura que se obtiene se despliega en la figura 11.13. En este caso los puntos se unen al eje horizontal con una línea recta. Si se quiere usar una línea punteada le añadimos a la instrucción `stem` la opción '-.'. El archivo queda como:

```
from matplotlib.pyplot import stem, show
from numpy import pi, arange
stem( arange(-pi, pi), '-.' )
show( )
```

La gráfica que se obtiene se muestra en la figura 11.14.

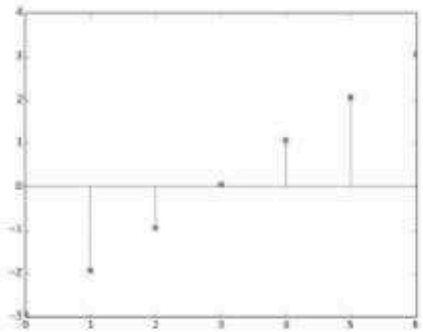


Figura 11.13 Gráfica de stem.

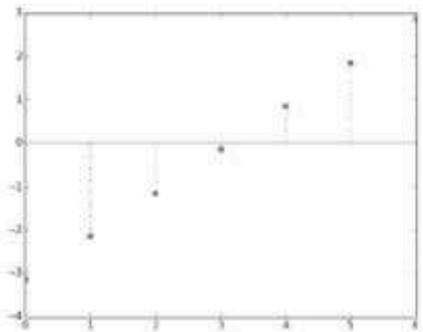


Figura 11.14 Gráfica de stem con cambio de escala.

11.7 Opciones de gráficas

Las gráficas las podemos enriquecer con color, cambio del grueso de la traza o ticks, entre otras cosas. Las tablas 11.1 y 11.2 nos muestran diferentes opciones que tenemos para el color que le podemos añadir a una gráfica. La tabla 11.1 nos presenta los símbolos o ticks que le podemos añadir a una curva para mejorar su presentación o para destacar alguna característica de la curva graficada. La tabla 11.2 nos muestra los colores que puede tener una curva.

Tabla 11.1 Marcadores de curvas.

Símbolo	Descripción
-	línea sólida
--	línea discontinua
-.	línea punto y guión
:	línea de puntos
.	puntos
o	círculos
^	triángulos hacia arriba
v	triángulos hacia abajo
<	triángulos hacia la izquierda
>	triángulos hacia la derecha
s	cuadrado
+	signo más
x	cruz
D	diamante
d	diamante delgado
h	hexágono
p	pentágono

Tabla 11.2 Colores de curvas

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

11.8 Gráficas tridimensionales

Las gráficas tridimensionales requieren tomar valores en dos variables que generalmente son `x`, `y` para dar valores a un eje `z`.

Ejemplo 11.12

Gráfica de una curva paramétrica

La gráfica de una curva paramétrica requiere que podamos escribir las ecuaciones en términos de un parámetro. La gráfica paramétrica se forma con la instrucción `plot` pero previamente tenemos que indicar que será una gráfica tridimensional con la instrución `Axes3D` que se importa de la biblioteca `mpl_toolkits.mplot3d`. Por ejemplo, para graficar las ecuaciones:

$$z = r$$
$$x = r * sin(\theta)$$
$$y = r * cos(\theta)$$

donde el valor de r toma ciertos valores dados por una instrucción `linspace`. La gráfica paramétrica se obtiene con el siguiente archivo:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.pyplot import title, figure, show, plot, legend
from numpy import linspace, sin, cos, pi
a = figure()
a.gca(projection = '3d')
teta = linspace(-4 * pi, 4 * pi, 100)
r = linspace(0, pi, 100)
z = r
x = r * sin(teta)
y = r * cos(teta)
plot( x, y, z, label = ' Curva Paramétrica ' )
legend( )
show( )
```

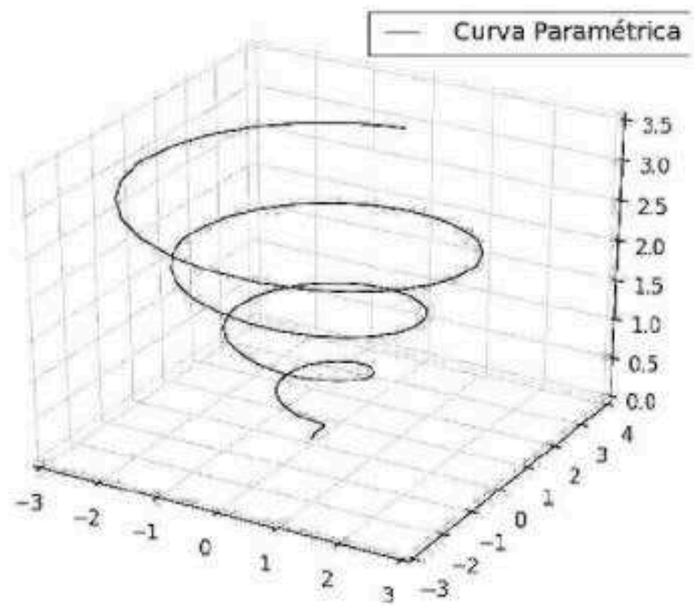


Figura 11.15 Gráfica paramétrica.

Ejemplo 11.13

Gráfica de una superficie

La gráfica de una curva de superficie requiere que calculemos una malla con `meshgrid` donde debemos especificar el rango, por ejemplo con

```
X = arange(-5, 5, 0.25)
Y = arange(-5, 5, 0.25)
X, Y = meshgrid(X, Y)
```

Las instrucciones `arange` y `meshgrid` deben importarse de `numpy`. La instrucción `meshgrid` crea los valores para los puntos `X`, `Y` donde vamos a evaluar la función deseada. También debemos indicar que se trata de una gráfica tridimensional con la instrución `Axes3D` que se importa de la biblioteca `mpl_toolkits.mplot3d`. Por

ejemplo, para graficar la función

$$f(\mathrm{X}, \mathrm{Y}) = \mathrm{sen}(\mathrm{X}^2 + \mathrm{Y}^2)$$

debemos evaluar esta función en los puntos `X, Y`. El siguiente archivo sirve para obtener la gráfica de superficie de la función *f(X,* `Y`):

```
from matplotlib import cm
from matplotlib.pyplot import plot, figure, show, title
from numpy import arange, sqrt, sin, meshgrid

fig = figure( )
ax = fig.gca(projection = '3d')

X = arange(-5, 5, 0.25)
Y = arange(-5, 5, 0.25)
X, Y = meshgrid(X, Y)
Z = sin( sqrt(X**2 + Y**2) )
A = ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1,
                    cmap = cm.coolwarm)
title( 'Gráfica de superficie ')
show( )
```

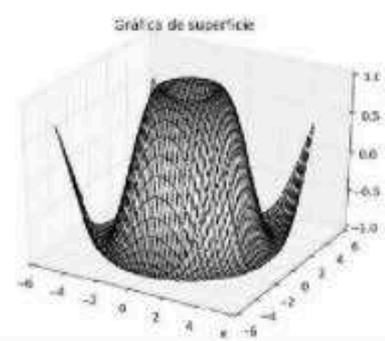


Figura 11.16 Gráfica de superficie.

Si ahora añadimos los parámetros

```
rstride = 2, cstride = 2, cmap = cm.coolwarm, linewidth = 0
```

para que quede como:

```
A = ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1,
                    cmap = cm.coolwarm, linewidth = 0)
```

dentro de la instrucción `plot_surface` obtenemos una figura con las líneas muy delgadas como se muestra en la figura 11.17.

Finalmente si añadimos la instrucción:

```
fig.colorbar(A, shrink = 0.5, aspect = 5)
```

Además, si cambiamos los pasos `rstride` y `cstride` a 2 obtenemos la gráfica de la figura 11.18. Ahí vemos que la superficie es más tosca y aparece la barra de colores lateral.

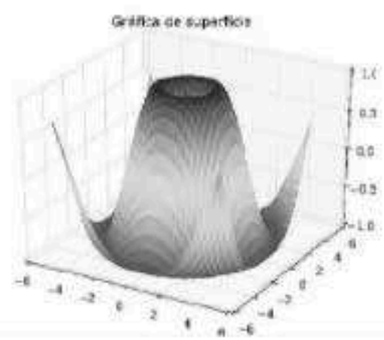


Figura 11.17 Gráfica de superficie con malla delgada.

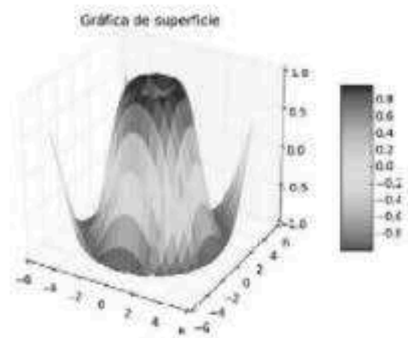


Figura 11.18 Gráfica de superficie con barra de colores.

La superficie de una figura la podemos observar desde distintos puntos de vista con sólo colocar el apuntador sobre la gráfica, presionar el botón izquierdo y arrastrar el apuntador. Esto nos permite observar la gráfica desde donde querramos verla.

Ejemplo 11.14

Gráfica de superficie de alambre (wireframe)

La gráfica de una superficie de alambre (wireframe) requiere los mismos pasos del ejemplo anterior pero cambiando la palabra `surface` por `wireframe`. Para apreciar mejor el resultado cambiamos en las instrucciones `arange` el paso de 0.25 a 0.5. El archivo completo es

```
from matplotlib import cm
from matplotlib.pyplot import plot, figure, show, title
from numpy import arange, sqrt, sin, meshgrid

fig = figure( )
ax = fig.gca(projection = '3d')
```

```
X = arange(-5, 5, 0.50)
Y = arange(-5, 5, 0.50)
X, Y = meshgrid(X, Y)
Z = sin( sqrt(X**2 + Y**2) )
A = ax.plot_wireframe(X, Y, Z)
title( ' Gráfica de superficie de alambre')
show( )
```

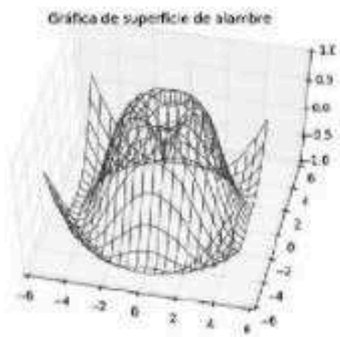


Figura 11.19 Gr àfica de superficie de alambre.

Ejemplo 11.15

Gráfica del campo magnético de un alambre usando `quiver`

El campo magnético se puede graficar usando las ecuaciones de las componentes del vector de campo magnético normalizado:

$$B_x = \frac{-y}{x^2 + y^2}$$
$$B_y = \frac{x}{x^2 + y^2}$$

Con el archivo siguiente evaluamos estas ecuaciones y el campo queda como se muestra en la figura 11.20. Se usan las instrucciones `quiver` y `quiverkey`:

```
from pylab import *
xmax = 2.0
xmin = -xmax
NX = 5
ymax = 2.0
ymin = -ymax
NY = 5


# Componentes vectoriales y puntos
x = linspace(xmin, xmax, NX)
y = linspace(ymin, ymax, NY)
X, Y = meshgrid(x, y)
S2 = X**2 + Y**2 # Este es el radio al cuadrado


# Campo magnético. El valor de 0.001 es para evitar división entre cero.
Bx = -Y/(S2 + 0.001)
By = +X/(S2 + 0.001)


figure( )
QP = quiver(X, Y, Bx, By)
quiverkey(QP, 0.85, 0.85, 1.0, 'Campo magnético')


# Límites de los ejes
dx = (xmax - xmin)/(NX - 1.)
dy=(ymax - ymin)/(NY - 1.)
axis( [xmin - dx, xmax + dx, ymin - dy, ymax + dy] )
title('Campo magnético de un alambre con corriente.')
xlabel('x (cm)')
ylabel('y (cm)')
show( )
```

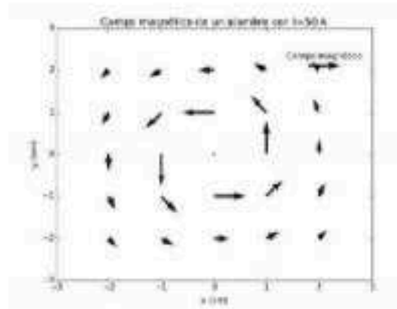


Figura 11.20 Gr àfica de `quiver` para mostrar el campo magrético normalizado alrededorde un alambre con corriente *I*.

Ejemplo 11.16

Gráfica de superficie con proyección sobre el plano x, y

En ocasiones deseamos realizar una proyección sobre el plano x, y de una superficie tridimensional. Esto lo hacemos graficando la superficie y el contorno lo que se puede hacer con `plot_suface` y `contourf`. El siguiente archivo realiza esto y produce la gráfica de la figura 11.21.

```
from pylab import arange, sin, sqrt, meshgrid, figure, cm
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.pyplot import show

fig = figure()
ax = Axes3D(fig)
X = arange( -4, 4, 0.25)
Y = arange( -4, 4, 0.25)
X, Y = meshgrid(X, Y)

R = sqrt(abs(X**3) + Y**2)
Z = sin(R)

ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1, cmap = cm.hot)
ax.contourf(X, Y, Z, zdir = 'z', offset = -2, cmap = cm.hot)
ax.set_zlim(-2, 2)
show()
```

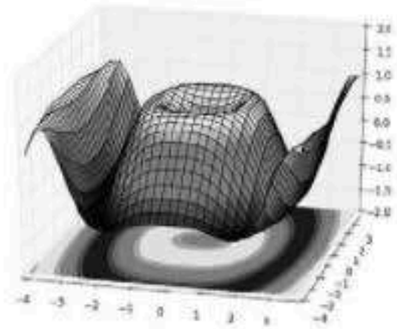


Figura 11.21 Gráfica de superficie con proyección sobre el plano x, y.

11.9 Instrucciones de Python del Capítulo 11

Instrucción	Descripción
`arange`	Crea una lista según los parámetros.
`aspect`	Razón de aspecto de la gráfica.
`autopct`	Espaciamiento de valores en una gráfica polar.
`Axes3D`	Indica que se grafica en 3D.
`cmap`	Indica que se usa mapa de colores.
`color`	Color de la curva de la gráfica.
`colorbar`	Se añade barra de colores.
`contour`	Realiza gráfica de contorno.
`coolwarm`	Color de la gráfica.
`cstride`	Parámetro de la gráfica de 3D.
`figsize`	Tamaño de la figura.
`figure`	Crea una nueva ventana para figura.
`grid`	Genera una rejilla para la figura.
`hist`	Grafica el histograma.
`label`	Etiqueta de la gráfica.
`legend`	Letrero dentro de la figura.
`linewidth`	Cambia el ancho de la línea.
`linspace`	Genera valores de la variable independiente.

Continúa

Instrucción	Descripción
`loc`	Posición de `legend` en la figura.
`matplotlib`	Biblioteca para graficar.
`meshgrid`	Rejilla para graficar en tres dimensiones.
`mpl_toolkits`	Biblioteca para graficar en 3D.
`normal`	Distribución estadística normal.
`numpy`	Biblioteca numérica.
`pie`	Realiza la gráfica de pie.
`plot`	Instrucción básica de graficación.
`plot_surface`	Grafica superficies.
`pyplot`	Biblioteca de graficación.
`polar`	Indica que la gráfica es polar.
`projection`	Obtiene la proyección de una gráfica 3D.
`quiver`	Realiza la gráfica de quiver.
`rstride`	Parámetro de la gráfica de 3D.
`show`	Despliega la gráfica realizada.
`size`	Tamaño de la gráfica.
`stem`	Realiza gráfica de puntos.
`subplot`	Realiza varias gráficas en una misma figura.
`title`	Título de la gráfica.
`upper left`	Posición de la leyenda izquierda superior.
`upper right`	Posición de la leyenda derecha superior.
`wireframe`	Gráfica de malla.
`xlim`	Límite de la coordenada x.
`xticks`	Valores de la coordenada x.
`ylim`	Límite de la coordenada y.
`zdir`	Dirección de z para visualización.
`zlim`	Límite de la coordenada z.

11.10 Conclusiones

En este capítulo se ha presentado la manera en que Python puede visualizar los datos. Esta es una de las características más importantes de Python. Se presentaron distintos tipos de gráficas que se pueden obteneren Python usando las bibliotecas `numpy` y `matplotlib`.

11.11 Ejercicios

1. Dados los puntos `y = [2, -2, 3, 8, 9, 0, 1]` obtenga la gráfica.
2. Dadas las parejas de puntos dados en el diccionario `{1:1, 2:1, 3:5, 4:9, 5:-10}` obtenga la gráfica.
3. Obtenga la gráfica de cos(x) de -2π a $+2\pi$. Use `linspace` con 100 puntos.
4. Repita el ejercicio anterior pero usando solamente 10 puntos. Compare los resultados.
5. Grafique la función $f(x) = x^2$ de x = -3 a x = +3. Añada una instrucción `legend` y una `label` apropiadas.
6. Grafique la función $\log(x)$ para $x > 0$. Use `linspace` para los valores de x.
7. Repita el ejercicio anterior para $\log10(x)$.
8. Repita el ejercicio anterior pero ahora use `logspace`. Compare los resultados.
9. Obtenga una gráfica de la función Chebyshev $C_2(x) = x^3 - 2x$ en el rango de valores x de -2 a +2.
10. En una misma figura obtenga gráficas de $f(x) = x^2 + 1$, $g(x) = \sqrt{x}$ y $h(x) = \text{sen}^2(x)$.
11. Para la función
$$r = \frac{2}{1 - r\cos\theta}$$
obtenga la gráfica polar.
12. Para la función
$$r = \frac{\frac{7}{2}}{2 + 4\ \text{sen}\ \theta}$$
obtenga la gráfica polar.
13. Para la función
$$r = 2 + 4\cos\theta$$
obtenga la gráfica polar.
14. Para la lista `[23, 45, 17, 15]` obtenga la gráfica de pie.
15. Para la lista `[0.3, 4.2, -1, 0, 7.8 ]` obtenga una gráfica usando la instrucción `stem`.

16. Para las ecuaciones

$$z = 1$$
$$x = r * sen(\theta)$$
$$y = r * cos(\theta)$$

obtenga una gráfica paramétrica en 3D.

17. Obtenga una gráfica de superficie para la función:

$$z = \frac{\text{sen}\sqrt{x^2 + y^2}}{x^2 + y^2}$$

en el rango para `(x, y)` de -10 a 10.

18. Realice una gráfica tipo `wireframe` para sen($\sqrt{|xy|}$).

19. Obtenga la gráfica anterior con una barra de color lateral.

20. Obtenga una gráfica de `contour` para

$$f(x, y) = \frac{1}{x^2 + y^2 + 1}$$

21. Obtenga una gráfica de `quiver` para el campo eléctrico cuyas coordenadas están dadas por:

$$\mathbf{E} = \left(\frac{\cos\theta}{\sqrt{x^2 + y^2}}, \frac{\text{sen}\ \theta}{\sqrt{x^2 + y^2}} \right)$$

Capítulo 12
Geolocalización y Análisis de Sentimientos

Con la colaboración de José Luis Velázquez García

12.1 Geolocalización

12.2 El módulo Geopy

12.3 Análisis de Sentimientos

12.4 La base de datos MongoDB

12.5 Análisis de los tweets

12.6 Conclusiones

Si no te pierdes, existe la posibilidad de que jamás seas hallado.

Anónimo

Objetivos

Entender el proceso de obtener información geográfica real acerca de cualquier punto de referencia en el mundo utilizando **geopy**, el cual es un módulo en Python para el acceso a distintas herramientas (libres) de ubicación o posicionamiento geográfico. Realizar análisis de sentimientos en tweets, así como extraer, analizar y almacenar información de Twitter con el propósito de realizar un análisis de sentimientos utilizando distintos módulos de Python como Tweepy, MongoDB y MeaningCloud.

12.1 Geolocalización

La geolocalización es la capacidad para obtener la ubicación geográfica real y latitud y longitud de un objeto o lugar tomando como referencia distintos dispositivos de rastreo global como satélites o torres de radiocomunicación, entre otros. Este proceso es normalmente empleado por distintos sistemas de información geográfica con el propósito de entender en que parte del mundo ocurren distintos fenómenos. En el caso de las redes sociales, la geolocalización se ha convertido en una herramienta muy útil para categorizar y entender donde se encuentran las concentraciones más altas de usuarios de las distintas redes. En este sentido el análisis de este tipo de información referenciada puede ser de gran ayuda para entender distintos problemas sociales asociados al uso de datos en la web, por lo que en esta sección se presenta **geopy** el cual es una herramienta muy útil para la extracción de este tipo de datos. En específico se mostrarán las distintas variantes y modalidades de esta herramienta para obtener información referenciada tomando en cuenta el lenguaje de programación Python.

12.2 El módulo geopy

geopy es un módulo en Python especialmente diseñado para obtener información geolocalizada de ciudades, países o cualquier dirección a lo largo de todo el mundo

a través de una conexión tipo cliente servidor con otras populares herramientas de geolocalizacion como OpenStreetMap Nominatim, Google Geocoding API (V3), Bing Maps, Yahoo! PlaceFinder, ArcGIS, entre otros. geopy está especialmente diseñado para obtener puntos de referencia de cualquier lugar así como también la medición de la distancia entre dichos puntos. En las siguientes secciones se presentan ejemplos de cómo poder usar este módulo para extraer información geolocalizada utilizando código Python (en el apéndice A se puede ver a detalle la forma de instalar geopy).

12.2.1 Geolocalización de un punto de interés

Dentro de todas las posibles conexiones de geopy con herramientas de geolocalización, `OpenStreetMap Nominatim` ofrece la posibilidad de obtener la latitud y longitud asociada a una cadena de texto que contenga una ciudad, país o cualquier dirección que proporcionemos, para ello lo primero que haremos será crear un objeto de tipo `Nominatim`. Teniendo este objeto accederemos a la función de `geocode`, la cual necesita como argumento la cadena de texto antes mencionada. El resultado de esta operación es un objeto que contiene el lugar proporcionado en la cadena de texto, la latitud y longitud como propiedades de la clase. En el siguiente programa se ejemplifica lo anterior. La figura 12.1 muestra el resultado.

```
"'Se importa dentro del programa el módulo de geopy
para acceder a OpenStreetMap Nominatim"'
from geopy.geocoders import Nominatim
# Se genera un objeto de la clase Nominatim
geolocator = Nominatim()
# Se crea un string con el lugar a buscar
place = "Ciudad de Mexico"
# Se utiliza la función geocode del objeto para obtener la latitud y longitud
location = geolocator.geocode(place)
# Se imprime la información obtenida
print("lugar: "+place+", coordenadas: "+str(location.latitude)+","+
        str(location.longitude))
```

En el caso de que tengamos múltiples locaciones (como en los tweets a extraer en la siguiente sección) es necesario importar y agregar la instrucción `time.sleep(1)` la cual hará que después de cada conexión exitosa con `OpenStreetMap Nominatim` espere un lapso de tiempo de un segundo para obtener la siguiente locación a buscar, esto hará que no sobrepasemos el número de conexiones máximas en un periodo de tiempo con geopy. Además de lo anterior, agregaremos en la función `geocode` un parámetro extra llamado `timeout=10` el cual esperará la conexión con geopy un periodo de tiempo de quince segundos y si después se cumple ese lapso se lanzará una

excepción y se pasará a la siguiente locación. En el siguiente programa (ver figura 12.2) se ejemplifica lo anterior:

```
# Se importa el módulo time
import time
from geopy.geocoders import Nominatim
# Se genera un objeto de la clase Nominatim
geolocator = Nominatim()
# Se crea una lista de lugares a buscar
places = ["Mexico","Buenos Aires Argentina","175 5th Avenue NYC"]
"'Se utiliza la función geocode del objeto para obtener la latitud y longitud
de cada lugar"'
for x in places:
      location = geolocator.geocode(x,timeout=10)
      # Se imprime la información obtenida
      print("lugar: "+x+", coordenadas: "+str(location.latitude)+
            ","+ str(location.longitude))
      time.sleep(1)
```

Figura 12.1 Geolocalización de un punto de interés.

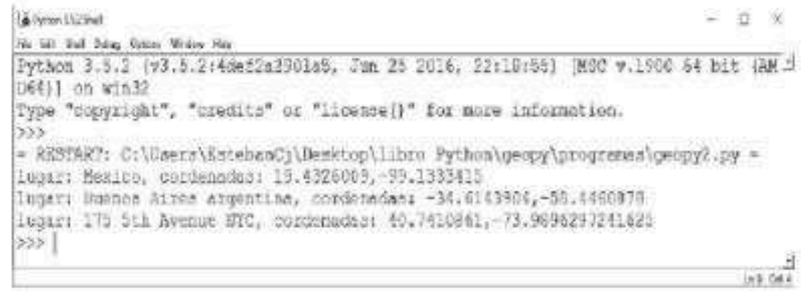


Figura 12.2 Geolocalización de varios puntos de interés.

12.2.2 Distancia de dos puntos de interés

Además de poder calcular la posición de un punto específico, geopy es también capaz de analizar y obtener el camino más corto entre dos puntos (en millas) inicializando un objeto de tipo `vincenty` o de tipo `great_circle`[1] , los cuales son dos algoritmos muy conocidos para calcular la distancia entre puntos tomando como referencia la estructura terráquea. En el siguiente programa se ejemplifica lo anterior:

```
'''Se importa dentro del programa el módulo de geopy
para acceder al algoritmo vincenty o great_circle'''
from geopy.distance import vincenty
from geopy.distance import great_circle
''' Se generan dos objetos con las coordenadas de Argentina
y México en formato de tupla'''
Argentina = (-34.9964962, -64.9672816)
Mexico = (19.4326009, -99.1333415)
# Se imprime la distancia en millas usando vincenty
print("Distancia entre México y Argentina (vincenty): "+
          str(vincenty(Argentina, Mexico).miles))
# Se imprime la distancia en millas usando great_circle
print("Distancia entre México y Argentina (great_circle): " +
          str(great_circle(Argentina, Mexico).miles))
```

[1] https://geopy.readthedocs.io/en/1.10.0/#module-geopy.distance

La figura 12.3 muestra una corrida del programa.

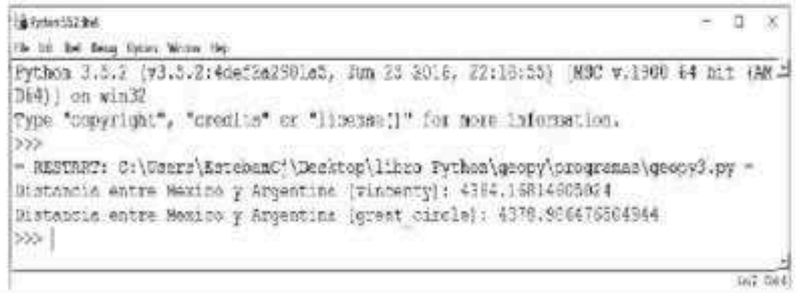


Figura 12.3 Distancia entre dos puntos de interés.

12.2.3 Visualización de distintos puntos de interés

Una de las opciones más interesantes asociadas a la extracción de información geolocalizada es poder visualizar su distribución en un mapa con el propósito de analizar el comportamiento de cierto problema o fenómeno. Tomando lo anterior en cuenta, en esta subsección se presenta una forma de poder mostrar los datos geolocalizados utilizando el módulo `basemap` (en el apéndice A se puede ver a detalle la forma de instalarlo), el cual genera una cartografía personalizada con los puntos de interés asociados. En el siguiente programa se ejemplifica la forma de crear dicho mapa utilizando el módulo `basemap` de Python:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np
# Latitud y longitud de varios puntos de interés
# México, Argentina y 175 5th Avenue NYC
lat = [19.4326009,-34.9964962,40.7410861]
lon = [-99.1333415,-64.9672816,-73.9896297241625]
# Se inicializa dimensión de la figura
plt.figure(figsize = (16,12))
'''Se inicializa:
1. El tipo de cartografía (robin)
```

2. La resolución
3. Las distribución de líneas costeras"'

```
eq_map =Basemap(projection='robin',
      lon_0 = 0,
      resolution = "h",
      area_thresh = 1000.0,
      llcrnrlon = -136.25,
      llcrnrlat = 56,
      urcrnrlon = -134.25,
      urcrnrlat = 57.75)
# Se dibujan las líneas costeras y países
eq_map.drawcoastlines()
eq_map.drawcountries()
# Se define el color de los países
eq_map.fillcontinents(color = "#e6e6ff")
eq_map.drawmapboundary()
# Se dibujan los meridianos y paralelos
eq_map.drawmeridians(np.arange(0, 360, 30))
eq_map.drawparallels(np.arange(-90, 90, 30))
"'Se dibuja cada uno de los puntos utilizando la
instrucción zip la cual empaqueta varias listas para
después iterarlas paralelamente."'
for lon, lat in zip(lon, lat):
      #Se crea un objeto de la clase eq_map
      x,y = eq_map(lon, lat)
      "' Se dibuja un circulo de color rojo (ro)
      con un radio de 17 y una transparencia de 0.8"'
      eq_map.plot(x, y, "ro", markersize = 17, alpha = 0.8)
# Se salva la imagen en formato png de alta resolución
plt.savefig("visualizacionDatos.png",bbox_inches = "tight",dpi = 100)
```

El resultado se muestra en la figura 12.4.

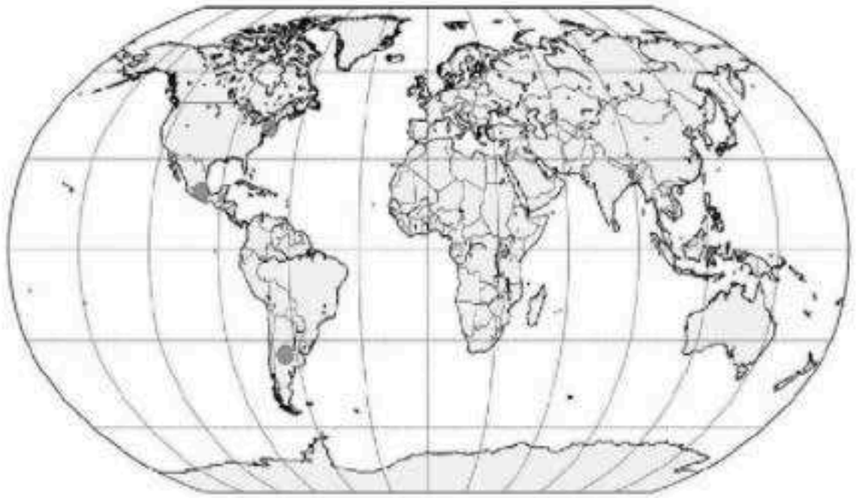


Figura 12.4 Visualización de distintos puntos de interés.

12.3 Análisis de sentimientos de Twitter

En la actualidad, el uso de las redes sociales tiene un gran impacto en la manera como la gente interactúa. Se usa para comunicar información a los amigos como en Facebook. También se usa para emitir opiniones o comunicar sucesos en tiempo real como se hace con Twitter. Estas dos redes sociales son sólo ejemplos de redes sociales. La información disponible en redes sociales se puede usar para conocer la tendencia en las opiniones emitidas por los usuarios de estas redes. En esta sección mostramos como se puede usar Python para realizar un análisis de sentimientos expresados por los usuarios en Twitter. Para este propósito requerimos software adicional que necesita instalarse así como generar código en Python[2] para la extracción de tweets, su almacenamiento y su posterior análisis.

[2]Esta sección requiere el uso de Python 2.7.

12.3.1 Extracción de tweets

Para poder extraer tweets es necesario hacerlo desde una aplicación de Twitter. Esta aplicación se registra en Twitter para obtener dos claves de acceso conocidas como `Access Token` y `Access Token Secret`. Para ilustrar el procedimiento supongamos una aplicación llamada mipagina.com. Entonces lo primero que necesitamos es entrar a nuestra cuenta de Twitter. Ahora podemos crear la aplicación. Ingresamos a la página `http://apps.twitter.com` y seleccionamos "Create New App". Le damos a nuestra aplicación el nombre `Prueba_Extraccion_libro_Python` como se muestra en la figure 12.5. Llenamos los espacios que se solicitan, aceptamos las condiciones y presionamos el botón `Create your Twitter application`. Esto nos genera la aplicación como se muestra en la figura 12.6. Ahí también aparece la clave en `Consumer Key`. Seleccionando la pestaña de `Keys and Access Tokens` obtenemos una nueva ventana mostrada en la figura 12.7 con las dos claves: `Consumer Key` y `Consumer Secret`.

Ahora necesitamos instalar una aplicación que nos permita acceder al API de Twitter. Esta aplicación es `tweepy` que está escrita en Python y que es software libre. La descargamos e instalamos en una ventana de comandos con `pip` como se explica en el apéndice A.

Figura 12.5 Formato para la creación de la aplicación.

Figura 12.6 Datos de la aplicación.

Figura 12.7 Datos de key y access tokens.

El siguiente paso es crear una clase heredando de la clase `MyStreamListener`, autenticar con las credenciales de la aplicación, iniciar el *stream* y observar en la ventana de comandos los tweets. Para la clase Python `MyStreamListener` primero necesitamos los encabezados que importan las bibliotecas o módulos necesarios y la autenticación:

```
import sys
import tweepy
import requests
import codecs

#llaves del API Twitter
consumer_key = 'kmgTaybmRu95EzBmeXHDDwtHr'
consumer_secret = 'tB1IYP5f6jpLnOSmvVmNplYGqouNe4sG4lvTFOvgje1M9Ie6mv'
```

```
access_token = '166739399-DwW4yMFyiajFBpKKHoOldO1e4t64s31jXrtgvdjD'
access_secret = 'aDgp1tZxoeqk7XIbcBf9V87S17HEhfB5gWRXXGbIGMxn2'
```

El siguiente paso es realizar las acciones que nos permitan acceder a la API y obtener los tweets de interés. Esta es la clase `MyStreamListener`:

```
class MyStreamListener(tweepy.StreamListener):
# Extender la clase streamListener

  def on_status(self, status):#Procesar nuevos tweets
    if not hasattr(status, 'retweeted_status' ):#Ignorar retweets
      # Codificar en UTF-8 el mensaje antes de imprimirlo
      print status.text.encode('utf-8

  def on_error(self, status_code):
    if status_code == 420:
      print " Número de intentos excesivos de conectarse al "
      print " streaming API, esperar y ejecutar de nuevo..."
    elif status_code == 401:#
      print " Credenciales de API incorrectas."
    else:
      print "Ocurrió un error"
    return False # Seguir la ejecucion.
              # True cancelaría la ejecución del programa

  def on_timeout(self):
    print "Timeout..."
    return True # Seguir la ejecución.
              # False cancelaría la ejecución del programa

tauth = tweepy.OAuthHandler(consumer_key, consumer_secret)#Autenticar
tauth.set_access_token(access_token, access_secret)#Autenticar
streamListen = MyStreamListener()#Instanciar clase "MyStreamListener"


#Crear stream
```

```
twStream = tweepy.Stream(auth = tauth, listener = streamListen )


#Especificar filtro & Iniciar Stream
twStream.filter(track=['#Puebla, #EPN, #Mexico, #UDLAP'], async=False)
```

Salvamos el programa con el nombre de `captura.py` en el fichero que hemos usado anteriormente `Analisis_Sentimientos`. Estamos listos para utilizar el API Stream de Twitter usando `Tweepy`. Para esto corremos nuestro programa de Python desde una ventana de comandos. En el último renglón del programa hemos seleccionado las palabras Puebla, EPN, México y UDLAP como las palabras de las que queremos saber los sentimientos de los usuarios de Twitter. Lo ejecutamos en la ventana de comandos con:

```
> C:\Analisis_Sentimientos\python captura.py
```

Lo que obtenemos son los tweets que los usuarios generan conteniendo las palabras deseadas: `#Puebla, #EPN, #Mexico, #UDLAP`. El resultado se muestra en la figura 12.8.

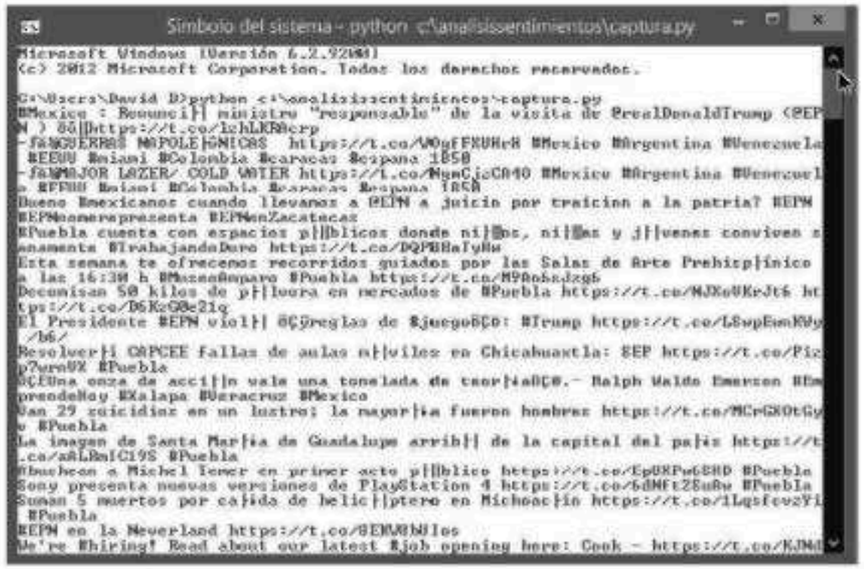


Figura 12.8 Tweets capturados.

12.3.2 La base de datos MongoDB

El siguiente paso es almacenar en una base de datos. Dado que los tweets no tienen una estructura definida se requiere usar una base de datos no estructurada como la que usa Mongo. Entonces vamos a la página http://www.mongodb.com de donde descargamos la aplicación que corresponda con la configuración de nuestro equipo. Para poder almacenar los tweets capturados debemos modificar nuestro programa captura.py para almacenarlos en la base de datos de Mongo. Para esto tenemos que instalar la biblioteca pymongo usando pip como se explica en el Apéndice A.

En el programa captura.py incluimos las bibliotecas que vamos a usar que son:

```
import sys
import tweepy
import requests
import codecs
from pymongo import MongoClient # Librería Mongodb
```

Configuramos la conexión a la base de datos Mongo con:

```
# Conexión a MongoDB
cliente = MongoClient() #Inicializar objeto
# Indicar parametros del servidor.
cliente = MongoClient('127.0.0.1', 27017)
bd = cliente.twitter # Seleccionar Schema.
tweets = bd.tweets #Seleccionar Colección.
```

La dirección '127.0.0.1', 27017 es donde Mongo almacena los tweets. Ahora modificamos el método on_status de la siguiente manera:

```
def on_status(self, status):#Procesar nuevos tweets
  if not hasattr(status, 'retweeted_status'):# Ignorar retweets
    print status.source.encode('utf-8')+"<>"+status.text.encode('utf-8')
    #Crear Objeto
```

```
tweet = {
  "idt": str(status.id),
  "tweet": status.text.encode('utf-8'),
  "fecha_creacion": status.created_at,
  "sentimiento": "";}
tweets.insert_one(tweet) # Almacenar tweet
```

Salvamos las modificaciones en el archivo `almacenar.py` en `Analisis_Sentimientos`. Los tweets se van a almacenar en un fichero con el nombre de `db` dentro de otro fichero `data` que debe estar en el directorio raíz, `C:` en Windows y `bash` en Mac. Los pasos son los siguientes:

- Abrir una ventana de comandos y correr `mongod`. Esto es para que la base de datos se abra y empiece a almacenar los tweets en `C:\data\db`.

```
> C:\Program Files\mongodb\Server\3.2\bin\mongod -dbpath C:\data\db
```

Figura 12.9 `mongod` listo para almacenar tweets.

Dependiendo del sistema operativo, algunas computadoras podrían requerir en su lugar escribir en la ventana de comandos:

```
> C:\Program Files\mongodb\Server\3.2\bin
```

```
\mongod -StorageEngine=mmapv1 -dbpath C:\data\db
```

Para Mac requerimos primero crear el fichero db. La instrucción es:

```
mkdir /RUTA/db

sudo /RUTA/bin/mongod -dbpath/RUTA/db
```

o simplemente:

```
sudo /RUTA/bin/mongod -dbpath /RUTA/
```

donde RUTA es la ubicación de la carpeta de `mongo` en la computadora. Para Mac se muestra el resultado en la Figura 12.9

- Abrir una nueva ventana de comandos y ejecutar el archivo `almacenar.py` desde Python con:

```
> C:\Analisis_Sentimientos\python almacenar.py
```

para esto debemos estar en el directorio de `Analisis_Sentimientos`.

- La figura 12.10 muestran como se empiezan a capturar los tweets.

```
dbaeziec — almacenar.py — 80×24
Last login: Sat Sep 10 12:18:01 on ttys020
[DavidBaezMacBook:~ dbaeziec$ python /Users/dbaeziec/analisis_sentimientos/almacen
ar.py
Twitter for iPhone<>Cifras 12 pm #MarchaForLaFamilia :
- 30 mil Tijuana
- 80 mil Querétaro
- 10 mil Toluca
- 20 mil Tuxtla Gtz
- 3 mil Atlacomulco
@FPN #Mexico
IFTTT<>Best #Sportsbook #NFL #NBA #MLB #NHL #Football #worldFootball #Soccer #Fu
tbol #Futebol https://t.co/[illegible] https://t.co/h2xiT9T6JI
Twitter Web Client<>Cuando ves a Tu País #Mexico en las alturas por #SuperHumano
s #Oro #Mexico #Paralympics @AmaliaPerez @Rio2016_en https://t.co/bmmw7UXDb7
Twitter for iPhone<>Gracias! @CanalOnceTV por la transmisión del partido de @Ag
uilasIPN somos muchos los fans del fútbol americano estudiantil en #Mexico
Facebook<>100 metros masculino #Mexico gana bronce [illegible]
[illegible] #SalvadorHernandezManDragon... https://t.co/nUjihCnWnE
Twitter for Android<>Sub13 @NacionFC507 en su partido en #CopaPuma2016 #Panama #
Futbol nuestra pasion. #Futbol #KingsSports https://t.co/SCVylisuiw
Twitter for Android<>EN BULGARIA, #Mexico listo para el Mundial de Carreras de
Montaña.Mañana 11Sep. #Atletismo @CCM_Mexico https://t.co/Vk10Mrk3O2
TLRCommunitySA<>[illegible]ONOCIENDO EL VATICANO https://t.co/BrCIltgiNv #Mexico #Argenti
no #Venezuela #CCLU #miami #Colombia #caracas #espana 1320
```

Figura 12.10 Tweets capturados.

12.3.3 Análisis de los tweets

Una vez descargados los datos, procedemos a analizarlos usando `MeaningCloud`. Esta es una compañía que ofrece productos SAAS (Service As A Software) enfocados al análisis semántico de texto, permitiendo a usuarios embeber el procesamiento en cualquier aplicación o sistema. En particular estamos interesados en usarlo para el análisis de sentimientos, para indicar la polaridad del texto, su subjetividad, ironía y expresión de desacuerdo.

Para usar MeaningCloud debemos registrarnos. Para dar inicio al registro, vamos a la página `https://www.meaningcloud.com`. Creamos una cuenta gratuita como desarrolladores y de esta manera nos asigna una cuenta después de confirmarla. Esta cuenta es nuestro *payload*(ver figura 12.11). Procedemos ahora a nuestro archivo en Python que llamamos `Analisis.py`.

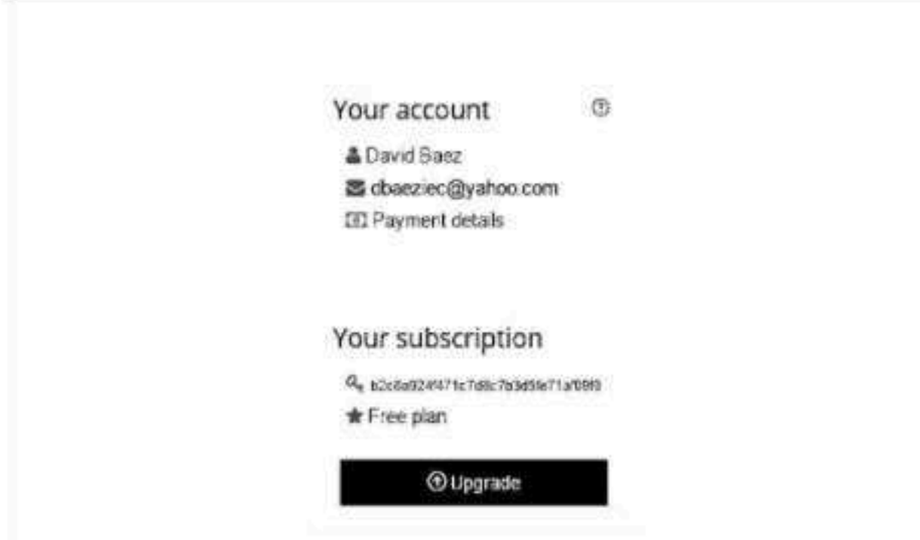


Figura 12.11 Cuenta de MeaningCloud.

Los encabezados son:

```
import sys
import requests
import demjson
```

Para usar `demjson` primero tendremos que importarlo desde la ventana de comandos como veremos más adelante. Continuamos con la conexión a Mongo como en el archivo `Captura.py`:

```
# Conexión a MongoDB
cliente = MongoClient()# Inicializar objeto.
#Indicar parámetros del servidor.
cliente = MongoClient('127.0.0.1', 27017)
bd = cliente.twitter # Seleccionar Schema.
tweets = bd.tweets # Seleccionar Colección.
```

Definimos el método para el llamado a `MeaningCloud`:

```
def sentimiento(tweet):
   url = "http://api.meaningcloud.com/sentiment-2.1"
```

Figura 12.11 Cuenta de MeaningCloud.

Los encabezados son:

```
import sys
import requests
import demjson
```

Para usar `demjson` primero tendremos que importarlo desde la ventana de comandos como veremos más adelante. Continuamos con la conexión a Mongo como en el archivo `Captura.py`:

```
# Conexión a MongoDB
cliente = MongoClient()# Inicializar objeto.
#Indicar parámetros del servidor.
cliente = MongoClient('127.0.0.1', 27017)
bd = cliente.twitter # Seleccionar Schema.
tweets = bd.tweets # Seleccionar Colección.
```

Definimos el método para el llamado a `MeaningCloud`:

```
def sentimiento(tweet):
   url = "http://api.meaningcloud.com/sentiment-2.1"
```

```
    headers = { 'content-type': 'application/x-www-form-urlencoded'}
    payload = "key=143d556c90b3089eab&lang=es&txt="+tweet+"&model="
    response=requests.request("POST", url,data=payload,headers=headers)
    return response
```

Finalmente, realizamos el *parsing* de cada tweet:

```
for tweet in tweets.find({"$where": "this.sentimiento.length == 0"}):
      print "Tweet:"+tweet['tweet']
      while True:# Analizar sentimiento de tweet.
         # Analizar tweet con el API.
         resultado = sentimiento(tweet['tweet'].encode('utf-8'))

         # Decodificar la respuesta JSON.
         json = demjson.decode(resultado.text)

         # Obtener el código de estado de la respuesta.
         estado = json['status']['code'];
         if estado == '104':
            print "Error 104 se excedió el límite de."
                  2 solicitudes/segundo, esperando 5 segundos..."
            time.sleep( 5 )
         elif estado != 0 # Ocurrió otro tipo de error.
            print "Ocurrió un error "+str(estado) + ", saliendo..."
            sys.exit(0);
         else:# Respuesta correcta
            print "Sentimiento:"+json['score_tag']
            tweets.update({"idt":tweet['idt']},
                    {'$set':{'sentimiento':json['score_tag']}})
            break
```

El archivo completo es:

```
import sys
import requests
```

```
import demjson

from pymongo import MongoClient# Libreria Mongodb

# Conexión a MongoDB
cliente = MongoClient()# Inicializar objeto.
#Indicar parámetros del servidor.
cliente = MongoClient('127.0.0.1', 27017)
bd = cliente.twitter # Seleccionar Schema.
tweets = bd.tweets # Seleccionar Colección.

def sentimiento(tweet):
   url = "http://api.meaningcloud.com/sentiment-2.1"
   headers = { 'content-type': 'application/x-www-form-urlencoded'}
   payload = "key=143d556c90b3089eab&lang=es&txt="+tweet+"&model="
   response=requests.request("POST", url,data=payload,headers=headers)
   return response

# Seleccionar tweets que no tienen sentimiento.
for tweet in tweets.find({"$where": "this.sentimiento.length == 0"}):
      print "Tweet:"+tweet['tweet']
      while True:# Analizar sentimiento de tweet.
        # Analizar tweet con el API.
        resultado = sentimiento(tweet['tweet'].encode('utf-8'))

        # Decodificar la respuesta JSON.
        json = demjson.decode(resultado.text)

        # Obtener el código de estado de la respuesta.
        estado = json['status']['code']';
        if estado == '104':
           print "Error 104 se excedió el límite de."
                2 solicitudes/segundo, esperando 5 segundos..."
           time.sleep( 5 )
        elif estado != 0 # Ocurrió otro tipo de error.
```

```
        print "Ocurrió un error "+str(estado) + ", saliendo..."
        sys.exit(0);
    else:# Respuesta correcta
        print "Sentimiento:"+json['score_tag']
        tweets.update({"idt":tweet['idt']},
                {'$set':{'sentimiento':json['score_tag']}})
        break
```

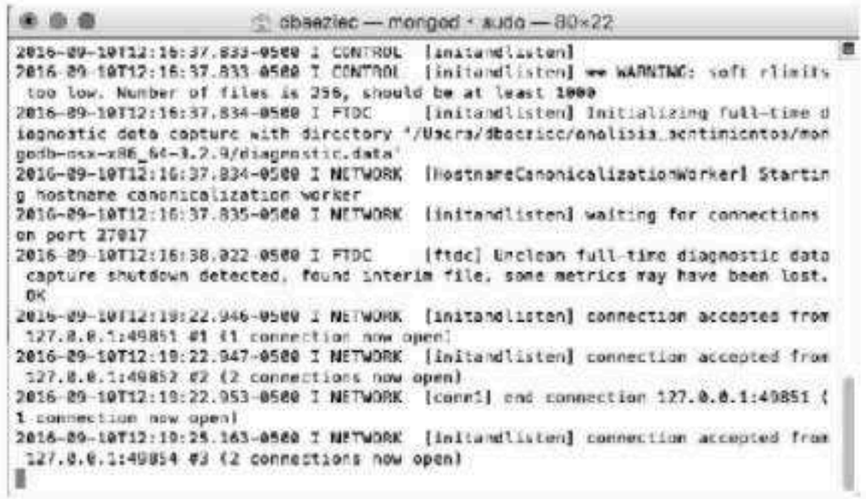


Figura 12.12 Mongo mostrando las conexiones abiertas.

Antes de correr el archivo `analisis.py` necesitamos tener abierta la base de datos corriendo `mongod` desde una ventana de comandos.

También debemos instalar `demjson` usando `pip` desde la ventana de comandos:

```
> pip -m install demjson
```

Debemos abrir una nueva ventana de comandos y en `Analisis_Sentimientos` lo corremos como:

```
> python analisis.py
```

La figura 12.12 muestra que ya hay conexiones en el mongod. La figura 12.13 muestra una corrida con los resultados del sentimiento de cada tweet.

```
o/gwCH3AeC4g #futbol https://t.co/7y9LR5YwR3
Sentimiento:P
Tweet:¡Conoce su distribución inteligente! #Casiopea #Puebla
Informes y citas: 1800343 https://t.co/LBCCyEHPkU
Sentimiento:P
Tweet:Zidane: "Ovacionaron a Madrid no sólo por hoy. Fue merecido" https://t.co/
WhpnsgWjl3 #AS #futbol
Sentimiento:P
Tweet:@DailySkip @ESNCPI #Thanks 4following we gotch back N #Texas &amp; #Mexico
Sentimiento:NONE
Tweet:#mexico Expo Auto Guanajuato dejará derrama económica por más de 18 mdp ht
tps://t.co/z83gLs25iW @Espanaprensa
Error 104 se excedió el límite de 2 solicitudes/segundo, esperando 5 segundos...
Sentimiento:NONE
Tweet:#mexico Suman 88 mil mexiquenses que logran empleo en ferias https://t.co/
8CPNRcOZDt @Espanaprensa
Sentimiento:P
Tweet:#mexico Lento inicio del Team Chile en los Juegos Paralímpicos de Río http
s://t.co/BtwRqeylAp @Espanaprensa
Error 104 se excedió el límite de 2 solicitudes/segundo, esperando 5 segundos...
Sentimiento:N
Tweet:#mexico Emite volcán Popocatépetl 158 exhalaciones de baja intensidad http
s://t.co/mfuKzreRFY @Espanaprensa
```

Figura 12.13 Tweets con sentimiento indicado.

Los resultados del análisis son de la siguiente manera:

Resultado	Significado
P+	Muy positivo
P	Positivo
NEU	Neutro
N	Negativo
N+	Muy negativo
NONE	Ninguno

12.4 Conclusiones

En este capítulo hemos mostrado aplicaciones de Python en gelocalización y análisis de sentimientos. Estas aplicaciones, al igual que muchas otras más, requieren de módulos disponibles de manera gratuita y que permiten realizar una gran cantidad de funciones y aplicaciones de Python. En particular, este capítulo hace uso de `geopy` para la geolocalización, de `tweepy` para el análisis de tweets y de `pymongo` para el uso de la base de datos de MongoDB.

Apéndice A
Instalación y configuración

A.1 Introducción
A.2 Instalación de Python
A.3 Instalación de easy_ install y pip
A.4 Instalación de Numpy
A.5 Instalación de Scipy
A.6 Instalación de Matplotlib
A.7 Instalación de Tweepy
A.8 Instalación de Pymongo
A.9 Instalación de Geopy
A.10 Instalación de Matplotlib Basemap

A.1 Introducción

En este apéndice se provee la información necesaria para instalar Python en una computadora con un sistema operativo Windows 7, 8 y 10, Unix/Linux o Mac OS X, así como se cubren los detalles concernientes a la instalación de los paquetes usados a lo largo del libro (Numpy, Scipy, Matplotlib, etc).

A.2 Instalación de Python

Para instalar Python es necesario descargarlo y configurarlo. A diferencia de otros programas, Python forma parte del llamado software libre, por lo que se puede encontrar de manera gratuita, así como la instalación es bastante sencilla independientemente del sistema operativo que use. Para poder instalar Python en su ordenador siga los siguientes pasos:

- **Windows**

  1. Se verifica si la arquitectura del sistema operativo es de 32 o 64 bits, para ello, escriba *Sistema* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows). Posteriormente verifique el tipo de arquitectura en la opción *Tipo de sistema*.
  2. Diríjase a la página https://www.python.org/downloads/windows/
  3. Elija la última versión de Python (la versión 3.5 al momento de escribir este libro y denotamos por 3.x) y seleccione la liga del archivo de instalación:
     a) Sistema de 32 bits: **Windows x86 MSI installer**
     b) Sistema de 64 bits: **Windows x86-64 MSI installer**
  4. Después de descargar el archivo (extensión .exe), haremos doble click sobre este y aparecerá el cuadro de dialogo de instalación de Python. Seleccionamos la opción personalizada: "Customize installation", como se muestra en la figura 1.
  5. Dentro de la opción personalizada seleccionamos todas las características adicionales como se muestra en la figura 2, después seguimos las opciones por defecto hasta que nos pida el directorio de instalación.
  6. Escogemos como directorio de instalación `C:\Python3x\` y seguiremos las opciones por defecto restantes del instalador[1].

[1] Puede escoger otro directorio de instalación pero deberá de tomarlo en cuenta para instalar `easy_install` y `pip`.

7. Para comprobar la instalación del programa, basta con acceder a la pestaña de *todas las aplicaciones* del icono de inicio de Windows y buscar la carpeta de Python, la cual contendrá la interfaz gráfica de programación de Python (IDLE) así como el acceso a la línea de comandos de Python y la forma de desinstalarlo.

Figura 1 Selección de instalación *personalizada (custom)*.

Figura 2 Selección de todas las casillas de características adicionales.

- **Mac OS X**

1. Se verifica si la arquitectura del sistema operativo es de 32 o 64 bits, para ello, seleccione el icono de Mac (junto a la pestaña de *Finder*) después

seleccione la opción de *Acerca de esta Mac*, posteriormente escoja la opcion de *Mas información* y busque la pestaña de *Procesador*, si esta indica que tiene un procesador core o core duo entonces la arquitectura del sistema será de 32 bits y en caso contrario será de 64 bits.

2. Diríjase a la página https://www.python.org/downloads/mac-osx/
3. Elija la última versión de Python (la versión 3.5 al momento de escribir este libro y que denotamos por 3.x) y seleccione la liga del archivo de instalación:
   a) Sistema de 32 bits: **Mac OS X 32-bit i386/PPC installer**
   b) Sistema de 64 bits: **Mac OS X 64-bit/32-bit installer**
4. Después de descargar el archivo (extensión .pkg), haremos doble click sobre este y aparecerá el cuadro de dialogo de instalación de Python, del cual seguiremos las opciones por defecto.
5. Para comprobar la instalación del programa abra una ventana de finder y acceda a la carpeta de Aplicaciones en la cual aparece el icono de Python. Al presionarlo aparecen los iconos de la interfaz gráfica de programación de Python (IDLE) así como un archivo HTML de documentación y un script que nos permitirá fijar la versión 3.x como la versión por defecto.

- **Unix y Linux**

1. Abra una terminal (shell).
2. Escriba el siguiente comando `sudo apt-get install python3`
3. Proporcione su nombre de usuario y contraseña y posteriormente espere a que termine la instalción.
4. Para comprobar la instalación del programa escriba en la terminal `python`

A.3 Instalación de easy_install y pip

Con el propósito de poder instalar paquetes o programas que no forman parte de la instalación original de Python, se ha puesto a disposición de los desarrolladores, un repositorio de paquetes de fácil acceso llamado **PyPi**[2], así como dos herramientas para poder instalar programas de este repositorio llamadas **easy_install**[3] y **Pip**[4].

En la instalación de Python3.x ya se incluye `pip`, el cual nos sirve para instalar `numpy` y `matplotlib` que se emplean en los capítulos 11 y 12.

[2]https://pypi.python.org/pypi
[3]https://pypi.python.org/pypi/setuptools
[4]https://pip.pypa.io /en/latest/installing.html

Tomando en cuenta que algunas versiones de Python3.x no tienen instalado `pip` de forma automática, en esta sección se muestran los pasos para instalar tanto `pip` como `easy install` en los distintos sistemas operativos para posteriormente utilizarlas en la instalación de algunos paquetes que son utilizados a lo largo del libro, como son.

- **easy_install en Windows**

  1. Verifique si en la carpeta *Scripts* en `C:\Python3x\Scripts` (véase la sección de instalación de Python) existen los archivos llamados `easy_install-3.x` y `easy_install`. En caso de existir puede pasar al paso 6.
  2. Diríjase a la página https://bootstrap.pypa.io/ez_setup.py.
  3. Copie el contenido de la página en un archivo de Python con el nombre `ez_setup.py`.
  4. Abra la interfaz gráfica de Python, escribiendo `IDLE` en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
  5. Abra el archivo `ez_setup.py` dentro de la interfaz grafica de Python y oprima le tecla F5 (ejecutar script).
  6. Diríjase a la ventana de variables de entorno del sistema de Windows, para ello, escriba *Editar las variables de entorno del sistema* en el recuadro de *Buscar en la web y en Windows*. Posteriormente seleccione la opción de variables de entorno.
  7. Seleccione la variable *path* en el recuadro de variables del sistema y elija la opción de *Editar*.
  8. Seleccione la opción *Nuevo* y agregue `C:\Python3x\Scripts`
  9. Acepte los cambios realizados sobre las variables de entorno y abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows*.
  10. Dentro de la terminal ejecute la instrucción `easy_install`. Si aparece la siguiente leyenda entonces lo habrá instalado de manera correcta:

      ```
      error: No urls, filenames, or requirements specified.
      ```

- **pip en Windows**

  1. Verifique si en la carpeta *Scripts* en `C:\Python35\Scripts` (véase la sección de instalación de Python) existen los archivos llamados `pip3.3`, `pip2` y `pip`. En caso de existir pude pasar al paso 6.
  2. Diríjase a la página https://bootstrap.pypa.io/get-pip.py.
  3. Copie el contenido de la página en un archivo de Python con el nombre get-pip.py.

4. Abra la interfaz gráfica de Python, escribiendo *IDLE* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
5. Abra el archivo *get-pip.py* dentro de la interfaz grafica de Python y oprima le tecla F5 (ejecutar script).
6. Diríjase a la ventana de variables de entorno del sistema de Windows, para ello, escriba *Editar las variables de entorno del sistema* en el recuadro de *Buscar en la web y en Windows*. Posteriormente seleccione la opción de variables de entorno.
7. Seleccione la variable *path* en el recuadro de variables del sistema y elija la opción de *Editar*.
8. Seleccione la opción *Nuevo* y agregue[5] `C:\Python3x\Scripts`
9. Acepte lo cambios realizados sobre las variables de entorno y abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows*.
10. Dentro de la terminal ejecute la instrucción `pip install`. Si aparece la siguiente leyenda entonces lo habrá instalado de manera correcta:

    ```
    You must give at least one requirement to install.
    ```

    Otro mensaje podría aparecer y de cualquier manera está bien la instalación.

- **`easy_install` en Mac OS X/ Unix y Linux**

  1. Diríjase a la página https://bootstrap.pypa.io/ez_setup.py.
  2. Copie el contenido de la página en un archivo de Python con el nombre `ez_setup.py`.
  3. Ejecute el archivo de Python, desde una terminal de Mac o una terminal (shell) de Unix/Linux utilizando la siguiente instrucción:
     `sudo python ez_setup.py`
  4. Dentro de una terminal de Mac o una terminal (shell) de Unix/Linux ejecute la instrucción `easy_install`. Si aparece la siguiente leyenda entonces lo habrá instalado de manera correcta:

     ```
     error: No urls, filenames, or requirements specified
     ```

- **pip en Mac OS X/ Unix y Linux**

[5] Si previamente instaló `easy_install`, entonces ya no será necesario volver a agregar la referencia a esa carpeta en la variable del sistema.

1. Diríjase a la página https://bootstrap.pypa.io/get-pip.py.
2. Copie el contenido de la página en un archivo de Python con el nombre get-pip.py.
3. Ejecute el archivo de Python, desde una terminal de Mac o una terminal (shell) de Unix/Linux utilizando la siguiente instrucción:
   `sudo python get-pip.py`
4. Dentro de una terminal de Mac o una terminal (shell) de Unix/Linux ejecute la instrucción `pip install`. Si aparece la siguiente leyenda entonces lo habrá instalado de manera correcta:

   ```
   You must give at least one requirement to install
   ```

A.4 Instalación de Numpy

A continuación se presenta los pasos para poder instalar el paquete **Numpy**[6], el cual es una biblioteca libre para trabajar con funciones matemáticas de alto nivel así como para operar con vectores o matrices de una manera rápida y sencilla.

- **Windows**

  1. Abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
  2. Dentro de la terminal ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `pip install numpy`
     b) `easy_install numpy`
  3. Cierre la terminal de línea de comandos de Windows.
  4. Para Comprobar la instalación del programa abra la interfaz gráfica de Python, escribiendo *IDLE* en el recuadro de *Buscar en la web y en Windows* y escriba lo siguiente después del prompt >>> :
     `import numpy`.
  5. En caso de una instalación exitosa no marcara ningún error.

- **Mac OS X/ Unix y Linux**

[6] http://www.numpy.org/

1. Abra una terminal de Mac o una terminal (shell) de Unix/Linux y ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
   a) `sudo pip install numpy`
   b) `sudo easy_install numpy`
2. Para comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:

```
>>> import numpy
```

3. En caso de una instalación exitosa no marcara ningún error.

Es importante destacar que puede que se necesite alguna dependencia extra de Numpy, por lo que si es necesario puede ejecutar alguna de las siguientes instrucciónes:

- `pip install nombrePrograma`
- `easy_install nombrePrograma`

A.5 Instalación de Scipy

A continuación se presenta los pasos para poder instalar el paquete **Scipy**[7], el cual es una biblioteca libre con distintos algoritmos y funciones matemáticas de alto nivel.

- **Windows**
  1. Abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
  2. Dentro de la terminal ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `pip install scipy`
     b) `easy_install scipy`
  3. En caso de que pip o easy_install no puedan instalar el paquete, puede bajar el archivo .whl de la versión de Scipy para Python 3.x de la siguiente página http://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy. Después en la

[7] https://www.scipy.org/index.html

terminal de línea de comandos diríjase a la carpeta donde está el archivo .whl y ejecute lo siguiente:
`pip install <nombre archivo>.whl`.

4. Cierre la terminal de línea de comandos de Windows.
5. Para Comprobar la instalación del programa abra la interfaz gráfica de Python, escribiendo *IDLE* en el recuadro de *Buscar en la web y en Windows* y escriba lo siguiente después del prompt >>> :
`import scipy`.
6. En caso de una instalación exitosa no marcara ningún error.

- **Mac OS X/ Unix y Linux**
  1. Abra una terminal de Mac o una terminal (shell) de Unix/Linux y ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con otra instrucción):
     a) `sudo pip install scipy`
     b) `sudo easy_install scipy`
     c) `sudo apt-get install python-scipy`
  2. Para comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:

```
>>> import scipy
```

  3. En caso de una instalación exitosa no marcara ningún error.

Es importante destacar que puede que se necesite alguna dependencia extra de Scipy, por lo que si es necesario puede ejecutar alguna de las siguientes instrucciónes (de la misma manera que con numpy):

- `pip install nombrePrograma`
- `easy_install nombrePrograma`

A.6 Instalación de Matplotlib

A continuación se presentan los pasos para poder instalar el paquete `Matplotlib`[8], el cual es una biblioteca libre para la generación de gráficos a partir de datos contenidos en listas o arrays en Python[9].

[8] http://matplotlib.org
[9] Antes de poder instalar matplotlib es necesario tener instalado numpy.

- **Windows**

  1. Abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
  2. Dentro de la terminal ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `pip install matplotlib`
     b) `easy_install matplotlib`
  3. Cierre la terminal de línea de comandos de Windows.
  4. Para Comprobar la instalación del programa abra la interfaz gráfica de Python, escribiendo *IDLE* en el recuadro de *Buscar en la web y en Windows* y escriba lo siguiente después del prompt >>> :
     `import matplotlib`
  5. En caso de una instalación exitosa no marcará ningún error.

- **Mac OS X/ Unix y Linux**

  1. Abra una terminal de Mac o una terminal (shell) de Unix/Linux y ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `sudo pip install matplotlib`
     b) `sudo easy_install matplotlib`
  2. Para Comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:

     ```
     >>> import matplotlib
     ```

  3. En caso de una instalación exitosa no marcará ningún error.

Es importante destacar que puede que se necesite alguna dependencia extra de Matplotlib, por lo que si es necesario puede ejecutar alguna de las siguientes instrucciónes (de la misma manera que con Numpy y Scipy):

- `pip install nombrePrograma`
- `easy_install nombrePrograma`

A.7 Instalación de Tweepy

A continuación se presentan los pasos para poder instalar el paquete `Tweepy`[10], el cual es una biblioteca libre para la extracción de tweets (mensajes de texto de no más de 140 caracteres) en tiempo real usando una interfaz de tipo cliente servidor con Twitter Streaming API[11].

- **Windows**
  1. Abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
  2. Dentro de la terminal ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `pip install tweepy`
     b) `easy_install tweepy`
  3. Cierre la terminal de línea de comandos de Windows.
  4. Para Comprobar la instalación del programa abra la interfaz gráfica de Python, escribiendo *IDLE* en el recuadro de *Buscar en la web y en Windows* y escriba lo siguiente después del prompt `>>>` :
     `import tweepy`
  5. En caso de una instalación exitosa no marcará ningún error.
- **Mac OS X/ Unix y Linux**
  1. Abra una terminal de Mac o una terminal (shell) de Unix/Linux y ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `sudo pip install tweepy`
     b) `sudo easy_install tweepy`
  2. Para Comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:

     ```
     >>> import tweepy
     ```

  3. En caso de una instalación exitosa no marcará ningún error.

Es importante destacar que puede que se necesite alguna dependencia extra de Tweepy, por lo que si es necesario puede ejecutar alguna de las siguientes instrucciónes (de la misma manera que con Matplotlib y Numpy):

[10] http://www.tweepy.org/
[11] https://dev.twitter.com/streaming/overview

- `pip install nombrePrograma`
- `easy_install nombrePrograma`

A.8 Instalación de Pymongo

A continuación se presentan los pasos para poder instalar el paquete `Pymongo`[12], el cual es una biblioteca libre para la conexión de tipo cliente servidor entre Python y una base de datos de tipo noSQL llamada MongoDB.

- **Windows**
  1. Abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
  2. Dentro de la terminal ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `pip install pymongo`
     b) `easy_install pymongo`
  3. Cierre la terminal de línea de comandos de Windows.
  4. Para Comprobar la instalación del programa abra la interfaz gráfica de Python, escribiendo *IDLE* en el recuadro de *Buscar en la web y en Windows* y escriba lo siguiente después del prompt `>>>` :
     `from pymongo import MongoClient`
  5. En caso de una instalación exitosa no marcará ningún error.
- **Mac OS X/ Unix y Linux**
  1. Abra una terminal de Mac o una terminal (shell) de Unix/Linux y ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `sudo pip install pymongo`
     b) `sudo easy_install pymongo`
  2. Para Comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:

```
>>> from pymongo import MongoClient
```

[12] https://docs.mongodb.com/getting-started/python/client/

3. En caso de una instalación exitosa no marcará ningún error.

Es importante destacar que puede que se necesite alguna dependencia extra de Pymongo, por lo que si es necesario puede ejecutar alguna de las siguientes instrucciónes (de la misma manera que con los demás paquetes):

- `pip install nombrePrograma`
- `easy_install nombrePrograma`

A.9 Instalación de Geopy

A continuación se presentan los pasos para poder instalar el paquete `Geopy`[13], el cual es una biblioteca libre diseñada para obtener información geolocalizada de ciudades, países o cualquier dirección a lo largo de todo el mundo a través de una conexión tipo cliente servidor con otras populares herramientas de geolocalización como OpenStreetMap o Google Geocoding API entre otros.

- **Windows**
  1. Abra una terminal de línea de comandos de Windows, para ello, escriba *Símbolo del sistema* en el recuadro de *Buscar en la web y en Windows* (junto al icono de inicio de Windows).
  2. Dentro de la terminal ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):
     a) `pip install geopy`
     b) `easy_install geopy`
  3. Cierre la terminal de línea de comandos de Windows.
  4. Para Comprobar la instalación del programa abra la interfaz gráfica de Python, escribiendo *IDLE* en el recuadro de *Buscar en la web y en Windows* y escriba lo siguiente después del prompt >>> :
     `import geopy.geocoders`
  5. En caso de una instalación exitosa no marcará ningún error.
- **Mac OS X/ Unix y Linux**
  1. Abra una terminal de Mac o una terminal (shell) de Unix/Linux y ejecute alguna de las siguientes opciones (En caso de error con alguna de las instrucciones, pruebe con la otra instrucción):

[13] https://pypi.python.org/pypi/geopy

a) `sudo pip install geopy`
b) `sudo easy_install geopy`

2. Para Comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:

```
>>> import geopy.geocoders
```

3. En caso de una instalación exitosa no marcará ningún error.

Es importante destacar que puede que se necesite alguna dependencia extra de Geopy, por lo que si es necesario puede ejecutar alguna de las siguientes instrucciónes (de la misma manera que con los demás paquetes):

- `pip install nombrePrograma`
- `easy_install nombrePrograma`

A.10 Instalación de Matplotlib Basemap

A continuación se presentan los pasos para poder instalar el paquete `Matplotlib Basemap`[14], el cual es una biblioteca libre que hace uso de herramientas de Matplotlib para visualizar puntos geolocalizados en distintas representaciones cartográficas[15] (mapas).

- **Windows**

1. Diríjase a la pagina https://sourceforge.net/projects/matplotlib/files/matplotlib-toolkits/
2. Seleccione la opción más reciente de basemap (basemap 1.07 al momento de escribir este libro).
3. Descargue el archivo .exe de basemap considerando si su sistema operativo es de 32 (win32) o 64 (amd64) bits así como la versión de Python 3.x.
4. Abra el archivo .exe y siga todas las instrucciones por defecto.
5. Para Comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:
`import mpl_toolkits.basemap`

[14] http://matplotlib.org/basemap/
[15] Antes de poder instalar Matplotlib Basemap es necesario tener instalado Matplotlib.

6. En caso de una instalación exitosa no marcará ningún error.

- **Unix y Linux**

1. Abra una una terminal (shell) de Unix/Linux y ejecute la siguientes instrucción:

```
sudo apt-get install python-mpltoolkits.basemap
```

2. Para Comprobar la instalación del programa abra la interfaz gráfica de programación de Python (IDLE) y escriba lo siguiente:
`import mpl_toolkits.basemap`
3. En caso de una instalación exitosa no marcará ningún error.

Apéndice B
Creación de ejecutables en Python

B.1 Introducción

B.2 Instalación de `pyinstaller`

B.3 Creación de ejecutables

B.4 Ejemplo

B.1 Introducción

En este apéndice se provee la información necesaria para realizar archivos ejecutables a partir de programas realizados en Python. Aunque existen una gran variedad de programas que sirven para este propósito, el que presentamos aquí lo hemos seleccionado por su sencillez. El paquete que usamos se llama `pyinstaller` y nos sirve para trabajar en los sistemas operativos Windows y en Mac.

B.2 Instalación de `pyinstaller`

El módulo `pyinstaller` es gratuito y se puede instalar desde la ventana de comandos del sistema. Para esto usamos el administrador de dependencias `pip` como se explica en el Apéndice A. En una ventana de comandos en Windows y en una ventana de Terminal en Mac y en Linux, `pyinstaller` se instala con:

```
pip install pyinstaller
```

Una vez instalado, en la ventana de comandos podemos checar que versión tenemos con:

```
pyinstaller --version
```

Para esta versión obtenemos

```
3.2
```

B.3 Creación de ejecutables

Para realizar un ejecutable de un archivo de Python, simplemente escribimos:

```
pyinstaller --onefile miarchivo.py
```

donde `miarchivo.py` es el nombre del archivo. Con esto se crean dos ficheros lla-

mados `dist` y `build`. En el fichero `dist` se crea el archivo `miarchivo.exe`. Para correrlo simplemente hacemos doble pulsación sobre él y se ejecutará en una ventana de comandos.

Este procedimiento es válido tanto para Windows como para Mac.

B.4 Ejemplo

Consideremos el script siguiente:

```
% Ejemplo de ejecutable
% Este ejemplo suma dos datos de entrada a y b.

a = float(input("Dame el valor de a: \n"))
b = float(input("Dame el valor de b: \n''))
c = a + b
print( " El valor de ", a, " + ", b, " es igual a ", c)
x = input("Pulsa una tecla para continuar.")
```

Lo salvamos con el nombre `suma.py` y para crear el ejecutable, abrimos una ventana de comandos en Windows o Terminal en Mac, nos colocamos en el directorio donde hayamos salvado el archivo `suma.py` y escribimos:

```
> pyinstaller --onefile suma.py
```

Al terminar, se habrá creado en el mismo directorio un fichero `dist` y un fichero `build`. Dentro del fichero `dist` se encuentra el archivo `suma.exe`. Al presionar el icono dos veces, se abre una nueva ventana de comandos donde se ejecuta el programa como se muestra en la figura B.1.

Figura B.1 Corrida de `suma.exe`.

Índice analítico

∧, 38
*, 164
**, 54
+, 164
., 320
//, 25
:, 63, 100
;, 23
=, 33
==, 64
#, 27, 32, 54
%, 41
&, 38, 54
ábaco, 2
índice, 139, 152, 173, 174, 177, 179
' ', 54
" ", 54
''' ''', 54
3d, 371

añadir, 158
abrir, 296
Access Token, 391
Access Token Secret, 391
acumulador, 98
add, 349
Aiken, 3
Al Khuarismi, 21
alambre, 375
aleatorios, 131
alfanumérica, 22, 48
alfanumérico, 304
algoritmo, 18, 19, 32, 60
algoritmo principal, 32
ALU, 6
análisis de sentimientos, 390
and, 38, 54
anidado, 60
append, 156, 158, 194, 208, 215, 244, 272
arange, 365, 375, 379
ArcGIS, 385
archivo, 296
argumentos, 252
aritméticos, 35
arreglo, 170
arreglos, 170, 212
Asignación, 23
asignación, 33, 35, 39, 64, 175
aspect, 374, 379
atributo, 317, 344
autopct, 366, 379
Axes3D, 378, 379

búsqueda, 204
búsqueda binaria, 205
Babbage, 2
basemap, 388
bibliotecas, 34
bin, 284
binario, 7, 283
Bing Maps, 385
bit, 8
bool, 32, 42, 54
borrar, 158
bucles, 96
build, 423
burbuja, 200
byte, 13

C, 14
C++, 14, 316

código abierto, 14
código fuente, 14
código objeto, 14
cadena, 33, 42, 48, 138, 164
cadenas, 138
capitalize, 146, 147
caracter, 48, 138
Case, 86
Caso, 83
casos, 60, 83
centígrados, 47
center, 146, 147
cerrar, 296
Charles Babbage, 3
chr, 32, 49
ciclo infinito, 102
ciclo Para, 106
ciclos, 96, 149
ciclos anidados, 60
Clase, 317
clase, 317, 322
clase padre, 325, 329, 330
class, 33, 319, 349
clave, 53, 161
clear, 162
close, 296, 299, 301, 313
cmap, 379
cociente, 264
colecciones, 170
color, 365, 379
colorbar, 374, 379
columnas, 212, 220
comentario, 27, 32
comentarios, 39
comillas, 138
comma-separated-values, 308
comparación, 64
compilador, 14
comprensión, 181, 217, 236
concatenación, 49, 153, 155
concatenación de cadenas, 143
concatenar, 52, 164
condición, 60
condición anidada, 77
condiciones, 60, 148
conectivos lógicos, 65
conjuntos, 170
constructor, 320, 321, 325, 345
Consumer Key, 391
Consumer Secret, 391
contador, 99, 102, 106, 150, 177
continue, 123
contour, 379
Conversión, 47, 283
coolwarm, 373, 379
copy, 162, 163
coseno, 209
count, 146, 147, 156
CPU, 6
crear un archivo, 296
cstride, 373, 379
csv, 308, 312, 313
csv.reader, 308, 313
csv.writer, 308, 311, 313

datos, 23
datos alfanuméricos, 300
datos numéricos, 301
decimal, 7, 283
decremento, 112
def, 257, 292
delimitador, 164
demjson, 402
desviación estándar, 187
diagonal, 247
diagonal principal, 247, 248
diccionario, 161
diccionarios, 53, 152
dimensión, 213
dinámicos, 33
dist, 423
división entera, 25
divmod, 264
doble signo igual, 64
dos puntos, 63, 100, 107
double, 42

ejecutable, 14, 422

elemento, 173, 213
elif, 89
else, 89
else-if, 78
encapsulación, 316, 317
encapsulamiento, 323
encapsular, 317
endswith, 146, 147
ENIAC, 3
Enigma, 3
entera, 22
Entonces, 61
escribir en un archivo, 296
Excel, 308
expandtabs, 146
expresión lógica, 61, 96, 102
extend, 157, 158
extender, 158

factorial, 126, 266, 282
Fahrenheit, 47
False, 54
Falso, 22
Fibonacci, 194, 275
figsize, 366, 379
figure, 360, 379
filas, 220
file, 296
FIN, 25
find, 146
FinPara, 175, 176
FinSi, 61, 176
float, 32, 42, 54
for, 106, 181, 183, 214
formato, 40, 41
fromkeys, 162, 163
función, 252
función lambda, 267
funciones, 250, 252, 255

gca, 371
geolocalización, 384
geopy, 384
get, 162, 322
getsizeof, 56
getter, 322, 328, 349
global, 271
Google Geocoding, 385
grid, 365, 379
Guido van Rossum, 30
Guido von Rossum, 14

handle, 296
has_key, 162
hasta, 106
herencia, 316–318, 324, 325
herencia simple, 326
hexadecimal, 11
hist, 379
histograma, 367
hot, 378

IBM, 3
identidad, 231
identificador, 32, 296
IDLE, 15, 30, 43
if, 63, 86, 89
if - else, 73
if-else, 78
igual, 33
import, 34, 54, 182, 238, 273
in, 148, 214
incremento, 33, 106
index, 157
indexación, 215
inicialización, 194
inicializar, 23, 98, 174
INICIO, 25
init, 320, 349
inmutabilidad, 144, 154
inmutable, 145, 159
input, 32, 54, 192
insert, 157, 158
insertar, 158
instancia, 317
instrucción, 14
instrucciones, 32
int, 32, 42, 54

intérprete, 30
intérpretes, 14
Integrated Development Environment, 15
interés compuesto, 47
interés simple, 47
invertir, 158
isalnum, 146, 147
isalpha, 146
isdigit, 146
islower, 146
isnum, 148
isnumeric, 146
isspace, 146
istitle, 146
isupper, 146
items, 162, 163
iterador, 218

Java, 14
John D. Hunter, 352
join, 155, 156, 164

key, 53, 161
keys, 162
keywords, 28

lógica, 22
lógicos, 35, 36
label, 358, 379
latitud, 384
lazos, 96
lectura, 303
leer de un archivo, 296
legend, 358, 363, 379
Leibnitz, 2
len, 49, 52, 54, 141, 153, 156, 157, 160, 164, 310
lenguaje de máquina, 13
lenguaje de programación, 29
linewidth, 365, 374, 379
linspace, 356, 379
list, 54, 160, 287
lista, 52, 214
listas, 152, 180
llamado, 259
llamado por referencia, 268
llamado por valor, 268
llave, 53
loc, 359, 363, 380
localidad de memoria, 12
longitud, 13, 49, 157, 384

más significativo, 8
máximo, 157
máximo común divisor, 283
método, 317
mínimo, 158
módulo, 39, 184, 264
MAC, 14
magnético, 376
mantisa, 22
mapa, 388
Mark I, 3
math, 34, 48, 54
MATLAB, 14
matplotlib, 256, 352, 354, 380
matrices, 212
matriz, 170, 212
matriz cuadrada, 213
Matsumoto, 131
max, 148, 156, 157, 160, 161
MeaningCloud, 398
memoria, 12
menos significativo, 8
Mersenne Twister, 131
meshgrid, 372, 380
Mientras, 96, 175, 179
min, 148, 156, 158, 160, 161
modo, 297
Mongo, 395
mongod, 402
Mostrar, 25, 40
mpl_toolkits, 380
mplot3d, 378
multidimensional, 212
multiplicación, 49, 50, 153, 155, 225
multiplicación de cadenas, 144
multiplicación de matrices, 227
multiplicar, 52

mutabilidad, 154
mutable, 154
MyStreamListener, 392

números aleatorios, 182
Newton, 60
Nishimura, 131
nombre, 32
Nominatim, 385
None, 54, 180, 183, 193, 194, 216
norma Euclidiana, 208
norma infinita, 209
norma p, 209
normal, 367, 380
not, 38, 54
numérico, 304
numpy, 256, 380

object code, 14
objeto, 317
objetos, 316
octal, 11
offset, 378
open, 296, 299, 301, 313
open source, 14
OpenStreetMap, 385
operador punto, 320, 349
operadores aritméticos, 35
operadores de asignación, 39
operadores lógicos, 36
operadores relacionales, 36
or, 54
or exclusivo, 38
orden, 213
ordenamiento, 196
ordenamiento de burbuja, 200
ordenamiento por selección, 196
ordenar, 159

palíndromo, 190, 286
palabra, 13
palabras clave, 22
palabras reservadas, 22
parámetros, 252
paréntesis, 61, 107, 109
Para, 96, 106, 175, 176, 179, 188, 220
paradigma, 316
pares, 53
Pascal, 2
paso por referencia, 268
paso por valor, 268
perímetro, 280
pi, 363
pie, 366, 380
PlaceFinder, 385
plot, 353, 380
plot(x, y), 353
plot_surface, 378, 380
polar, 365, 380
polaridad, 398
polimorfismo, 316–318, 331
POO, 316
pop, 157, 158
porción de cadena, 140
pow, 48, 54
pprint, 309, 313
primo, 102, 254
principal, 248
print, 31, 32, 40, 41, 54
procedimientos, 250, 261
programación orientada a objetos, 316
projection, 371, 380
promedio, 178, 186
proyección, 378
pseudoaleatorio, 131
pseudocódigo, 21
punto y coma, 23
pyinstaller, 422
pylab, 377
pymongo, 395
pyplot, 353, 380
Python, 14, 29

quiver, 376, 380
quiverkey, 376

randint, 131, 183, 244, 274
random, 131, 182, 238, 244, 273
range, 181, 183, 214

real, 22
Recibir, 25, 40, 98
recursividad, 266, 267
redes sociales, 384, 390
Regresa, 253
relacionales, 35, 36
remove, 157, 158
renglones, 212
replace, 148
residuo, 123, 264
resta de matrices, 221
return, 257, 292
reverse, 157, 158
reversed, 287
ROM, 12
rstride, 373, 380

salto de línea, 298
salto de renglón, 40, 41
sangría, 63, 100, 107, 253
scripts, 30
secuencia, 32
Según, 83
self, 320, 349
sentimientos, 390
set, 322
setter, 322, 328, 349
show, 353, 363, 366, 380
shrink, 374
Si, 61, 176
Si_no, 73
simétrica, 237
Simula 67, 316
sin, 357
sintaxis, 14
size, 367, 380
slice, 154
sobescritura, 324
sobrecarga, 324, 325
sobreescritura, 330
sort, 157, 159
source code, 14
split, 141, 164
sqrt, 187
stem, 368, 380
str, 32, 33, 42, 49, 54, 162, 163
string, 33, 42
strings, 138
sub, 349
subalgoritmos, 250
subcadena, 142, 156
subgráficas, 362
sublista, 154
subplot, 362, 380
suma, 49, 153, 155, 174
suma de cadenas, 143
suma de matrices, 221
sumar, 52
super, 328, 349
superficie, 372
swapcase, 146
Switch, 86
Switch-Case, 86

tabulador, 43
tamaño, 179, 180, 193
Taylor, 281
ticks, 357
timeout, 385
tipo, 22, 32
tipo de variable, 22
tipos dinámicos, 23, 33
title, 146, 380
transpuesta, 233
traza, 248
triángulo, 280
triangular, 247
triangular inferior, 248
triangular superior, 247
True, 54, 146
tupla, 159, 260
tuplas, 152
tuple, 156, 160
Turing, 3
tweepy, 391
tweets, 390
Twitter, 390
type, 54, 162, 163

Unidad de control, 6
Unidad de entrada, 6
Unidad de memoria, 6
Unidad de salida, 6
Unidad Lógica Aritmética, 6
Unix, 14
update, 162
upper, 146
upper left, 359, 363, 380
upper right, 363, 380
USB, 12

valor, 32, 53, 161
valor de verdad, 61
valor esperado, 186
valor medio, 186
values, 162, 163
variable, 22
variable global, 269
variables alfanuméricas, 48
variables booleanas, 36
variables locales, 269
varianza, 186
vector, 170, 171
vectores, 180
Verdadero, 22
visualización, 352
von Neumann, 3

while, 100, 181
Windows, 14
wireframe, 375, 380
with, 300, 313
write, 297, 301, 313
writelines, 299, 301, 313
writerow, 313

xlim, 355, 380
xticks, 357, 380

ylim, 355, 363, 380

zdir, 380
zfill, 148
zlim, 380