Algoritmos y Programación I

Aprendiendo a programar usando Python como herramienta

2da edición

Última revisión: 23 de agosto de 2018

Contenidos

Prólogo **8**

1 Conceptos básicos **10**
1.1 Computadoras y programas . . . 10
1.2 El mito de la máquina todopoderosa . . . 11
1.3 Cómo darle instrucciones a la máquina usando Python . . . 12
1.4 Variables . . . 14
1.5 Funciones . . . 15
1.6 Una instrucción un poco más compleja: el ciclo definido . . . 17
1.7 Construir programas y módulos . . . 19
1.8 Interacción con el usuario . . . 20
1.9 Estado y computación . . . 22
1.9.1 Depuración de programas . . . 23
1.10 Ejercicios . . . 24

2 Programas sencillos **25**
2.1 Construcción de programas . . . 25
2.2 Realizando un programa sencillo . . . 26
2.3 Piezas de un programa Python . . . 28
2.3.1 Nombres . . . 28
2.3.2 Expresiones . . . 29
2.3.3 No sólo de números viven los programas . . . 30
2.3.4 Instrucciones . . . 31
2.3.5 Ciclos definidos . . . 31
2.4 Una guía para el diseño . . . 32
2.5 Calidad de software . . . 32
2.6 Ejercicios . . . 34

3 Funciones **35**
3.1 Documentación de funciones . . . 35
3.2 Imprimir versus devolver . . . 36
3.3 Cómo usar una función en un programa . . . 37
3.4 Alcance de las variables . . . 39
3.5 Un ejemplo completo . . . 40
3.6 Devolver múltiples resultados . . . 42
3.7 Módulos . . . 44
3.7.1 Módulos estándar . . . 45
3.8 Resumen . . . 45

3.9 Ejercicios . . . 48

4 Decisiones **49**
4.1 Expresiones booleanas . . . 49
4.1.1 Expresiones de comparación . . . 50
4.1.2 Operadores lógicos . . . 50
4.2 Comparaciones simples . . . 51
4.3 Múltiples decisiones consecutivas . . . 53
4.4 Ejercicios . . . 55
4.5 Resumen . . . 56
4.6 Ejercicios . . . 58

5 Más sobre ciclos **59**
5.1 Ciclos indefinidos . . . 60
5.2 Ciclo interactivo . . . 60
5.3 Ciclo con centinela . . . 61
5.4 Cómo cortar un ciclo . . . 63
5.4.1 Otras formas de controlar el flujo de un ciclo . . . 64
5.5 Ejercicios . . . 66
5.6 Resumen . . . 67
5.7 Ejercicios . . . 68

6 Cadenas de caracteres **70**
6.1 Operaciones con cadenas . . . 70
6.1.1 Obtener la longitud de una cadena . . . 71
6.1.2 Recorrer una cadena . . . 71
6.1.3 Preguntar si una cadena contiene una subcadena . . . 71
6.1.4 Acceder a una posición de la cadena . . . 72
6.2 Segmentos de cadenas . . . 73
6.3 Las cadenas son inmutables . . . 73
6.4 Procesamiento sencillo de cadenas . . . 74
6.5 Darle formato a las cadenas . . . 76
6.6 Nuestro primer juego . . . 78
6.7 Resumen . . . 85
6.8 Ejercicios . . . 87

7 Tuplas y listas **89**
7.1 Tuplas . . . 89
7.1.1 Elementos y segmentos de tuplas . . . 89
7.1.2 Las tuplas son inmutables . . . 90
7.1.3 Longitud de tuplas . . . 90
7.1.4 Empaquetado y desempaquetado de tuplas . . . 91
7.1.5 Ejercicios con tuplas . . . 92
7.2 Listas . . . 93
7.2.1 Longitud de la lista. Elementos y segmentos de listas . . . 93
7.2.2 Cómo mutar listas . . . 94
7.2.3 Cómo buscar dentro de las listas . . . 95
7.3 Ordenar listas . . . 100
7.4 Listas y cadenas . . . 100

7.4.1 Ejercicios con listas y cadenas . . . 101
7.5 Resumen . . . 102
7.6 Ejercicios . . . 105
Apéndice 7.A Funciones de orden superior . . . 107

8 Más sobre listas **111**
8.1 Búsqueda lineal . . . 111
8.2 Búsqueda sobre listas ordenadas . . . 113
8.3 Resumen . . . 117
8.4 Ejercicios . . . 118
Apéndice 8.A Filtros, transformaciones y acumulaciones . . . 119
8.A.1 Listas por comprensión . . . 120

9 Diccionarios **123**
9.1 Qué es un diccionario . . . 123
9.2 Utilizando diccionarios en Python . . . 124
9.3 Algunos usos de diccionarios . . . 125
9.4 Resumen . . . 126
9.5 Ejercicios . . . 128
Apéndice 9.A Conjuntos . . . 129

10 Contratos y Mutabilidad **131**
10.1 Pre y Postcondiciones . . . 131
10.1.1 Precondiciones . . . 131
10.1.2 Postcondiciones . . . 131
10.1.3 Aseveraciones . . . 132
10.1.4 Ejemplos . . . 132
10.2 Invariantes de ciclo . . . 133
10.2.1 Comprobación de invariantes desde el código . . . 134
10.3 Mutabilidad e Inmutabilidad . . . 134
10.3.1 Parámetros mutables e inmutables . . . 135
10.4 Resumen . . . 136
Apéndice 10.A Acertijo MU . . . 138

11 Manejo de archivos **139**
11.1 Cerrar un archivo . . . 140
11.2 Ejemplo de procesamiento de archivos . . . 140
11.3 Modo de apertura de los archivos . . . 141
11.4 Escribir en un archivo . . . 141
11.5 Agregar información a un archivo . . . 142
11.6 Manipular un archivo en forma binaria . . . 143
11.7 Persistencia de datos . . . 144
11.7.1 Persistencia en archivos CSV . . . 146
11.7.2 Persistencia en archivos binarios . . . 147
11.8 Directorios . . . 148
11.9 Resumen . . . 148
11.10Ejercicios . . . 151
Apéndice 11.A Agenda con archivos CSV . . . 152

12 Manejo de errores y excepciones **155**
12.1 Errores . . . 155
12.2 Excepciones . . . 155
12.2.1 Manejo de excepciones . . . 156
12.2.2 Manejo de excepciones con archivos . . . 158
12.2.3 Procesamiento y propagación de excepciones . . . 159
12.2.4 Acceso a información de contexto . . . 160
12.3 Validaciones . . . 160
12.3.1 Comprobaciones por contenido . . . 161
12.3.2 Entrada del usuario . . . 161
12.3.3 Comprobaciones por tipo . . . 162
12.3.4 Comprobaciones por características . . . 163
12.4 Resumen . . . 164
Apéndice 12.A Ejemplo: programa con manejo de excepciones . . . 166

13 Procesamiento de archivos **167**
13.1 Corte de control . . . 167
13.2 Apareo . . . 169
13.3 Resumen . . . 169

14 Objetos **171**
14.1 Tipos . . . 171
14.2 Qué es un objeto . . . 172
14.3 Definiendo nuevos tipos . . . 173
14.3.1 Nuestra primera clase: Punto . . . 173
14.3.2 Agregando validaciones al constructor . . . 174
14.3.3 Agregando operaciones . . . 175
14.4 Métodos especiales . . . 176
14.4.1 Conversión a cadena de texto . . . 177
14.4.2 Métodos para operar matemáticamente . . . 178
14.5 Composición de objetos . . . 179
14.6 Mutabilidad . . . 180
14.7 Creando clases más complejas . . . 181
14.7.1 Métodos para comparar objetos . . . 183
14.7.2 Ordenar de menor a mayor listas de hoteles . . . 183
14.7.3 Otras formas de comparación . . . 184
14.7.4 Comparación sólo por igualdad o desigualdad . . . 185
14.8 Resumen . . . 185
14.9 Ejercicios . . . 188

15 Polimorfismo, Herencia y Delegación **190**
15.1 Polimorfismo . . . 190
15.1.1 Interfaz . . . 190
15.1.2 Redefinición de métodos . . . 191
15.2 Herencia . . . 193
15.3 Delegación . . . 195
15.4 Resumen . . . 197
15.5 Ejercicios . . . 198

16 Listas enlazadas **199**
16.1 Una clase sencilla de *vagones* . . . 199
16.1.1 Caminos . . . 201
16.1.2 Referenciando el principio de la lista . . . 201
16.2 Tipos abstractos de datos . . . 201
16.3 La clase `ListaEnlazada` . . . 202
16.3.1 Construcción de la lista . . . 203
16.3.2 Eliminar un elemento de una posición . . . 204
16.3.3 Eliminar un elemento por su valor . . . 206
16.3.4 Insertar nodos . . . 207
16.4 Invariantes de objetos . . . 208
16.5 Otras listas enlazadas . . . 209
16.6 Iteradores . . . 210
16.7 Resumen . . . 213
16.8 Ejercicios . . . 214

17 Pilas y colas **215**
17.1 Pilas . . . 215
17.1.1 Pilas representadas por listas . . . 215
17.1.2 Uso de pila: calculadora científica . . . 217
17.1.3 ¿Cuánto cuestan los métodos? . . . 220
17.2 Colas . . . 221
17.2.1 Colas implementadas sobre listas . . . 221
17.2.2 Colas y listas enlazadas . . . 222
17.3 Resumen . . . 224
17.4 Ejercicios . . . 225
Apéndice 17.A Implementación de la pila y cola . . . 227

18 Modelo de ejecución de funciones y recursión **230**
18.1 La pila de ejecución de las funciones . . . 230
18.2 Pasaje de parámetros . . . 232
18.3 Devolución de resultados . . . 234
18.4 La recursión y cómo puede ser que funcione . . . 235
18.5 Una función recursiva matemática . . . 236
18.6 Algoritmos recursivos y algoritmos iterativos . . . 238
18.7 Un ejemplo de recursión elegante . . . 239
18.8 Un ejemplo de recursión poco eficiente . . . 241
18.9 Diseño de algoritmos recursivos . . . 242
18.10 Un primer diseño recursivo . . . 243
18.11 Pasaje de la información . . . 244
18.12 Modificación de la firma . . . 245
18.13 Limitaciones . . . 246
18.14 Resumen . . . 247
18.15 Ejercicios . . . 248

19 Ordenar listas **250**
19.1 Ordenamiento por selección 250
19.1.1 Invariante en el ordenamiento por selección 252
19.1.2 ¿Cuánto cuesta ordenar por selección? 252
19.2 Ordenamiento por inserción 253
19.2.1 Invariante del ordenamiento por inserción 255
19.2.2 ¿Cuánto cuesta ordenar por inserción? 255
19.2.3 Inserción en una lista ordenada 255
19.3 Resumen 255
19.4 Ejercicios 257

20 Algunos ordenamientos recursivos **258**
20.1 Ordenamiento por mezcla, o *Merge sort* 258
20.1.1 ¿Cuánto cuesta el *Merge sort*? 259
20.2 Ordenamiento rápido o *Quick sort* 261
20.2.1 ¿Cuánto cuesta el *Quick sort*? 262
20.2.2 Una versión mejorada de *Quick sort* 263
20.3 Resumen 265
20.4 Ejercicios 266

Licencia y Copyright **267**

Prólogo

Durante mucho tiempo nos preguntamos cómo diseñar un curso de Algoritmos y Programación I (primera materia de programación de las carreras de Ingeniería en Informática y Licenciatura en Análisis de Sistemas de la Facultad de Ingeniería de la UBA) que al mismo tiempo fuera atractivo para los futuros profesionales de la informática, les permitiera aprender a resolver problemas, y atacara muy directamente el problema de la deserción temprana de estudiantes.

Muchos de los estudiantes que ingresan a primer año de las carreras de informática en nuestro país lo hacen sin saber programar pese a ser nativos digitales, para los cuales las computadoras y muchos programas forman parte de su vida cotidiana. El primer curso de programación plantea entonces varios desafíos: enseñar una metodología para la resolución de problemas, un lenguaje formal para escribir los programas, y al mismo tiempo hacer que los estudiantes no se sientan abrumados, tengan éxito en este primer esfuerzo y se sientan atraídos por la posibilidad de escribir sus propios programas.

En este sentido, la elección del lenguaje de programación para dicho curso no es un tema menor: el problema radica en elegir un lenguaje que al mismo tiempo sea suficientemente expresivo, tenga una semántica clara y cuyas complicaciones sintácticas sean mínimas.

Hacía tiempo que buscábamos un lenguaje con todas estas características cuando, durante el debate posterior a un panel sobre programación en el nivel que tuvo lugar en la conferencia Frontiers in Engineering Education 2007 organizada por la IEEE, el Dr. John Impagliazzo, de Hofstra University, relató cómo sus cursos habían pasado de duras experiencias, con altas tasas de deserción, a una situación muy exitosa, por el solo hecho de haber cambiado el lenguaje de programación de ese curso de Java a Python. Después de ese estimulante encuentro con Impagliazzo nos pusimos manos a la obra para diseñar este curso. Fue una grata sorpresa enterarnos también de que el venerable curso de programación de MIT también había migrado a Python: todo hacía pensar que nuestra elección de lenguaje no era tan descabellada como se podía pensar.

Este libro pretende entonces ser una muy modesta contribución a la discusión sobre cómo enseñar a programar en primer año de una carrera de Informática a través de un lenguaje que tenga una suave curva de aprendizaje, de modo tal que este primer encuentro le resulte a los estudiantes placentero y exitoso, sin que los detalles del lenguaje los distraiga del verdadero objetivo del curso: la resolución de problemas mediante computadoras.

Agradecimientos

Queremos agradecer a la Facultad de Ingeniería (en particular a su Comisión de Publicaciones) y a Eudeba por la publicación de este libro. Pero también a todos los que apoyaron desde un primer momento la escritura y mejora continua de lo que fueron durante varios años las notas de nuestro curso de Algoritmos y Programación I: a la Comisión Curricular de la Licenciatura

en Análisis de Sistemas y al Departamento de Computación (y a su director, Gustavo López) por apoyar la iniciativa de dar este curso como piloto, a quienes leyeron y discutieron los manuscritos desde sus primeras versiones y colaboraron con el curso (Melisa Halsband, Alberto Bertogli, Sebastián Santisi, Pablo Antonio, Pablo Najt, Leandro Ferrigno, Martín Albarracín, Gastón Kleiman, Matías Gavinowich), a quienes fueron primero alumnos y luego colaboradores de esta experiencia educativa (Agustín Santiago, Alejandro Levinas, Ana Czarnitzki, Ariel Vergara, Bruno Merlo Schurmann, Damián Schenkelman, Daniela Riesgo, Daniela Soto, Débora Martín, Diego Alfonso, Emiliano Sorbello, Ezequiel Genender, Fabricio Pontes Harsich, Fabrizio Graffe, Federico Barrios, Federico Esteban, Federico López, Florencia Álvarez Etcheverry, Gastón Goncalves, Gastón Martínez, Ignacio Garay, Ignacio Sueiro, Javier Choque, Jennifer Woites, Juan Costamagna, Lucas Perea, Manuel Battan, Manuel Soldini, Manuel Porto, Martín Buchwald, Martín Coll, Milena Farotto, Nicolás Poncet, Pablo Musumeci, Robinson Fang), a los amigos que, como Pablo Jacovkis, Elena García y Alejandro Tiraboschi se ofrecieron a revisar manuscritos y a hacer sugerencias, y a todos los alumnos de nuestro curso de Algoritmos y Programación I de la FIUBA que, desde 2009, han usado este texto y nos han ayudado a mejorarlo permanentemente. De todos modos, por supuesto, los errores son nuestra responsabilidad.

Buenos Aires, diciembre de 2012.

Unidad 1

Algunos conceptos básicos

En esta unidad hablaremos de lo que es un programa de computadora e introduciremos unos cuantos conceptos referidos a la programación y a la ejecución de programas. Utilizaremos en todo momento el lenguaje de programación Python para ilustrar esos conceptos.

1.1 Computadoras y programas

En la actualidad, la mayoría de nosotros utilizamos computadoras permanentemente: para mandar correos electrónicos, navegar por Internet, chatear, jugar, escribir textos.

Las computadoras se usan para actividades tan disímiles como predecir las condiciones meteorológicas de la próxima semana, guardar historias clínicas, diseñar aviones, llevar la contabilidad de las empresas o controlar una fábrica. Y lo interesante aquí (y lo que hace apasionante a esta carrera) es que el mismo aparato sirve para realizar todas estas actividades: uno no cambia de computadora cuando se cansa de chatear y quiere jugar al solitario.

Muchos definen una computadora moderna como "una máquina que almacena y manipula información bajo el control de un programa que puede cambiar". Aparecen acá dos conceptos que son claves: por un lado se habla de una *máquina* que almacena información, y por el otro lado, esta máquina está controlada por *un programa que puede cambiar*.

Una calculadora sencilla, de esas que sólo tienen 10 teclas para los dígitos, una tecla para cada una de las 4 operaciones, un signo igual, encendido y CLEAR, también es una máquina que almacena información y que está controlada por un programa. Pero lo que diferencia a esta calculadora de una computadora es que en la calculadora el programa no puede cambiar.

Un *programa de computadora* es un conjunto de *instrucciones* paso a paso que le indican a una computadora cómo realizar una tarea dada, y en cada momento uno puede elegir ejecutar un programa de acuerdo a la tarea que quiere realizar.

Las instrucciones se deben escribir en un lenguaje que nuestra computadora entienda. Los lenguajes de programación son lenguajes diseñados especialmente para dar órdenes a una computadora, de manera exacta y no ambigua. Sería muy agradable poder darle las órdenes a la computadora en castellano, pero el problema del castellano, y de las lenguas habladas en general, es su ambigüedad.

Si alguien nos dice "*Comprá el collar sin monedas*", no sabremos si nos pide que compremos el collar que no tiene monedas, o que compremos un collar y que no usemos monedas para la compra. Habrá que preguntarle a quien nos da la orden cuál es la interpretación correcta. Pero tales dudas no pueden aparecer cuando se le dan órdenes a una computadora.

Este curso va a tratar precisamente de cómo se escriben programas para hacer que una

computadora realice una determinada tarea. Vamos a usar un lenguaje específico, Python, porque es sencillo y elegante, pero éste no será un curso de Python sino un curso de programación.

Sabías que…

Existe una gran cantidad de programas desarrollados en Python, desde herramientas para servidores, como **mailman**, hasta programas amigables para usuarios finales, como **emesene**, pasando por aplicaciones empresariales, **openerp**, **tryton**; herramientas de desarrollo, **meld**, **mercurial**, **bazaar**, **trac**; plataformas web, **django**, **turbogears**, **zope**; clientes de bittorrent, **bittorrent**, **bittornado**, **deluge**; montones de juegos de todo tipo, y muchísimas aplicaciones más.

Estos programas mencionados son software libre, por lo que se puede obtener y estudiar el código con el que están hechos.

1.2 El mito de la máquina todopoderosa

Muchas veces la gente se imagina que con la computadora se puede hacer cualquier cosa; o que si bien hubo tareas que no eran posibles de realizar hace 50 años, sí lo serán cuando las computadoras crezcan en poder (memoria, velocidad), y se vuelvan máquinas todopoderosas.

Sin embargo eso no es así: existen algunos problemas, llamados *no computables* que nunca podrán ser resueltos por una computadora digital, por más poderosa que ésta sea. La computabilidad es la rama de la computación que se ocupa de estudiar qué tareas son computables y qué tareas no lo son.

De la mano del mito anterior, viene el mito del lenguaje todopoderoso: hay problemas que son no computables porque en realidad se utiliza algún lenguaje que no es el apropiado.

En realidad todas las computadoras pueden resolver los mismos problemas, y eso es independiente del lenguaje de programación que se use. Las soluciones a los problemas computables se pueden escribir en cualquier lenguaje de programación. Eso no significa que no haya lenguajes más adecuados que otros para la resolución de determinados problemas, pero la adecuación está relacionada con temas tales como la elegancia, la velocidad, la facilidad para describir un problema de manera simple, etc., nunca con la capacidad de resolución.

Los problemas no computables no son los únicos escollos que se le presentan a la computación. Hay otros problemas que si bien son computables demandan para su resolución un esfuerzo enorme en tiempo y en memoria. Estos problemas se llaman *intratables*. El análisis de algoritmos se ocupa de separar los problemas tratables de los intratables, encontrar la solución más barata para resolver un problema dado, y en el caso de los intratables, resolverlos de manera aproximada: no encontramos la verdadera solución porque no nos alcanzan los recursos para eso, pero encontramos una solución bastante buena y que nos insume muchos menos recursos (el orden de las respuestas de Google a una búsqueda es un buen ejemplo de una solución aproximada pero no necesariamente óptima).

En este curso trabajaremos con problemas no sólo computables sino también tratables. Y aprenderemos a medir los recursos que nos demanda una solución, y empezaremos a buscar la solución menos demandante en cada caso particular.

Algunos ejemplos de los problemas que encararemos y de sus soluciones:

Problema 1.1. Dado un número N se quiere calcular N^{33}.

Una solución posible, por supuesto, es hacer el producto $N \cdot N \cdots N$, que involucra 32 multiplicaciones.

Otra solución, mucho más eficiente es:

- Calcular $N \cdot N$.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^4.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^8.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^{16}.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^{32}.
- Al resultado anterior mutiplicarlo por N con lo cual conseguimos el resultado deseado con sólo 6 multiplicaciones.

Cada una de estas dos soluciones representa un *algoritmo*, es decir un método de cálculo, diferente. Para un mismo problema puede haber algoritmos diferentes que lo resuelven, cada uno con un costo distinto en términos de recursos computacionales involucrados.

Sabías que…

La palabra *algoritmo* no es una variación de *logaritmo*, sino que proviene de *algorismo*. En la antigüedad, los *algoristas* eran los que calculaban usando la numeración arábiga y mientras que los *abacistas* eran los que calculaban usando ábacos. Con el tiempo el *algorismo* se deformó en *algoritmo*, influenciado por el término *aritmética*.

A su vez, el uso de la palabra *algorismo* proviene del nombre de un matemático persa famoso, en su época y para los estudiosos de esa época, Abu Abdallah Muhammad ibn Mûsâ al-Jwârizmî, que literalmente significa: "Padre de Ja'far Mohammed, hijo de Moises, nativo de Jiva". Al-Juarismi, como se lo llama usualmente, escribió en el año 825 el libro "Al-Kitâb al-mukhtasar fî hîsâb al-gabr wa'l-muqâbala" (Compendio del cálculo por el método de completado y balanceado), del cual surgió también la palabra "álgebra".

Hasta hace no mucho tiempo se utilizaba el término algoritmo para referirse únicamente a formas de realizar ciertos cálculos, pero con el surgimiento de la computación, el término algoritmo pasó a abarcar cualquier método para obtener un resultado.

Problema 1.2. Tenemos que permitir la actualización y consulta de una guía telefónica.

Para este problema no hay una solución única: hay muchas y cada una está relacionada con un contexto de uso. ¿De qué guía estamos hablando: la guía de una pequeña oficina, un pequeño pueblo, una gran ciudad, la guía de la Argentina? Y en cada caso ¿de qué tipo de consulta estamos hablando: hay que imprimir un listado una vez por mes con la guía completa, se trata de una consulta en línea, etc.? Para cada contexto hay una solución diferente, con los datos guardados en una *estructura de datos* apropiada, y con diferentes algoritmos para la actualización y la consulta.

1.3 Cómo darle instrucciones a la máquina usando Python

El lenguaje Python nos provee de un *intérprete*, es decir un programa que interpreta las órdenes que le damos a medida que las escribimos. La forma más típica de invocar al intérprete es ejecutar el comando `python3` en la terminal.

Sabías que…

Python fue creado a finales de los años 80 por un programador holandés llamado Guido van Rossum, quien sigue siendo aún hoy el líder del desarrollo del lenguaje.

La versión 2.0, lanzada en 2000, fue un paso muy importante para el lenguaje ya que era mucho más madura, incluyendo un *recolector de basura*. La versión 2.2, lanzada en diciembre de 2001, fue también un hito importante ya que mejoró la orientación a objetos. La última versión de esta línea es la 2.7 que fue lanzada en noviembre de 2010 y aún está vigente.

En diciembre de 2008 se lanzó la rama 3.0, cuya versión actual es la 3.6, de diciembre de 2016, y es la que utilizamos en este libro. Python 3 fue diseñado para corregir algunos defectos de diseño en el lenguaje, y muchos de los cambios introducidos son incompatibles con las versiones anteriores. Por esta razón, las ramas 2.x y 3.x coexisten con distintos grados de adopción.

Atención

De forma tal de aprovechar al máximo este libro, recomendamos instalar Python 3 en una computadora, y acompañar la lectura probando todos los ejemplos de código y haciendo los ejercicios.

En `https://www.python.org/downloads/` se encuentran los enlaces para descargar Python, y en `http://docs.python.org.ar/tutorial/3/interpreter.html` hay más información acerca de cómo ejecutar el intérprete en cada sistema operativo.

```
$ python3
Python 3.6.0 (default, Dec 23 2016, 11:28:25)
[GCC 6.2.1 20160830] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Para orientarnos, el intérprete muestra los símbolos `>>>` (llamaremos a esto el *prompt*), indicando que podemos escribir a continuación una *sentencia* u orden que será evaluada.

Algunas sentencias sencillas, por ejemplo, permiten utilizar el intérprete como una calculadora simple con números enteros. Para esto escribimos la *expresión* que queremos resolver luego del *prompt* y presionamos la tecla Enter. El intérprete de Python evalúa la expresión y muestra el resultado en la línea siguiente. Luego nos presenta nuevamente el *prompt*.

```
>>> 2+3
5
>>>
```

Python permite utilizar las operaciones `+`, `-`, `*`, `/`, `//` y `**` (suma, resta, multiplicación, división, división entera y potenciación). La sintaxis es la convencional (valores intercalados con operaciones), y se puede usar paréntesis para modificar el orden de asociación natural de las operaciones (potenciación, producto/división, suma/resta).

```
>>> 5*7
35
>>> 2+3*7
23
>>> (2+3)*7
35
>>> 10/4
2.5
```

```
>>> 10//4
2
>>> 5**2
25
```

En Python tenemos dos tipos de datos numéricos: los *números enteros* y los *números de punto flotante*. Los números enteros representan el valor entero exacto que ingresemos. Los números flotantes son parecidos a la notación científica, almacenan una cierta cantidad de dígitos significativos y un exponente. Según los operandos y las operaciones que hagamos usaremos la aritmética de los enteros o de los flotantes.

Vamos a elegir enteros cada vez que necesitemos recordar un valor exacto: la cantidad de alumnos, cuántas veces repito una operación, un número de documento, la plata en una cuenta bancaria[1]. En cambio vamos a elegir flotantes cuando nos interese más la magnitud, lo cual suele ser típico en la física y la ingeniería: la temperatura, el seno de un ángulo, la distancia recorrida, el número de Avogadro, el factorial de un número[2].

Además de efectuar operaciones matemáticas, Python nos permite trabajar con porciones de texto, que llamaremos *cadenas*, y que se introducen entre comillas simples (`'`) o dobles (`"`):

```
>>> '¡Hola Mundo!'
'¡Hola Mundo!'
>>> 'abcd' + 'efgh'
'abcdefgh'
>>> 'abcd' * 3
'abcdabcdabcd'
```

1.4 Variables

Python nos permite asignarle un nombre a un valor, de forma tal de "recordarlo" para usarlo posteriormente, mediante la sentencia `<nombre> = <expresión>`.

```
>>> x = 8
>>> x
8
>>> y = x * x
>>> 2 * y
128
>>> lenguaje = 'Python'
>>> 'Estoy programando en ' + lenguaje
'Estoy programando en Python'
```

En este ejemplo creamos tres *variables*, llamadas `x`, `y` y `lenguaje`, y las asociamos a los valores 8, 64 y `'Python'`, respectivamente. Luego podemos usar esas variables como parte de cualquier expresión, y en el momento de evaluarla, Python reemplazará las variables por su valor asociado.

[1] ¿Pero la moneda no tiene decimales?, ¡sí!, pero puedo almacenar los centavos y eso es entero, ¿no es más importante recordar la suma exacta que hay en la cuenta?

[2] ¿Pero el factorial no es entero?, ¡sí!, pero si lo necesito, por ejemplo, para calcular un polinomio de Taylor y ahí está dividiendo me importa más su magnitud que su valor exacto.

1.5 Funciones

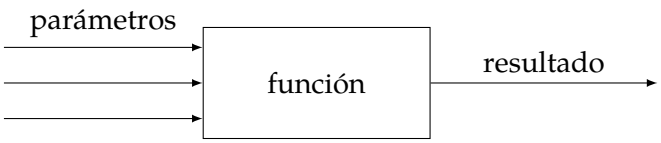
Para efectuar algunas operaciones particulares necesitamos introducir el concepto de *función*:

```
>>> abs(10)
10
>>> abs(-10)
10
>>> max(5, 9, -3)
9
>>> min(5, 9, -3)
-3
>>> len("abcd")
4
```

Una función es un fragmento de programa que permite efectuar una operación determinada. `abs`, `max`, `min` y `len` son ejemplos de funciones de Python: la función `abs` permite calcular el valor absoluto de un número, `max` y `min` permiten obtener el máximo y el mínimo entre un conjunto de números, y `len` permite obtener la longitud de una cadena de texto.

Una función puede recibir 0 o más *parámetros* o *argumentos* (expresados entre entre paréntesis, y separados por comas), efectúa una operación y devuelve un *resultado*. Por ejemplo, la función `abs` recibe un parámetro (un número) y su resultado es el valor absoluto del número.

Figura 1.1: Una función recibe parámetros y devuelve un resultado.



Python viene equipado con muchas funciones, pero ya hemos dicho que, como programadores, debíamos ser capaces de escribir nuevas instrucciones para la computadora. Los programas de correo electrónico, navegación por Internet, chat, juegos, escritura de textos o predicción de las condiciones meteorológicas de los próximos días no son más que grandes programas implementados introduciendo nuevas funciones a la máquina, escritas por uno o muchos programadores.

Si queremos crear una función (que llamaremos `hola_marta`) que devuelve la cadena de texto "Hola Marta! Estoy programando en Python.", lo que debemos hacer es ingresar el siguiente conjunto de líneas en Python:

```
>>> def hola_marta(): ❶
...     return "Hola Marta! Estoy programando en Python." ❷
...
>>>
```

❶ `def hola_marta():` le indica a Python que estamos escribiendo una función cuyo nombre es `hola_marta`, y los paréntesis indican que la función no recibe ningún parámetro.

❷ La instrucción `return` `<expresion>` indica cuál será el resultado de la función.

La sangría[3] con la que se escribe la línea `return` es importante: le indica a Python que estamos escribiendo el *cuerpo* de la función (es decir, las instrucciones que la componen), que podría estar formado por más de una sentencia. La línea en blanco que dejamos luego de la instrucción `return` le indica a Python que terminamos de escribir la función (y por eso aparece nuevamente el *prompt*).

Si ahora queremos que la máquina ejecute la función `hola_marta`, debemos escribir `hola_marta()` a continuación del *prompt* de Python:

```
>>> hola_marta()
'Hola Marta! Estoy programando en Python.'
>>>
```

Se dice que estamos *invocando* a la función `hola_marta`. Al invocar una función, se ejecutan las instrucciones que habíamos escrito en su cuerpo.

Nuestro amigo Pablo seguramente se pondrá celoso porque escribimos una función que saluda a Marta, y nos pedirá que escribamos una función que lo salude a él. Y así procederemos entonces:

```
>>> def hola_pablo():
...     return "Hola Pablo! Estoy programando en Python."
```

Pero, si para cada amigo que quiere que lo saludemos debemos que escribir una función distinta, parecería que la computadora no es una gran solución. A continuación veremos, sin embargo, que podemos llegar a escribir una única función que se personalice en cada invocación, para saludar a quien queramos. Para eso están precisamente los parámetros.

Escribamos entonces una función `hola` que nos sirva para saludar a cualquiera, de la siguiente manera:

```
>>> def hola(alguien):
...     return "Hola " + alguien + "! Estoy programando en Python."
```

En este caso, además de indicar el nombre de la función (`hola`), debemos darle un nombre al parámetro (`alguien`), cuyo valor será reemplazado por una cadena de texto cuando se invoque a la función. Por ejemplo, podemos invocarla dos veces, para saludar a Ana y a Juan:

```
>>> hola("Ana")
'Hola Ana! Estoy programando en Python.'
>>> hola("Juan")
'Hola Juan! Estoy programando en Python.'
```

Problema 1.5.1. Escribir una función que calcule el cuadrado de un número dado.

Solución.

```
def cuadrado(n):
    return n * n
```

Para invocarla, deberemos hacer:

```
>>> cuadrado(5)
25
```

[3] La sangría puede ingresarse utilizando dos o más espacios, o presionando la tecla Tab. Es importante prestar atención en no mezclar espacios con tabs, para evitar "confundir" al intérprete.

Problema 1.5.2. Piensa un número, duplícalo, súmale 6, divídelo por 2 y resta el número que elegiste al comienzo. El número que queda es siempre 3.

Solución. Si bien es muy sencillo probar matemáticamente que el resultado de la secuencia de operaciones será siempre 3 sin importar cuál sea el número elegido, podemos aprovechar nuestros conocimientos de programación y probarlo empíricamente.

Para esto escribamos una función que reciba el número elegido y devuelva el número que queda luego de efectuar las operaciones:

```
def f(elegido):
    return ((elegido * 2) + 6) / 2 - elegido
```

Tal vez el cuerpo de la función quedó poco entendible. Podemos mejorarlo dividiendo la secuencia de operaciones en varias sentencias más pequeñas:

```
def f(elegido):
    n = elegido * 2
    n = n + 6
    n = n / 2
    n = n - elegido
    return n
```

Aquí utilizamos una variable llamada `n` y luego en cada sentencia vamos reemplazando el valor de `n` por un valor nuevo.

Las dos soluciones que presentamos son equivalentes. Veamos si al invocar a `f` con distintos números siempre devuelve 3 o no:

```
>>> f(9)
3.0
>>> f(4)
3.0
>>> f(118)
3.0
>>> f(165414606)
3.0
>>> f(0)
3.0
>>> f(-15)
3.0
```

1.6 Una instrucción un poco más compleja: el ciclo definido

Problema 1.6.1. Supongamos que queremos calcular la suma de los primeros 5 números cuadrados.

Solución. Dado que ya tenemos la función `cuadrado`, podemos aprovecharla y hacer algo como esto:

```
>>> def suma_5_cuadrados():
    suma = 0
    suma = suma + cuadrado(1)
    suma = suma + cuadrado(2)
    suma = suma + cuadrado(3)
```

```
    suma = suma + cuadrado(4)
    suma = suma + cuadrado(5)
    return suma

>>> suma_5_cuadrados()
55
```

Esto resuelve el problema, pero resulta poco satisfactorio. ¿Y si quisiéramos encontrar la suma de los primeros 100 números cuadrados? En ese caso tendríamos que repetir la línea `suma = suma + cuadrado(...)` 100 veces. ¿Se puede hacer algo mejor que esto?

Para resolver este tipo de problema (repetir un cálculo para los valores contenidos en un intervalo dado) de una manera más eficiente, introducimos el concepto de *ciclo definido*, que tiene la siguiente forma:

```
for x in range(n1, n2):
    <hacer algo con x>
```

Esta instrucción se lee como:

- Generar la secuencia de valores enteros del intervalo $[n1, n2)$, y
- Para cada uno de los valores enteros que toma `x` en el intervalo generado, se debe hacer lo indicado por `<hacer algo con x>`.

La instrucción que describe el rango en el que va a realizar el ciclo (`for x in range(...)`) es el *encabezado del ciclo*, y las instrucciones que describen la acción que se repite componen el *cuerpo del ciclo*. Todas las instrucciones que describen el cuerpo del ciclo deben tener una sangría mayor que el encabezado del ciclo.

Solución. Usemos un ciclo definido para resolver el problema anterior de manera más compacta:

```
>>> def suma_5_cuadrados():
...     suma = 0
...     for x in range(1, 6): ❶
...         suma = suma + cuadrado(x)
...     return suma
```

❶ Notar que en nuestro ejemplo necesitamos recorrer todos los valores enteros entre 1 y 5, y el rango generado por `range(n1, n2)` es *abierto* en `n2`. Es decir, `x` tomará los valores `n1`, `n1 + 1`, `n1 + 2`, ..., `n2 - 1`. Por eso es que usamos `range(1, 6)`.

Problema 1.6.2. Hacer una función más genérica que reciba un parámetro `n` y calcule la suma de los primeros `n` números cuadrados.

Solución.

```
>>> def suma_cuadrados(n):
...     suma = 0
...     for x in range(1, n + 1):
...         suma = suma + cuadrado(x)
...     return suma
```

```
>>> suma_cuadrados(5)
55
>>> suma_cuadrados(100)
338350
```

1.7 Construir programas y módulos

El intérprete interactivo es muy útil para probar cosas, acceder a la ayuda, inspeccionar el lenguaje, etc, pero tiene una gran limitación: ¡cuando cerramos el intérprete perdemos todas las definiciones! Para conservar los programas que vamos escribiendo, debemos escribir el código utilizando algún editor de texto, y guardar el archivo con la extensión `.py`.

> **Sabías que…**
>
> El intérprete interactivo de python nos provee una ayuda en línea; es decir, nos puede dar la documentación de cualquier función o instrucción. Para obtenerla llamamos a la función `help()`. Si le pasamos por parámetro el nombre de una función (por ejemplo `help(abs)` o `help(range)`) nos dará la documentación de esa función. Para obtener la documentación de una instrucción la debemos poner entre comillas; por ejemplo: `help('for')`, `help('return')`.

En el código 1.1 se muestra nuestro primer programa, `cuad100.py`, que nos permite calcular la suma de los primeros 100 cuadrados.

Código 1.1 `cuad100.py`**:** Imprime la suma de los primeros 100 números cuadrados

```
1 def suma_cuadrados(n):
2     suma = 0
3     for x in range(1, n + 1):
4         suma = suma + cuadrado(x)
5     return suma
6
7 print("La suma de los primeros 100 cuadrados es", suma_cuadrados(100))
```

En la última línea del programa introducimos una función nueva: `print()`. La función `print` recibe uno o más parámetros de cualquier tipo y los imprime en la pantalla. ¿Por qué no habíamos utilizado `print` hasta ahora?

En el modo interactivo, Python imprime el resultado de cada expresión luego de evaluarla:

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

En cambio, cuando Python ejecuta un programa `.py` no imprime absolutamente nada en la pantalla, a menos que le indiquemos explícitamente que lo haga. Por eso es que en `cuad100.py` debemos llamar a la función `print` para mostrar el resultado.

Para ejecutar el programa debemos abrir una consola del sistema y ejecutar `python cuad100.py`:

```
$ python cuad100.py
La suma de los primeros 100 cuadrados es 338350
```

1.8 Interacción con el usuario

Ya vimos que la función `print` nos permite mostrar información al usuario del programa. En algunos casos también necesitaremos que el usuario ingrese datos al programa. Por ejemplo:

Problema 1.8.1. Escribir en Python un programa que pida al usuario que escriba su nombre, y luego lo salude.

Solución. Ya habíamos escrito la función `hola` que nos permitía saludar a una persona si sabíamos su nombre. Pero aún no sabemos cómo obtener el nombre del usuario. Para esto podemos usar la función `input`, como se muestra en el Código 1.2.

Código 1.2 `saludar.py`**:** Saluda al usuario posr su nombre

```
1 def hola(nombre):
2     return "Hola " + nombre + "!"
3
4 def saludar():
5     nombre = input("Por favor ingrese su nombre: ")
6     saludo = hola(nombre)
7     print(saludo)
8
9 saludar()
```

En la función `saludar` usamos la función `input` para pedirle al usuario su nombre. `input` presenta al usuario el mensaje que le pasamos por parámetro, y luego le permite ingresar una cadena de texto. Cuando el usuario presiona la tecla Enter, `input` devuelve la cadena ingresada. Luego llamamos a `hola` para generar el saludo, y a `print` para mostrarlo al usuario.

Para ejecutar el programa, nuevamente escribimos en la consola del sistema:

```
$ python saludar.py
Por favor ingrese su nombre: Alan
Hola Alan!
```

Problema 1.8.2. Escribir en Python un programa que haga lo siguiente:

1. Muestra un mensaje de bienvenida por pantalla.
2. Le pide al usuario que introduzca dos números enteros $n1$ y $n2$.
3. Imprime el cuadrado de todos los números enteros del intervalo $[n1, n2)$.
4. Muestra un mensaje de despedida por pantalla.

Solución. La solución a este problema se encuentra en el Código 1.3.

Como siempre, podemos ejecutar el programa en la consola del sistema:

```
$ python cuadrados.py
Se calcularán cuadrados de números
Ingrese un número entero: 5
Ingrese otro número entero: 8
25
```

Código 1.3 `cuadrados.py`**:** Imprime los cuadrados solicitados

```
def imprimir_cuadrados():
    print("Se calcularán cuadrados de números")

    n1 = int(input("Ingrese un número entero: "))
    n2 = int(input("Ingrese otro número entero: "))

    for x in range(n1, n2):
        print(x * x)

    print("Es todo por ahora")

imprimir_cuadrados()
```

```
36
49
Es todo por ahora
```

En el Código 1.3 aparece una función que no habíamos utilizado hasta ahora: `int`. ¿Por qué es necesario utilizar `int` para resolver el problema?

En un programa Python podemos operar con cadenas de texto o con números. Las representaciones dentro de la computadora de un número y una cadena son muy distintas. Por ejemplo, el número `12345678` se almacena en forma binaria y típicamente ocupa 4 bytes, mientras que la cadena `"12345678"` es una sucesión de caracteres en la que cada dígito se almacena de forma separada y ocupa (cada uno) un byte.

La función `input` interpreta cualquier valor que el usuario ingresa mediante el teclado como una cadena de caracteres. Es decir, `input` siempre devuelve una cadena, incluso aunque el usuario haya ingresado una secuencia de dígitos.

Por eso es que introducimos la función `int`, que devuelve el parámetro que recibe *convertido* a un número entero:

```
>>> int("42")
42
```

¿... y `return`?

Cuando introdujimos el concepto de función dijimos que una función recibe 0 o más parámetros y devuelve un resultado. Pero en la función `saludar` que escribimos en el Código 1.2 no hay ninguna instrucción `return`... es decir, `saludar` es una función que no recibe parámetros *¡y no devuelve nada!*.

Esto es perfectamente válido: no necesitamos que `saludar` reciba parámetros porque estamos utilizando la función `input` para obtener la entrada del usuario, y no necesitamos que la función devuelva nada, porque su único cometido es *imprimir* un mensaje en la pantalla.

Sin embargo, las funciones que reciben parámetros y devuelven resultados suelen ser mucho más *reutilizables*. En la unidad 3 exploraremos un poco más este concepto.

1.9 Estado y computación

A lo largo de la ejecución de un programa las variables pueden cambiar el valor con el que están asociadas. En un momento dado uno puede detenerse a observar a qué valor se refiere cada una de las variables del programa. Esa "foto" que indica en un momento dado a qué valor hace referencia cada una de las variables se denomina *estado*. También hablaremos del *estado de una variable* para indicar a qué valor está asociada esa variable, y usaremos la notación `n` → 13 para describir el estado de la variable `n` (e indicar que está asociada al número 13).

A medida que las variables cambian de valores a los que se refieren, el programa va cambiando de estado. La sucesión de todos los estados por los que pasa el programa en una ejecución dada se denomina *computación*.

Para ejemplificar estos conceptos veamos qué sucede cuando se ejecuta el programa `cuadrados.py`:

Instrucción	Qué sucede	Estado
`print("Se calcularán cuadrados de números")`	Se despliega el texto "Se calcularán cuadrados de números" en la pantalla.	
`n1 = int(input("Ingrese un número entero: "))`	Se despliega el texto "Ingrese un número entero: " en la pantalla y el programa se queda esperando que el usuario ingrese un número.	
	Supondremos que el usuario ingresa el número 3 y luego oprime la tecla Enter. Se asocia el número 3 con la variable `n1`.	`n1` → 3
`n2 = int(input("Ingrese otro número entero: "))`	Se despliega el texto "Ingrese otro número entero:" en la pantalla y el programa se queda esperando que el usuario ingrese un número.	`n1` → 3
	Supondremos que el usuario ingresa el número 5 y luego oprime la tecla Enter. Se asocia el número 5 con la variable `n2`.	`n1` → 3 `n2` → 5
`for x in range(n1, n2):`	Se asocia el primer número de `[n1,n2)` con la variable `x` y se ejecuta el cuerpo del ciclo.	`n1` → 3 `n2` → 5 `x` → 3
`    print(x * x)`	Se imprime por pantalla el valor de `x * x` (9)	`n1` → 3 `n2` → 5 `x` → 3
`for x in range(n1, n2):`	Se asocia el segundo número de `[n1,n2)` con la variable `x` y se ejecuta el cuerpo del ciclo.	`n1` → 3 `n2` → 5 `x` → 4

`print(x*x)`	Se imprime por pantalla el valor de x * x (16)	`n1` → 3 `n2` → 5 `x` → 4
`for x in range(n1, n2):`	Como no quedan más valores por tratar en `[n1, n2)`, se sale del ciclo.	`n1` → 3 `n2` → 5 `x` → 4
`print("Es todo por ahora")`	Se despliega por pantalla el mensaje "Es todo por ahora"	`n1` → 3 `n2` → 5 `x` → 4

1.9.1 Depuración de programas

Una manera de seguir la evolución del estado es insertar instrucciones de impresión en sitios críticos del programa. Esto nos será de utilidad para detectar errores y también para comprender cómo funcionan determinadas instrucciones.

Por ejemplo, podemos insertar llamadas a la función `print` en el Código 1.3 para inspeccionar el contenido de las variables:

```
def imprimir_cuadrados():
    print("Se calcularán cuadrados de números")

    n1 = int(input("Ingrese un número entero: "))
    print("el valor de n1 es:", n1)
    n2 = int(input("Ingrese otro número entero: "))
    print("el valor de n2 es:", n2)

    for x in range(n1, n2):
        print("el valor de x es:", x)
        print(x * x)

    print("Es todo por ahora")

imprimir_cuadrados()
```

En este caso, la salida del programa será:

```
$ python cuadrados.py
Se calcularán cuadrados de números
Ingrese un número entero: 5
el valor de n1 es: 5
Ingrese otro número entero: 8
el valor de n2 es: 8
el valor de x es: 5
25
el valor de x es: 6
36
el valor de x es: 7
49
Es todo por ahora
```

Si utilizamos este método para depurar el programa, tendremos que recordar eliminar las llamadas `print` una vez que terminemos.

1.10 Ejercicios

Ejercicio 1.10.1. Correr tres veces el programa `cuadrados.py` con valores de entrada (3,5), (3,3) y (5,3) respectivamente. ¿Qué sucede en cada caso?

Ejercicio 1.10.2. La salida del programa `cuadrados.py` es poco informativa. Modificar el programa para que ponga el número junto a su cuadrado.

Ejercicio 1.10.3. Escribir un programa que pregunte al usuario:

a) su nombre, y luego lo salude.

b) dos números, y luego muestre el producto.

Ejercicio 1.10.4. Implementar algoritmos que permitan:

a) Calcular el perímetro de un rectángulo dada su base y su altura.

b) Calcular el área de un rectángulo dada su base y su altura.

c) Calcular el área de un rectángulo (alineado con los ejes x e y) dadas sus coordenadas $x1$, $x2$, $y1$, $y2$.

d) Calcular el perímetro de un círculo dado su radio.

e) Calcular el área de un círculo dado su radio.

f) Calcular el volumen de una esfera dado su radio.

g) Dados los catetos de un triángulo rectángulo, calcular su hipotenusa.

Ejercicio 1.10.5. Mostrar el resultado de ejecutar estos bloques de código en el intérprete de python:

a)
```
>>> for i in range(5):
...     print(i * i)
```

b)
```
>>> for i in range(2, 6):
...     print(i, 2 ** i)
```

Ejercicio 1.10.6. Implementar un algoritmo que, dado un número entero n, permita calcular su factorial.

Ejercicio 1.10.7. Implementar algoritmos que resuelvan los siguientes problemas:

a) Dados dos números, imprimir la suma, resta, división y multiplicación de ambos.

b) Dado un número entero n, imprimir su tabla de multiplicar.

Ejercicio 1.10.8. Escribir un programa que le pida una palabra al usuario, para luego imprimirla 1000 veces, en una única línea, con espacios intermedios.

Ayuda: Investigar acerca del parámetro `end` de la función `print`.

Unidad 2

Programas sencillos

En esta unidad empezaremos a resolver problemas sencillos, y a programarlos en Python.

2.1 Construcción de programas

Cuando nos disponemos a escribir un programa debemos seguir una cierta cantidad de pasos para asegurarnos de que tendremos éxito en la tarea. La acción irreflexiva (me siento frente a la computadora y escribo rápidamente y sin pensar lo que me parece que es la solución) no constituye una actitud profesional (e ingenieril) de resolución de problemas. Toda construcción tiene que seguir una metodología, un protocolo de desarrollo.

Existen muchas metodologías para construir programas, pero en este curso aplicaremos una sencilla, que es adecuada para la construcción de programas pequeños, y que se puede resumir en los siguientes pasos:

1. **Analizar el problema.** Entender profundamente *cuál* es el problema que se trata de resolver, incluyendo el contexto en el cual se usará.

   > Una vez analizado el problema, asentar el análisis por escrito.

2. **Especificar la solución.** Éste es el punto en el cual se describe *qué* debe hacer el programa, sin importar el cómo. En el caso de los problemas sencillos que abordaremos, deberemos decidir cuáles son los datos de entrada que se nos proveen, cuáles son las salidas que debemos producir, y cuál es la relación entre todos ellos.

   > Al especificar el problema a resolver, documentar la especificación por escrito.

3. **Diseñar la solución.** Éste es el punto en el cuál atacamos el *cómo* vamos a resolver el problema, cuáles son los algoritmos y las estructuras de datos que usaremos. Analizamos posibles variantes, y las decisiones las tomamos usando como dato de la realidad el contexto en el que se aplicará la solución, y los costos asociados a cada diseño.

   > Luego de diseñar la solución, asentar por escrito el diseño, asegurándonos de que esté completo.

4. **Implementar el diseño.** Traducir a un lenguaje de programación (en nuestro caso, y por el momento, Python) el diseño que elegimos en el punto anterior.

> La implementación también se debe documentar, con comentarios dentro y fuera del código, al respecto de qué hace el programa, cómo lo hace y por qué lo hace de esa forma.

5. **Probar el programa.** Diseñar un conjunto de pruebas para probar cada una de sus partes por separado, y también la correcta integración entre ellas. Utilizar la *depuración* como instrumento para descubir dónde se producen ciertos errores.

   > Al ejecutar las pruebas, documentar los resultados obtenidos.

6. **Mantener el programa.** Realizar los cambios en respuesta a nuevas demandas.

   > Cuando se realicen cambios, es necesario documentar el análisis, la especificación, el diseño, la implementación y las pruebas que surjan para llevar estos cambios a cabo.

2.2 Realizando un programa sencillo

Al leer un artículo en una revista norteamericana que contiene información de longitudes expresadas en millas, pies y pulgadas, queremos poder convertir esas distancias de modo que sean fáciles de entender. Para ello, decidimos escribir un programa que convierta las longitudes del sistema inglés al sistema métrico decimal.

Antes de comenzar a programar, utilizamos la guía de la sección anterior, para analizar, especificar, diseñar, implementar y probar el problema.

1. **Análisis del problema.** En este caso el problema es sencillo: nos dan un valor expresado en millas, pies y pulgadas y queremos transformarlo en un valor en el sistema métrico decimal. Sin embargo hay varias respuestas posibles, porque no hemos fijado en qué unidad queremos el resultado. Supongamos que decidimos que queremos expresar todo en metros.

2. **Especificación.** Debemos establecer la relación entre los datos de entrada y los datos de salida. Ante todo debemos averiguar los valores para la conversión de las unidades básicas. Buscando en Internet encontramos la siguiente tabla:

   - 1 milla = 1.609344 km
   - 1 pie = 30.48 cm
   - 1 pulgada = 2.54 cm

   > **⚠ Atención**
   >
   > A lo largo de todo el curso usaremos punto decimal, en lugar de coma decimal, para representar valores no enteros, dado que esa es la notación que utiliza Python.

   La tabla obtenida no traduce las longitudes a metros. La manipulamos para llevar todo a metros:

   - 1 milla = 1609.344 m

- 1 pie = 0.3048 m
- 1 pulgada = 0.0254 m

Si una longitud se expresa como L millas, F pies y P pulgadas, su conversión a metros se calculará como:

$$M = 1609.344 * L + 0.3048 * F + 0.0254 * P$$

Hemos especificado el problema. Pasamos entonces a la próxima etapa.

3. **Diseño.** La estructura de este programa es sencilla: leer los datos de entrada, calcular la solución, mostrar el resultado, o *Entrada-Cálculo-Salida*.

   Antes de escribir el programa, escribiremos en *pseudocódigo* (un castellano preciso que se usa para describir lo que hace un programa) una descripción del mismo:

```
Leer cuántas millas tiene la longitud dada
 (y referenciarlo con la variable millas)

Leer cuántos pies tiene la longitud dada
 (y referenciarlo con la variable pies)

Leer cuántas pulgadas tiene la longitud dada
 (y referenciarlo con la variable pulgadas)

Calcular metros = 1609.344 * millas +
    0.3048 * pies + 0.0254 * pulgadas

Mostrar por pantalla la variable metros
```

4. **Implementación.** Ahora estamos en condiciones de traducir este pseudocódigo a un programa en lenguaje Python:

Código 2.1 `ametrico.py`**:** Convierte medidas inglesas a sistema metrico

```
1  def main():
2      print("Convierte medidas inglesas a sistema metrico")
3
4      millas = int(input("Cuántas millas?: "))
5      pies = int(input("Y cuántos pies?: "))
6      pulgadas = int(input("Y cuántas pulgadas?: "))
7
8      metros = 1609.344 * millas + 0.3048 * pies + 0.0254 * pulgadas
9      print("La longitud es de ", metros, " metros")
10
11 main()
```

Nota. En nuestra implementación decidimos dar el nombre `main` a la función principal del programa. Esto no es más que una convención: "main" significa "principal" en inglés.

5. **Prueba.** Probaremos el programa con valores para los que conocemos la solución:

   - 1 milla, 0 pies, 0 pulgadas (el resultado debe ser 1609.344 metros).

- 0 millas, 1 pie, 0 pulgada (el resultado debe ser 0.3048 metros).
- 0 millas, 0 pies, 1 pulgada (el resultado debe ser 0.0254 metros).

La prueba la documentaremos con la sesión de Python correspondiente a las tres invocaciones a `ametrico.py`.

En la sección anterior hicimos hincapié en la necesidad de documentar todo el proceso de desarrollo. En este ejemplo la documentación completa del proceso lo constituye todo lo escrito en esta sección.

2.3 Piezas de un programa Python

Para poder empezar a programar en Python es necesario conocer los elementos que constituyen un programa en dicho lenguaje y las reglas para construirlos.

Cuando empezamos a hablar en un idioma extranjero es posible que nos entiendan pese a que cometamos errores. No sucede lo mismo con los lenguajes de programación: la computadora no nos entenderá si nos desviamos un poco de alguna de las reglas.

2.3.1 Nombres

Ya hemos visto que se usan nombres para denominar a los programas (`ametrico`) y para denominar a las funciones dentro de un módulo (`main`). Cuando queremos dar nombres a valores usamos variables (`millas`, `pies`, `pulgadas`, `metros`). Todos esos nombres se llaman *identificadores* y Python tiene reglas sobre qué es un identificador válido y qué no lo es.

Un identificador comienza con una letra o con guión bajo (_) y luego sigue con una secuencia de letras, números y guiones bajos. Los espacios no están permitidos dentro de los identificadores.

Los siguientes son todos identificadores válidos de Python:

- `hola`
- `hola12t`
- `_hola`
- `Hola`

Python distingue mayúsculas de minúsculas, así que `Hola` es un identificador y `hola` es otro identificador.

Por convención, no usaremos identificadores que empiezan con mayúscula.

Los siguientes son todos identificadores inválidos de Python:

- `hola a12t`
- `8hola`
- `hola\%`

- `Hola*9`

Python reserva 31 palabras para describir la estructura del programa, y no permite que se usen como identificadores. Cuando en un programa nos encontramos con que un nombre no es admitido pese a que su formato es válido, seguramente se trata de una de las palabras de esta lista, a la que llamaremos de *palabras reservadas*. Esta es la lista completa de las palabras reservadas de Python:

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

2.3.2 Expresiones

Una *expresión* es una porción de código Python que produce o calcula un *valor* (resultado).

- La expresión más sencilla es un valor *literal*. Por ejemplo, la expresión `12345` produce el valor numérico 12345.
- Una expresión puede ser una *variable*, y el valor que produce es el que tiene asociada la variable en el estado. Por ejemplo, si `x` → 5 en el estado, entonces el resultado de la expresión `x` es el valor 5.
- Usamos *operaciones* para combinar expresiones y construir expresiones más complejas:
  - Si `x` es como antes, `x + 1` es una expresión cuyo resultado es 6.
  - Si en el estado `millas` → 1, `pies` → 0 y `pulgadas` → 0, entonces `1609.344 * millas + 0.3048 * pies + 0.0254 * pulgadas` es una expresión cuyo resultado es 1609.344.
  - La exponenciación se representa con el símbolo `**`. Por ejemplo, `x**3` significa x^3.
  - Se pueden usar paréntesis para indicar un orden de evaluación: `((b * b) - (4 * a * c)) / (2 * a)`.
  - Igual que en la notación matemática, si no hay paréntesis en la expresión, primero se agrupan las exponenciaciones, luego los productos y cocientes, y luego las sumas y restas.
  - Hay que prestar atención con lo que sucede con los cocientes:
    - La expresión `6 / 4` produce el valor `1.5`.
    - La expresión `6 // 4` produce el valor `1`, que es el resultado de la *división entera* entre 6 y 4.
    - La expresión `6 % 4` produce el valor `2`, que es el *resto de la división entera* entre 6 y 4.

> Los números pueden ser tanto enteros (`111`, `-24`), como reales (`12.5`, `12.0`, `-12.5`). Dentro de la computadora se representan de manera diferente, y se comportan de manera diferente frente a las operaciones. En Python, los números enteros se denominan `int` (de *integer*), y los números reales `float` (de *floating point*).

- Una expresión puede ser una *llamada a una función*: si `f` es una función que recibe un parámetro, y `x` es una variable, la expresión `f(x)` produce el valor que devuelve la función `f` al llamarla pasándole el valor de `x` por parámetro.

  Algunos ejemplos:

  - `input()` produce el valor ingresado por teclado tal como se lo digita.
  - `abs(x)` produce el valor absoluto del número pasado por parámetro.

Ejercicio 2.1. Aplicando las reglas matemáticas de asociatividad, decidir cuáles de las siguientes expresiones son iguales entre sí:

a) `((b * b) - (4 * a * c)) / (2 * a)`

b) `((b * b) - (4 * a * c)) // (2 * a)`

c) `(b * b - 4 * a * c) / (2 * a)`

d) `b * b - 4 * a * c / 2 * a`

e) `(b * b) - (4 * a * c / 2 * a)`

f) `1 / 2 * b`

g) `b / 2`

Ejercicio 2.2. Escribir un programa que le asigne a `a`, `b` y `c` los valores 10, 100 y 1000 respectivamente y evalúe las expresiones del ejercicio anterior.

Ejercicio 2.3. Escribir un programa que le asigne a `a`, `b` y `c` los valores 10.0, 100.0 y 1000.0 respectivamente y evalúe las expresiones del ejercicio anterior.

2.3.3 No sólo de números viven los programas

No sólo tendremos expresiones numéricas en un programa Python. Recordemos el programa que se usó para saludar a muchos amigos:

```
>>> def hola(alguien):
...     return "Hola " + alguien + "! Estoy programando en Python."
```

Para invocar a ese programa y hacer que saludara a Ana había que escribir `hola("Ana")`.

La variable `alguien` en dicha invocación queda ligada a un valor que es una *cadena de caracteres* (letras, dígitos, símbolos, etc.), en este caso, `"Ana"`.

En Python se puede usar también una notación con comillas simples para referirse a las cadenas de caracteres: `'Ana'`.

Como en la sección anterior, veremos las reglas de qué constituyen expresiones con caracteres:

- Una expresión puede ser simplemente una cadena de texto. El resultado de la expresión `'Ana'` es precisamente el valor `'Ana'`.
- Una variable puede estar asociada a una cadena de texto: si `amiga` → `'Ana'` en el estado, entonces el resultado de la expresión `amiga` es el valor `'Ana'`.
- Usamos operaciones para combinar expresiones y construir expresiones más complejas, pero atención con qué operaciones están permitidas sobre cadenas:

  - El signo `+` no representa la suma sino la *concatenación* de cadenas: Si `amiga` es como antes, `amiga + 'Laura'` es una expresión cuyo valor es `AnaLaura`.

Atención

No se puede sumar cadenas con números.

```
>>> amiga="Ana"
>>> amiga+'Laura'
'AnaLaura'
>>> amiga+3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

– El signo * permite repetir una cadena una cantidad de veces: `amiga * 3` es una expresión cuyo valor es `'AnaAnaAna'`.

Atención

No se pueden multiplicar cadenas entre sí

```
>>> amiga * 3
'AnaAnaAna'
>>> amiga * amiga
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

2.3.4 Instrucciones

Las *instrucciones* son las órdenes que entiende Python. Ya hemos usado varias instrucciones:

- hemos retornado valores de una función mediante la instrucción `return`,
- hemos asociado valores con variables,
- hemos usado un ciclo `for` para repetir un cálculo.

2.3.5 Ciclos definidos

Hemos ya usado la instrucción `for` en el programa que calcula cuadrados de enteros en un rango.

```
for x in range(n1, n2):
    print(x * x)
```

Este ciclo se llama definido porque de entrada, y una vez leídos `n1` y `n2`, se sabe exactamente cuántas veces se ejecutará el cuerpo y qué valores tomará `x`.

Un ciclo definido es de la forma

```
for <variable> in <secuencia de valores>:
    <cuerpo>
```

En nuestro ejemplo la secuencia de valores es el intervalo de enteros `[n1, n1+1, ..., n2-1]` y la variable es `x`.

La secuencia de valores se puede indicar como:

- `range(n)`. Establece como secuencia de valores a `[0, 1, ..., n-1]`.
- `range(n1, n2)`. Establece como secuencia de valores a `[n1, n1+1, ..., n2-1]`.
- Se puede definir a mano una secuencia entre corchetes. Por ejemplo,

  ```
  for x in [1, 3, 9, 27]:
      print(x * x)
  ```

  imprimirá los cuadrados de los números 1, 3, 9 y 27.

2.4 Una guía para el diseño

En su artículo "How to program it", Simon Thompson plantea algunas preguntas a sus alumnos que son muy útiles para la etapa de diseño:

- ¿Has visto este problema antes, aunque sea de manera ligeramente diferente?
- ¿Conoces un problema relacionado? ¿Conoces un programa que pueda ser útil?
- Observa la especificación. Intenta encontrar un problema que te resulte familiar y que tenga la misma especificación o una parecida.
- Supongamos que hay un problema relacionado, y que ya fue resuelto. ¿Puedes usarlo? ¿Puedes usar sus resultados? ¿Puedes usar sus métodos? ¿Puedes agregarle alguna parte auxiliar a ese programa del que ya dispones?
- Si no puedes resolver el problema propuesto, intenta resolver uno relacionado. ¿Puedes imaginarte uno relacionado que sea más fácil de resolver? ¿Uno más general? ¿Uno más específico? ¿Un problema análogo?
- ¿Puedes resolver una parte del problema? ¿Puedes sacar algo útil de los datos de entrada? ¿Puedes pensar qué información es útil para calcular las salidas? ¿De qué manera se puede manipular las entradas y las salidas de modo tal que estén "más cerca" unas de las otras?
- ¿Utilizaste todos los datos de entrada? ¿Utilizaste las condiciones especiales sobre los datos de entrada que aparecen en el enunciado? ¿Has tenido en cuenta todos los requisitos que se enuncian en la especificación?

2.5 Calidad de software

Los programas que hemos construido hasta ahora son pequeños y simples. Existen proyectos de software profesionales de tamaños muy diversos, yendo desde programas sencillos desarrollados por una única persona hasta proyectos gigantescos, con millones de líneas de código y desarrollados durante años por miles de personas.

Cuanto más grande es un proyecto de software, más difícil es su construcción y mantenimiento, y más tenemos que prestar atención a la *calidad* con la que está construido. Presentamos aquí una lista no completa de propiedades que contribuyen a la calidad, y algunas preguntas que podemos hacer para medir cuánto contribuye cada factor:

Sabías que…

Uno de los proyectos de código abierto más colosales es el núcleo del sistema operativo Linux. Fue publicado por primera vez en 1991, y aun hoy sigue en desarrollo activo. El código fuente es público[a], y cualquiera puede contribuir aportando mejoras. Hasta la versión 4.13 publicada en 2017 participaron más de 15 000 personas, creando en total más de 24 millones de líneas de código.

[a] https://github.com/torvalds/linux

- **Confiabilidad:** ¿El sistema resuelve el problema inicial en forma correcta? ¿Lo resuelve siempre o a veces falla? ¿Cuántas veces falla en un período de tiempo?
- **Testabilidad:** ¿Qué tan fácil es probar que el sistema funciona correctamente? ¿Hay algún proceso de pruebas automáticas o manuales?
- **Performance:** ¿Cuánto tarda el sistema en producir un resultado? ¿Cuántos recursos consume (memoria, espacio en disco, etc.)?
- **Usabilidad:** ¿Puede un nuevo usuario aprender a utilizar el sistema fácilmente? ¿Las operaciones más comunes son fáciles de realizar?
- **Mantenibilidad:** ¿Qué tan legible y entendible es el código? ¿Qué tan fácil es modificar el comportamiento del programa o agregar nuevas funcionalidades?
- **Escalabilidad:** ¿Cómo se comporta el sistema cuando se incrementa la demanda (cantidad de usuarios, cantidad de datos, etc.)?
- **Portabilidad:** ¿El sistema puede funcionar en diferentes plataformas (arquitecturas de procesador, sistemas operativos, navegadores web, etc.)?
- **Seguridad:** ¿Los datos sensibles están protegidos de ataques informáticos? ¿Qué tan difícil es para un atacante tomar el control, desestabilizar o dañar el sistema?

Por supuesto, cada proyecto es particular y algunos de las propiedades mencionadas tendrán más o menos prioridad según el caso. En particular en este curso nos concentraremos más en que nuestros programas sean confiables y mantenibles, y también prestaremos atención a la performance (sobre todo al comparar diferentes algoritmos).

2.6 Ejercicios

En los ejercicios a continuación, utilizar los conceptos de análisis, especificación y diseño antes de realizar la implementación.

Ejercicio 2.6.1. Ciclos definidos

a) Escribir un ciclo definido para imprimir por pantalla todos los números entre 10 y 20.

b) Escribir un ciclo definido que salude por pantalla a sus cinco mejores amigos/as.

c) Escribir un programa que use un ciclo definido con rango numérico, que pregunte los nombres de sus cinco mejores amigos/as, y los salude.

d) Escribir un programa que use un ciclo definido con rango numérico, que pregunte los nombres de sus seis mejores amigos/as, y los salude.

e) Escribir un programa que use un ciclo definido con rango numérico, que averigue a cuántos amigos quieren saludar, les pregunte los nombres de esos amigos/as, y los salude.

Ejercicio 2.6.2. Escribir un programa que le pregunte al usuario una cantidad de pesos, una tasa de interés y un número de años y muestre como resultado el monto final a obtener. La fórmula a utilizar es:

$$C_n = C \times (1 + \frac{x}{100})^n$$

Donde C es el capital inicial, x es la tasa de interés y n es el número de años a calcular.

Ejercicio 2.6.3. Escribir un programa que convierta un valor dado en grados Fahrenheit a grados Celsius. Recordar que la fórmula para la conversión es: $F = \frac{9}{5}C + 32$

Ejercicio 2.6.4. Utilice el programa anterior para generar una tabla de conversión de temperaturas, desde 0 °F hasta 120 °F, de 10 en 10.

Ejercicio 2.6.5. Escribir un programa que imprima todos los números pares entre dos números que se le pidan al usuario.

Ejercicio 2.6.6. Escribir un programa que le pregunte al usuario un número n e imprima los primeros n números triangulares, junto con su índice. Los números triangulares se obtienen mediante la suma de los números naturales desde 1 hasta n. Es decir, si se piden los primeros 5 números triangulares, el programa debe imprimir:

```
1 - 1
2 - 3
3 - 6
4 - 10
5 - 15
```

Nota: hacerlo usando y sin usar la ecuación $\sum_{i=1}^{n} i = n\,(n+1)/2$. ¿Cuál realiza más operaciones?

Ejercicio 2.6.7. Escribir un programa que tome una cantidad m de valores ingresados por el usuario, a cada uno le calcule el factorial e imprima el resultado junto con el número de orden correspondiente.

Ejercicio 2.6.8. Escribir un programa que imprima por pantalla todas las fichas de dominó, de una por línea y sin repetir.

Ejercicio 2.6.9. Modificar el programa anterior para que pueda generar fichas de un juego que puede tener números de 0 a n.

Unidad 3

Funciones

En la primera unidad vimos que el programador puede definir nuevas instrucciones, que llamamos *funciones*. En particular lo aplicamos a la construcción de una función llamada `hola` que saluda a todos a quienes queramos saludar:

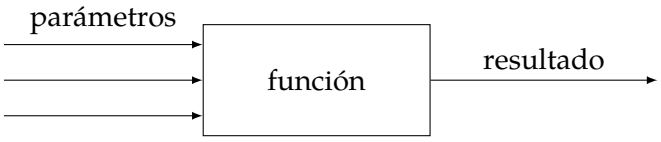
```
def hola(alguien):
    return "Hola " + alguien + "! Estoy programando en Python."
```

La función `hola` recibe un único *parámetro* (`alguien`). Para llamar a una función debemos asociar cada uno de los parámetros con algún valor determinado (que se denomina *argumento*). Por ejemplo, podemos invocar a la función `hola` dos veces, para saludar a Ana y a Juan, haciendo que `alguien` se asocie al valor `"Ana"` en la primera llamada y al valor `"Juan"` en la segunda. La función en cada caso devolverá un *resultado* que que se calcula a partir del argumento.

```
>>> hola("Ana")
'Hola Ana! Estoy programando en Python.'
>>> hola("Juan")
'Hola Juan! Estoy programando en Python.'
```

En general, las funciones pueden recibir ninguno, uno o más parámetros (separados por comas), y pueden o no devolver un resultado.

Figura 3.1: Una función recibe parámetros y devuelve un resultado.



3.1 Documentación de funciones

Cada función escrita por un programador realiza una tarea específica. Cuando la cantidad de funciones disponibles para ser utilizadas es grande, puede ser difícil recordar exactamente qué hace cada función. Es por eso que es extremadamente importante documentar en cada función cuál es la tarea que realiza, cuáles son los parámetros que recibe y qué es lo que devuelve, para que a la hora de utilizarla sea lo pueda hacer correctamente.

La documentación de una función se coloca luego del encabezado de la función, en un párrafo encerrado entre """. Así, para la función vista en el ejemplo anterior:

```
def hola(alguien):
    """Devuelve un saludo dirigido a la persona indicada por parámetro."""
    return "Hola " + alguien + "! Estoy programando en Python."
```

Sabías que…

Cuando una función definida está correctamente documentada, es posible acceder a su documentación mediante la función `help` provista por Python. Suponiendo que la función `hola` está definida en el archivo `saludo.py`:

```
$ python -i saludo
>>> help(hola)
Help on function hola in module __main__:

hola(alguien)
    Devuelve un saludo dirigido a la persona indicada por parámetro.
```

De esta forma no es necesario mirar el código de una función para saber lo que hace, simplemente llamando a `help` es posible obtener esta información.

3.2 Imprimir versus devolver

Supongamos que tenemos una medida de tiempo expresada en horas, minutos y segundos, y queremos calcular la cantidad total de segundos. Cuando nos disponemos a escribir una función en Python para resolver este problema nos enfrentamos con dos posibilidades:

1. *Devolver* el resultado con la instrucción `return`.
2. *Imprimir* el resultado llamando a la función `print`.

A continuación mostramos ambas implementaciones:

```
def devolver_segundos(horas, minutos, segundos):
    """Transforma en segundos una medida de tiempo expresada en
       horas, minutos y segundos"""
    return 3600 * horas + 60 * minutos + segundos

def imprimir_segundos(horas, minutos, segundos):
    """Imprime una medida de tiempo expresada en horas, minutos y
       segundos, luego de transformarla en segundos"""
    print(3600 * horas + 60 * minutos + segundos)
```

Veamos si funcionan:

```
>>> devolver_segundos(1, 10, 10)
4210
>>> imprimir_segundos(1, 10, 10)
4210
```

Aparentemente el comportamiento de ambas funciones es idéntico, pero hay una gran diferencia. La función `devolver_segundos` nos permite hacer algo como esto:

```
>>> s1 = devolver_segundos(1, 10, 10)
>>> s2 = devolver_segundos(2, 32, 20)
>>> s1 + s2
13350
```

En cambio, la función `imprimir_segundos` nos impide utilizar el resultado de la llamada para hacer otras operaciones; lo único que podemos hacer es mostrarlo en pantalla. Por eso decimos que `devolver_segundos` es más *reutilizable*. Por ejemplo, podemos reutilizar `devolver_segundos` en la implementación de `imprimir_segundos`, pero no a la inversa:

```
def imprimir_segundos(horas, minutos, segundos):
    """Imprime una medida de tiempo expresada en horas, minutos y
       segundos, luego de transformarla en segundos"""
    print(devolver_segundos(horas, minutos, segundos))
```

Contar con funciones es de gran utilidad, ya que nos permite ir armando una biblioteca de instrucciones con problemas que vamos resolviendo, y que se pueden reutilizar en la resolución de nuevos problemas (como partes de un problema más grande, por ejemplo) tal como lo sugiere Thompson en "How to program it".

Sin embargo, más útil que tener una biblioteca donde los resultados se imprimen por pantalla, es contar con una biblioteca donde los resultados se devuelven, para que la gente que usa esas funciones manipule esos resultados a voluntad: los imprima, los use para realizar cálculos más complejos, etc.

> En general, una función es más reutilizable si devuelve un resultado (utilizando `return`) en lugar de imprimirlo (utilizando `print`). Análogamente, una función es más reutilizable si recibe parámetros en lugar de leer datos mediante la función `input`.

Ejercicio 3.1. Escribir una función `repite_hola` que reciba como parámetro un número entero `n` y escriba por pantalla el mensaje `"Hola"` `n` veces. Invocarla con distintos valores de `n`.

Ejercicio 3.2. Escribir otra función `repite_hola` que reciba como parámetro un número entero `n` y devuelva la cadena formada por `n` concatenaciones de `"Hola"`. Invocarla con distintos valores de `n`.

Ejercicio 3.3. Escribir una función `repite_saludo` que reciba como parámetro un número entero `n` y una cadena `saludo` y escriba por pantalla el valor de `saludo` `n` veces. Invocarla con distintos valores de `n` y de `saludo`.

Ejercicio 3.4. Escribir otra función `repite_saludo` que reciba como parámetro un número entero `n` y una cadena `saludo` devuelva el valor de `n` concatenaciones de `saludo`. Invocarla con distintos valores de `n` y de `saludo`.

3.3 Cómo usar una función en un programa

Una función es útil porque nos permite repetir la misma instrucción (puede que con argumentos distintos) todas las veces que las necesitemos en un programa.

Supongamos que necesitamos un programa que permita transformar tres duraciones de tiempo en segundos:

1. **Análisis:** El programa debe pedir al usuario tres duraciones expresadas en horas, minutos y segundos, y las tiene que mostrar en pantalla expresadas en segundos.

2. **Especificación:**

   - **Entradas:** Tres duraciones leídas de teclado y expresadas en horas, minutos y segundos.
   - **Salidas:** Mostrar por pantalla cada una de las duraciones ingresadas, convertidas a segundos. Para cada juego de datos de entrada (h, m, s) se obtiene entonces $3600h + 60m + s$, y se muestra ese resultado por pantalla.

3. **Diseño:**

   - Se tienen que leer tres conjuntos de datos y para cada conjunto hacer lo mismo; se trata entonces de un programa con estructura de ciclo definido de tres pasos:

     ```
     repetir 3 veces:
         <hacer cosas>
     ```

   - El cuerpo del ciclo (`<hacer cosas>`) tiene la estructura *Entrada-Cálculo-Salida*. En pseudocódigo:

     ```
     Leer cuántas horas tiene el tiempo dado
      (y referenciarlo con la variable h)

     Leer cuántos minutos tiene tiene el tiempo dado
      (y referenciarlo con la variable m)

     Leer cuántos segundos tiene el tiempo dado
      (y referenciarlo con la variable s)

     Mostrar por pantalla 3600 * h + 60 * m + s
     ```

     Pero la conversión a segundos es exactamente lo que hace nuestra función `devolver_segundos`. Si la renombramos a `a_segundos`, podemos hacer que el cuerpo del ciclo se diseñe como:

     ```
     Leer cuántas horas tiene la duración dada
      (y referenciarlo con la variable h)

     Leer cuántos minutos tiene tiene la duración dada
      (y referenciarlo con la variable m)

     Leer cuántas segundos tiene la duración dada
      (y referenciarlo con la variable s)

     Invocar la función a_segundos(h, m, s) y
     mostrar el resultado en pantalla.
     ```

   - El pseudocódigo final queda:

     ```
     repetir 3 veces:
         Leer cuántas horas tiene la duración dada
          (y referenciarlo con la variable h)

         Leer cuántos minutos tiene la duración dada
     ```

```
     (y referenciarlo con la variable m)

    Leer cuántos segundos tiene la duración dada
     (y referenciarlo con la variable s)

    Invocar la función a_segundos(h, m, s) y
    mostrar el resultado en pantalla.
```

4. **Implementación:** A partir del diseño, se escribe el programa Python que se muestra en el Código 3.1, que se guardará en el archivo `tres_tiempos.py`.

Código 3.1 `tres_tiempos.py`**:** Lee tres tiempos y los imprime en segundos

```
1  def a_segundos(horas, minutos, segundos):
2      """Transforma en segundos una medida de tiempo expresada en
3         horas, minutos y segundos"""
4      return 3600 * horas + 60 * minutos + segundos
5
6  def main():
7      """Lee tres tiempos expresados en horas, minutos y segundos,
8         y muestra en pantalla su conversión a segundos"""
9      for x in range(3):
10         h = int(input("Cuantas horas?: "))
11         m = int(input("Cuantos minutos?: "))
12         s = int(input("Cuantos segundos?: "))
13         print("Son", a_segundos(h, m, s), "segundos")
14
15 main()
```

5. **Prueba:** Probamos el programa con las ternas $(1,0,0)$, $(0,1,0)$ y $(0,0,1)$:

```
$ python tres_tiempos.py
Cuantas horas?: 1
Cuantos minutos?: 0
Cuantos segundos?: 0
Son 3600 segundos
Cuantas horas?: 0
Cuantos minutos?: 1
Cuantos segundos?: 0
Son 60 segundos
Cuantas horas?: 0
Cuantos minutos?: 0
Cuantos segundos?: 1
Son 1 segundos
```

3.4 Alcance de las variables

Ya hemos visto que podemos definir variables, ya sea dentro o fuera del cuerpo de una función. Veamos un ejemplo, utilizando la función `suma_cuadrados` de la unidad 1:

```
>>> def suma_cuadrados(n):
...     suma = 0
```

```
...     for x in range(1, n + 1):
...         suma = suma + cuadrado(x)
...     return suma
```

¿Qué pasa si intentamos utilizar la variable `suma` fuera de la función?

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

Las variables y los parámetros que se declaran dentro de una función no existen fuera de ella, y por eso se las denomina *variables locales*. Fuera de la función se puede acceder únicamente al valor que devuelve mediante `return`.

Veamos en detalle qué sucede cuando invocamos a la función mediante la instrucción:

```
>>> y = suma_cuadrados(5)
```

- Se invoca a `suma_cuadrados` con el argumento 5, y se ejecuta el cuerpo de la función con la variable local `n` → 5.
- La función declara una variable local `suma` → 0.
- Cuando la ejecución llega a la línea `return suma`, la variable `suma` → 55. Por lo tanto, la función devuelve el valor 55.
- La función termina su ejecución, y con ella dejan de existir todas sus variables locales: `n` y `suma`.
- Se declara la variable `y` → 55, que es el valor que devolvió la función.

Si la función no devolviera ningún valor, la variable `y` no quedaría asociada a ningún valor.

3.5 Un ejemplo completo

Problema 3.1. Un usuario nos plantea su problema: necesita que se facture el uso de un teléfono. Nos informará la tarifa por segundo, cuántas comunicaciones se realizaron, la duración de cada comunicación expresada en horas, minutos y segundos. Como resultado deberemos informar la duración en segundos de cada comunicación y su costo.

Solución. Aplicaremos los pasos aprendidos:

1. **Análisis:**
   - ¿Cuántas tarifas distintas se usan? Una sola (la llamaremos p).
   - ¿Cuántas comunicaciones se realizaron? La cantidad de comunicaciones (a la que llamaremos n) se informa cuando se inicia el programa.
   - ¿En qué formato vienen las duraciones de las comunicaciones? Vienen como ternas (h, m, s).

- ¿Qué se hace con esas ternas? Se convierten a segundos y se calcula el costo de cada comunicación multiplicando el tiempo por la tarifa.

2. **Especificación:**

   - **Entradas:**
     - Una tarifa p expresada en pesos/segundo.
     - Una cantidad n de llamadas telefónicas.
     - n duraciones de llamadas leídas de teclado y expresadas en horas, minutos y segundos.
   - **Salidas:** Mostrar por pantalla las n duraciones ingresadas, convertidas a segundos, y su costo. Para cada juego de datos de entrada (h, m, s) se imprime:

$$3600h + 60m + s$$

$$p \cdot (3600h + 60m + s)$$

3. **Diseño:**

   Lo primero que hacemos es buscar un programa que haga algo análogo y ver si se lo puede modificar para resolver nuestro problema. Hay similitudes entre el requerimiento y el programa `tres_tiempos` que desarrollamos anteriormente. Veamos las diferencias entre sus especificaciones.

`tres_tiempos.py`	`tarifador.py`

`repetir 3 veces:`
`  <hacer cosas>`
	`leer el valor de p`
	`leer el valor de n`
	`repetir n veces:`
	`  <hacer cosas>`

El cuerpo del ciclo:
`Leer el valor de h`
`Leer el valor de m`
`Leer el valor de s`
`Mostrar a_segundos(h, m, s)`
	El cuerpo del ciclo:
	`Leer el valor de h`
	`Leer el valor de m`
	`Leer el valor de s`
	`duracion = a_segundos(h, m, s)`
	`costo = duracion * p`
	`Mostrar duracion y costo`

4. **Implementación:** El programa resultante se muestra en el Código 3.2.

5. **Prueba:** Lo probamos con una tarifa de $ 0.40 el segundo y tres ternas de `(1,0,0)`, `(0,1,0)` y `(0,0,1)`:

```
$ python tarifador.py
Cuanto cuesta 1 segundo de comunicacion?: 0.40
Cuantas comunicaciones hubo?: 3
Cuantas horas?: 1
Cuantos minutos?: 0
Cuantos segundos?: 0
```

Código 3.2 `tarifador.py`**:** Programa para calcular el costo de uso de un teléfono.

```
def main():
    """El usuario ingresa la tarifa por segundo, cuántas
       comunicaciones se realizaron, y la duracion de cada
       comunicación expresada en horas, minutos y segundos. Como
       resultado se informa la duración en segundos de cada
       comunicación y su costo."""

    p = float(input("¿Cuánto cuesta 1 segundo de comunicacion?: "))
    n = int(input("¿Cuántas comunicaciones hubo?: "))
    for x in range(n):
        h = int(input("¿Cuántas horas?: "))
        m = int(input("¿Cuántos minutos?: "))
        s = int(input("¿Cuántos segundos?: "))
        duracion = a_segundos(h, m, s)
        costo = duracion * p
        print("Duracion:", duracion, "segundos. Costo: $", costo)

def a_segundos(horas, minutos, segundos):
    """Transforma en segundos una medida de tiempo expresada en
       horas, minutos y segundos"""
    return 3600 * horas + 60 * minutos + segundos

main()
```

```
Duracion: 3600 segundos. Costo: $ 1440.0
Cuantas horas?: 0
Cuantos minutos?: 1
Cuantos segundos?: 0
Duracion: 60 segundos. Costo: $ 24.0
Cuantas horas?: 0
Cuantos minutos?: 0
Cuantos segundos?: 1
Duracion: 1 segundos. Costo: $ 0.4
```

6. **Mantenimiento:**

   Ejercicio 3.5. Corregir el programa para que:

   - Informe el costo en pesos y centavos, en lugar de un número decimal.
   - Informe cuál fue el total facturado en la corrida.

3.6 Devolver múltiples resultados

Problema 3.2. Escribir una función que, dada una duración en segundos sin fracciones (representada por un número entero), calcule la misma duración en horas, minutos y segundos.

Solución. La especificación es sencilla:

- La cantidad de horas es la duración informada en segundos dividida por 3600 (división entera).
- La cantidad de minutos es el resto de la división del paso 1, dividido por 60 (división entera).
- La cantidad de segundos es el resto de la división del paso 2.
- Es importante notar que si la duración no se informa como un número entero, todas las operaciones que se indican más arriba carecen de sentido.

¿Cómo hacemos para devolver más de un valor? En realidad lo que se espera de esta función es que devuelva una terna de valores: si ya calculamos `h`, `m` y `s`, lo que debemos devolver es la terna `(h, m, s)`:

```
def a_hms(segundos):
    """Dada una duración entera en segundos
       se la convierte a horas, minutos y segundos"""
    h = segundos // 3600
    m = (segundos % 3600) // 60
    s = (segundos % 3600) % 60
    return h, m, s
```

Esto es lo que sucede al invocar esta función:

```
>>> h, m, s = a_hms(3661)
>>> print("Son", h, "horas", m, "minutos", s, "segundos")
Son 1 horas 1 minutos 1 segundos
```

Sabías que…

Cuando la función debe devolver múltiples resultados, se empaquetan todos juntos en una *n-upla* (secuencia de valores separados por comas) del tamaño adecuado.

Esta característica está presente en Python, Ruby, Haskell y algunos otros pocos lenguajes. En los lenguajes en los que esta característica no está presente, como C, Pascal o Java, es necesario recurrir a otras técnicas más complejas para poder obtener un comportamiento similar.

Respecto de la variable que hará referencia al resultado de la invocación, se podrá usar tanto una n-upla de variables, como en el ejemplo anterior (en cuyo caso podremos nombrar en forma separada cada uno de los resultados), o bien se podrá usar una sola variable (en cuyo caso se considerará que el resultado tiene un solo nombre y la forma de una n-upla):

```
>>> hms = a_hms(3661)
>>> print(hms)
(1, 1, 1)
```

Atención

Si se usa una n-upla de variables para referirse a un resultado, la cantidad de variables tiene que coincidir con la cantidad de valores que se devuelven.

```
>>> x, y = a_hms(3661)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> x, y, w, z = a_hms(3661)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
```

3.7 Módulos

A medida que los programas se hacen más grandes y complejos suele ser conveniente dividirlos en *módulos*. Cada uno de los programas que escribimos hasta ahora están formados por un único módulo, ya que cada archivo `.py` es un módulo.

Código 3.3 `saludos.py`**:** Módulo con funciones para saludar

```
def hola(nombre)
    return "Hola, " + nombre

def adios(nombre)
    return "Adiós, " + nombre
```

Código 3.4 `main.py`**:** Módulo principal del programa

```
import saludos

def main()
    nombre = input("¿Cuál es tu nombre?")
    print(saludos.hola(nombre))
    print(saludos.adios(nombre))

main()
```

En Código 3.3 y Código 3.4 se muestra un ejemplo de un programa formado por dos módulos, `saludos` y `main`:

- El módulo `saludos` define dos funciones: `hola` y `adios`. Notar que lo único que hacemos es definir funciones pero nunca las llamamos, justamente porque las vamos a invocar desde el módulo `main`.
- Lo primero que hacemos en el módulo `main` es utilizar la instrucción de Python `import saludos`, para indicar al intérprete que queremos utilizar las funciones definidas en el módulo `saludos`. Luego las invocamos, con la diferencia de que tenemos que anteceder el

nombre de cada función con el nombre del módulo y un ".", en este caso `saludos.hola` y `saludos.chau`. Y finalmente llamamos a la función `main()`.

Para ejecutar el programa lo hacemos con el comando `python main.py`. Cuando el intérprete encuentre la instrucción `import saludo` automáticamente buscará el archivo `saludos.py` y lo ejecutará.

3.7.1 Módulos estándar

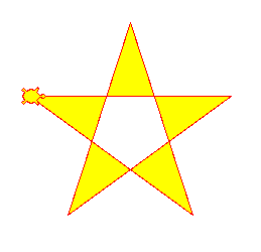
Se dice que "Python viene con las baterías incluidas". Esto es porque el intérprete incluye un conjunto numeroso de módulos ya implementados con utilidades de uso general: matemática, acceso al sistema operativo y la red, depuración, criptografía, compresión, interfaces gráficas… ¡Incluso hay una tortuga!

Sabías que…

El lenguaje de programación *Logo*, creado en 1967 y utilizado principalmente con fines educativos, introdujo la idea de crear dibujos utilizando la metáfora de una *tortuga* que se mueve por la pantalla obedeciendo a comandos simples.

El módulo `turtle` de Python nos permite crear dibujos usando un sistema muy similar al de Logo:

```
import turtle

turtle.shape("turtle")
turtle.color('red', 'yellow')
turtle.begin_fill()
for i in range(5):
    turtle.forward(200)
    turtle.right(144)
turtle.end_fill()
turtle.done()
```



La lista completa de módulos incluidos y sus respectivas instrucciones de uso se puede ver en https://docs.python.org/3/library/index.html.

3.8 Resumen

- Una función puede recibir ninguno, uno o más parámetros. Adicionalmente puede leer datos de la entrada del teclado.
- Una función puede no devolver nada, o devolver uno o más valores. Adicionalmente puede imprimir mensajes para comunicarlos al usuario.
- No es posible acceder a las variables definidas dentro de una función desde el programa principal. Si se quiere utilizar algún valor calculado en la función, será necesario devolverlo.
- Cuando una función realice un cálculo o una operación, es preferible que reciba los datos necesarios mediante los parámetros de la función, y que devuelva el resultado. Las funciones que leen datos del teclado o imprimen mensajes son menos reutilizables.

- Es altamente recomendable documentar cada función que se escribe, para poder saber qué parámetros recibe, qué devuelve y qué hace sin necesidad de leer el código.

Referencia Python

`def funcion(param1, param2, param3):`

Permite definir funciones, que pueden tener ninguno, uno o más parámetros. El cuerpo de la función debe estar un nivel de sangría más adentro que la declaración de la función.

```
def funcion(param1, param2, param3):
    # hacer algo con los parametros
```

Documentación de funciones

Si en la primera línea de la función se ingresa un comentario encerrado entre comillas, este comentario pasa a ser la documentación de la función, que puede ser accedida mendiante el comando `help(funcion)`.

```
def funcion():
    """Esta es la documentación de la función"""
    # hacer algo
```

`return valor`

Dentro de una función se utiliza la instrucción `return` para indicar el valor que la función debe devolver.

Una vez que se ejecuta esta instrucción, se termina la ejecución de la función, sin importar si es la última línea o no.

Si la función no contiene esta instrucción, no devuelve nada.

`return valor1, valor2, valor3`

Si se desea devolver más de un valor, se los *empaqueta* en una n-upla de valores. Esta n-upla puede o no ser desempaquetada al invocar la función:

```
def f(valor):
    # operar
    return a1, a2, a3

# desempaquetado:
v1, v2, v3 = f(x)
# empaquetado
v = f(y)
```

`import modulo`

Permite utilizar funciones y valores definidos en el módulo especificado. Las referencias deben ser precedidas por el nombre del módulo y "`.`".

```
>>> import math
>>> math.cos(2 * math.pi)
1.0
```

`import modulo as variable`

Hace lo mismo que `import modulo`, pero nos permite llamar al módulo con una variable nombrada por nosotros.

```
>>> import math as matematica
>>> matematica.cos(2 * matematica.pi)
1.0
```

`from modulo import ref1, ref2, ...`

Similar a `import modulo`, pero importando únicamente las funciones y valores especificados, y además eliminando la necesidad de anteponer el nombre del módulo al utilizarlos:

```
>>> from math import cos, pi
>>> cos(2 * pi)
1.0
```

3.9 Ejercicios

Ejercicio 3.9.1. Escribir dos funciones que permitan calcular:

a) La duración en segundos de un intervalo dado en horas, minutos y segundos.

b) La duración en horas, minutos y segundos de un intervalo dado en segundos.

Ejercicio 3.9.2. Usando las funciones del ejercicio anterior, escribir un programa que pida al usuario dos intervalos expresados en horas, minutos y segundos, sume sus duraciones, y muestre por pantalla la duración total en horas, minutos y segundos.

Ejercicio 3.9.3. Escribir una función que, dados cuatro números, devuelva el mayor producto de dos de ellos. Por ejemplo, si recibe los números 1, 5, -2, -4 debe devolver 8, que es el producto más grande que se puede obtener entre ellos ($8 = -2 \times -4$).

Ejercicio 3.9.4. Área de polígonos

a) Escribir una función que reciba las coordenadas de un vector en $\mathbb{R}^3$ (`x,y,z`) y devuelva la norma del vector, dada por $\|\overrightarrow{(x,y,z)}\| = \sqrt{x^2+y^2+z^2}$.
Ejemplo: `norma(3, 2, -4)` → `5.3851`

b) Escribir una función que reciba las coordenadas de dos vectores en $\mathbb{R}^3$ (`x1,y1,z1,x2,y2,z2`) y devuelva las coordenadas del vector diferencia (debe devolver 3 valores numéricos).
Ejemplo: `diferencia(8, 7, -3, 5, 3, 2)` → `(3, 4, -5)`

c) Escribir una función que reciba las coordenadas de dos vectores en $\mathbb{R}^3$ y devuelva las coordenadas del producto vectorial, definido como:

$$\overrightarrow{(x_1,y_1,z_1)} \times \overrightarrow{(x_2,y_2,z_2)} = \overrightarrow{(y_1z_2 - z_1y_2,\ z_1x_2 - x_1z_2,\ x_1y_2 - y_1x_2)}$$

Ejemplo: `producto_vec(1, 4, -2, 3, -1, 0)` → `(-2, -6, -13)`

d) Utilizando las funciones anteriores, escribir una función que reciba las coordenadas de 3 puntos en $\mathbb{R}^3$ y devuelva el área del triángulo que conforman.
Ayuda: Si A, B y C son 3 puntos en el espacio, la norma del producto vectorial $\overrightarrow{AB} \times \overrightarrow{AC}$ es igual al doble del área del triángulo ABC.
Ejemplo: `area_triangulo(5, 8, -1, -2, 3, 4, -3, 3, 0)` → `19.4551`

e) Escribir una función que reciba las coordenadas de 4 puntos en el plano $\mathbb{R}^2$ (`x1,y1,x2,y2,x3,y3,x4,y4`) que conforman un cuadrilátero convexo, y devuelva el área del mismo.
Ayuda: Aprovechar las funciones escritas anteriormente, asumiendo que los puntos dados están en $\mathbb{R}^3$ con coordenada $z = 0$.
Ejemplo: `area_cuadrilatero(4, 3, 5, 10, -2, 8, -3, -5)` → `65`

Unidad 4

Decisiones

Problema 4.1. Debemos leer un número y, si el número es positivo, debemos escribir en pantalla el cartel "Número positivo".

Solución. Especificamos nuestra solución: se deberá leer un número x. Si $x > 0$ se escribe el mensaje `"Número positivo"`.

Diseñamos nuestra solución:

1. Solicitar al usuario un número, guardarlo en x.
2. Si $x > 0$, imprimir `"Número positivo"`

Es claro que la primera línea se puede traducir como

```
    x = int(input("Ingrese un número: "))
```

Sin embargo, con las instrucciones que vimos hasta ahora no podemos tomar el tipo de decisiones que nos planteamos en la segunda línea de este diseño.

Para resolver este problema introducimos una nueva instrucción que llamaremos *condicional* y tiene la siguiente forma:

```
if <condición>:
    <hacer algo si se da la condición>
```

Donde `if` es una palabra reservada.

¿Qué es la condición que aparece luego de la palabra reservada `if`? Antes de seguir adelante con la construcción debemos introducir un nuevo tipo de expresión que nos indicará si se da una cierta situación o no. Hasta ahora las expresiones con las que trabajamos fueron de tipo numérica y de tipo texto. Pero ahora la respuesta que buscamos es de tipo *sí* o *no*.

4.1 Expresiones booleanas

Además de los tipos numéricos (`int`, `float`), y las cadenas de texto (`str`), Python introduce un tipo de dato llamado *booleano* (`bool`). Una *expresión booleana* o *expresión lógica* puede tomar dos valores posibles: `True` (sí) o `False` (no).

```
>>> n = 3    # n es de tipo 'int' y toma el valor 3
>>> b = True # b es de tipo 'bool' y toma el valor True
```

4.1.1 Expresiones de comparación

En el ejemplo que queremos resolver, la condición que queremos ver si se cumple o no es que `x` sea mayor que cero. Python provee las llamadas *expresiones de comparación* que sirven para comparar valores entre sí, y que por lo tanto permiten codificar ese tipo de pregunta. En particular la pregunta de si `x` es mayor que cero, se codifica en Python como `x > 0`.

De esta forma, `5 > 3` es una expresión booleana cuyo valor es `True`, y `5 < 3` también es una expresión booleana, pero su valor es `False`.

```
>>> 5 > 3
True
>>> 3 > 5
False
```

Las expresiones booleanas de comparación que provee Python son las siguientes:

Expresión	Significado
`a == b`	`a` es igual a `b`
`a != b`	`a` es distinto de `b`
`a < b`	`a` es menor que `b`
`a <= b`	`a` es menor o igual que `b`
`a > b`	`a` es mayor que `b`
`a >= b`	`a` es mayor o igual que `b`

A continuación, algunos ejemplos de uso de estos operadores:

```
>>> 6 == 6
True
>>> 6 != 6
False
>>> 6 > 6
False
>>> 6 >= 6
True
>>> 6 > 4
True
>>> 6 < 4
False
>>> 6 <= 4
False
>>> 4 < 6
True
```

4.1.2 Operadores lógicos

De la misma manera que se puede operar entre números mediante las operaciones de suma, resta, etc., también existen tres operadores lógicos para combinar expresiones booleanas: `and` (y), `or` (o) y `not` (no).

El significado de estos operadores es igual al del castellano, pero vale la pena recordarlo:

Expresión	Significado
`a and b`	El resultado es `True` solamente si `a` es `True` y `b` es `True` de lo contrario el resultado es `False`
`a or b`	El resultado es `True` si `a` es `True` o `b` es `True` (o ambos) de lo contrario el resultado es `False`
`not a`	El resultado es `True` si `a` es `False` de lo contrario el resultado es `False`

Algunos ejemplos:

- `a > b and a > c` es verdadero si `a` es simultáneamente mayor que `b` y que `c`.

```
>>> 5 > 2 and 5 > 3
True
>>> 5 > 2 and 5 > 6
False
```

- `a > b or a > c` es verdadero si `a` es mayor que `b` o `a` es mayor que `c`.

```
>>> 5 > 2 or 5 > 3
True
>>> 5 > 2 or 5 > 6
True
>>> 5 > 8 or 5 > 6
False
```

- `not a > b` es verdadero si `a > b` es falso (o sea si `a <= b` es verdadero).

```
>>> 5 > 8
False
>>> not 5 > 8
True
>>> 5 > 2
True
>>> not 5 > 2
False
```

4.2 Comparaciones simples

Volvemos al problema que nos plantearon: Debemos leer un número y, si el número es positivo, debemos escribir en pantalla el mensaje `"Número positivo"`.

Recordemos la instrucción `if` que acabamos de introducir y que sirve para tomar decisiones simples. Dijimos que su formato más sencillo es:

```
if <condición>:
    <hacer algo si se da la condición>
```

cuyo significado es el siguiente: se evalúa `<condición>` (que debe ser una expresión lógica), y si el resultado es `True` (verdadero) se ejecutan las acciones indicadas como `<hacer algo si se da la condición>`.

Como ahora ya sabemos también cómo construir condiciones de comparación, estamos en condiciones de implementar nuestra solución. Escribimos la función `positivo()` que hace lo pedido:

```
def positivo():
    x = int(input("Ingrese un número: "))
    if x > 0:
        print("Número positivo")
```

y la probamos:

```
>>> positivo()
Ingrese un número: 4
Número positivo
>>> positivo()
Ingrese un número: -25
>>> positivo()
Ingrese un número: 0
```

Problema 4.2. Necesitamos además un mensaje `"Número no positivo"` cuando no se cumple la condición.

Modificamos la especificación consistentemente y modificamos el diseño:

1. Solicitar al usuario un número, guardarlo en x.
2. Si $x > 0$, imprimir `"Número positivo"`
3. En caso contrario, imprimir `"Número no positivo"`

La negación de $x > 0$ es $\neg(x > 0)$ que se traduce en Python como `not x > 0`, por lo que implementamos nuestra solución en Python como:

```
def positivo_o_no():
    x = int(input("Ingrese un número: "))
    if x > 0: ❶
       print("Número positivo")
    if not x > 0: ❷
       print("Número no positivo")
```

Probamos la nueva solución y obtenemos el resultado buscado:

```
>>> positivo_o_no()
Ingrese un número: 4
Número positivo
>>> positivo_o_no()
Ingrese un número: -25
Número no positivo
>>> positivo_o_no()
Ingrese un número: 0
Número no positivo
```

Sin embargo hay algo que nos preocupa: si ya averiguamos una vez, en ❶, si `x > 0`, ¿Es realmente necesario volver a preguntarlo en ❷?.

Existe una construcción alternativa para la estructura de decisión:

Si se da la condición C, hacer S, de lo contrario, hacer T. Esta estructura tiene la forma:

```
if <condición>:
    <hacer algo si se da la condición>
else:
    <hacer otra cosa si no se da la condición>
```

Donde `if` y `else` son palabras reservadas.

Su significado es el siguiente: se evalúa `<condición>`, si el resultado es `True` se ejecutan las acciones indicadas como `<hacer algo si se da la condición`, y si el resultado es `False` se ejecutan las acciones indicadas como `<hacer otra cosa si no se da la condición>`.

Volvemos a nuestro diseño:

1. Solicitar al usuario un número, guardarlo en x.
2. Si $x > 0$, imprimir `"Número positivo"`
3. En caso contrario, imprimir `"Número no positivo"`

Este diseño se implementa como:

```
def positivo_o_no():
    x = int(input("Ingrese un número: "))
    if x > 0:
       print("Número positivo")
    else:
       print("Número no positivo")
```

y lo probamos:

```
>>> positivo_o_no()
Ingrese un número: 4
Número positivo
>>> positivo_o_no()
Ingrese un número: -25
Número no positivo
>>> positivo_o_no()
Ingrese un número: 0
Número no positivo
```

Es importante destacar que, en general, negar la condición del `if` y poner `else` no son intercambiables, porque no necesariamente producen el mismo efecto en el programa. Notar qué sucede en los dos programas que se transcriben a continuación. ¿Por qué se dan estos resultados?:

```
>>> def pn1():
...    x = int(input("Ingrese un nro: "))
...    if x > 0:
...       print("Número positivo")
...       x = -x
...    if x < 0:
...       print("Número no positivo")
...
>>> pn1()
Ingrese un nro: 25
Número positivo
Número no positivo
```

```
>>> def pn2():
...    x = int(input("Ingrese un nro: "))
...    if x > 0:
...       print("Número positivo")
...       x = -x
...    else:
...       print("Número no positivo")
...
>>> pn2()
Ingrese un nro: 25
Número positivo
```

4.3 Múltiples decisiones consecutivas

La decisión de incluir una decisión en un programa, parte de una lectura cuidadosa de la especificación. En nuestro caso la especificación nos decía:

Si el número es positivo escribir un mensaje `"Número positivo"`, de lo contrario escribir un mensaje `"Número no positivo"`.

Veamos qué se puede hacer cuando se presentan tres o más alternativas:

Problema 4.3. Si el número es positivo escribir un mensaje `"Número positivo"`, si el número es igual a 0 un mensaje `"Igual a 0"`, y si el número es negativo escribir un mensaje `"Número negativo"`.

Una posibilidad es considerar que se trata de una estructura con dos casos como antes, sólo que el segundo caso es complejo (es nuevamente una alternativa):

1. Solicitar al usuario un número, guardarlo en `x`.
2. Si $x > 0$, imprimir `"Número positivo"`
3. De lo contrario:
   (a) Si $x = 0$, imprimir `"Igual a 0"`
   (b) De lo contrario, imprimir `"Número no positivo"`

Este diseño se implementa como:

```
def pos_cero_o_neg():
    x = int(input("Ingrese un número: "))
    if x > 0:
        print("Número positivo")
    else:
        if x == 0:
            print("Igual a 0")
        else:
            print("Número negativo")
```

Esta estructura se conoce como de *alternativas anidadas* ya que dentro de una de las ramas de la alternativa (en este caso la rama del `else`) se anida otra alternativa.

Pero ésta no es la única forma de implementarlo. Existe otra construcción, equivalente a la anterior pero que no exige sangrías cada vez mayores en el texto. Se trata de la estructura de *alternativas encadenadas*, que tiene la forma

```
if <condición_1>:
    <hacer algo_1 si se da la condición_1>
elif <condición_2>:
    <hacer algo_2 si se da la condición_2>
...
...
elif <condición_n>:
    <hacer algo_n si se da la condición_n>
else:
    <hacer otra cosa si no se da ninguna de las condiciones anteriores>
```

Donde `if`, `elif` y `else` son palabras reservadas.
En nuestro ejemplo:

```
def pos_cero_o_neg():
    x = int(input("Ingrese un número: "))
    if x > 0:
```

```
        print("Número positivo")
    elif x == 0:
        print("Igual a 0")
    else:
        print("Número negativo")
```

Se evalúa la primera alternativa, si es verdadera se ejecuta su cuerpo. De lo contrario se evalúa la segunda alternativa, si es verdadera se ejecuta su cuerpo, etc. Finalmente, si todas las alternativas anteriores fallaron, se ejecuta el cuerpo del `else`.

Sabías que…

No sólo mediante los operadores vistos (como > o ==) es posible obtener expresiones booleanas. En Python, se consideran *verdaderos* los valores numéricos distintos de 0, las cadenas de caracteres que no son vacías, y en general cualquier valor que no sea 0 o vacío. Mientras que los valores nulos o vacíos se consideran *falsos*.

Así, en el ejemplo anterior la línea

```
elif x == 0:
```

también podría escribirse de la siguiente manera:

```
elif not x:
```

Además, en Python existe un valor especial llamado `None` que se utiliza comúnmente para representar la ausencia de un valor. Podemos preguntar si una variable `v` es `None` simplemente con:

```
if v is None:
```

O, como `None` también es considerado un valor nulo,

```
if not v:
```

4.4 Ejercicios

Ejercicio 4.1. El usuario del tarifador nos pide ahora una modificación, ya que no es lo mismo la tarifa por segundo de las llamadas cortas que la tarifa por segundo de las llamadas largas. Al inicio del programa se informará la duración máxima de una llamada corta, la tarifa de las llamadas cortas y la de las largas. Se deberá facturar con alguno de los dos valores de acuerdo a la duración de la comunicación.

Ejercicio 4.2. Mantenimiento del tarifador:

a) Al nuevo programa que cuenta con llamadas cortas y largas, agregarle los adicionales, de modo que:

- Los montos se escriban como pesos y centavos.
- Se informe además cuál fue el total facturado en la corrida.

b) Modificar el programa para que sólo informe cantidad de llamadas cortas, valor total de llamadas cortas facturadas, cantidad de llamadas largas, valor total de llamadas largas facturadas, y total facturado. Al llegar a este punto debería ser evidente que es conveniente separar los cálculos en funciones aparte.

Ejercicio 4.3. Dados tres puntos en el plano expresados como coordenadas (x, y) informar cuál es el que se encuentra más lejos del centro de coordenadas.

4.5 Resumen

- Para poder tomar decisiones en los programas y ejecutar una acción u otra, es necesario contar con una **estructura condicional**.
- Las **condiciones** son expresiones *booleanas*, es decir, cuyos valores pueden ser *verdadero* o *falso*, y se las confecciona mediante operadores entre distintos valores.
- Mediante **expresiones lógicas** es posible modificar o combinar expresiones booleanas.
- La estructura condicional puede contar, opcionalmente, con un bloque de código que se ejecuta si no se cumplió la condición.
- Es posible *anidar* estructuras condicionales, colocando una dentro de otra.
- También es posible *encadenar* las condiciones, es decir, colocar una lista de posibles condiciones, de las cuales se ejecuta la primera que sea verdadera.

Referencia Python

`if <condición>:`

Bloque condicional. Las acciones a ejecutar si la condición es verdadera deben tener un mayor nivel de sangría.

```
if <condición>:
    # acciones a ejecutar si condición es verdadera
```

`else:`

Un bloque que se ejecuta cuando no se cumple la condición correspondiente al `if`. Sólo se puede utilizar `else` si hay un `if` correspondiente. Debe escribirse al mismo nivel que `if`, y las acciones a ejecutar deben tener un nivel de sangría mayor.

```
if <condición>:
    # acciones a ejecutar si condición es verdadera
else:
    # acciones a ejecutar si condición es falsa
```

`elif <condición>:`

Bloque que se ejecuta si no se cumplieron las condiciones anteriores pero sí se cumple la condición especificada. Sólo se puede utilizar `elif` si hay un `if` correspondiente, se lo debe escribir al mismo nivel que `if`, y las acciones a ejecutar deben escribirse en un bloque de sangría mayor. Puede haber tantos `elif` como se quiera, todos al mismo nivel.

```
if <condición1>:
    # acciones a ejecutar si condición1 es verdadera
elif <condición2>:
    # acciones a ejecutar si condición2 es verdadera
else:
```

```
    # acciones a ejecutar si ninguna condición fue verdadera
```

Operadores de comparación

Son los que forman las expresiones booleanas.

Expresión	Significado
`a == b`	`a` es igual a `b`
`a != b`	`a` es distinto de `b`
`a < b`	`a` es menor que `b`
`a <= b`	`a` es menor o igual que `b`
`a > b`	`a` es mayor que `b`
`a >= b`	`a` es mayor o igual que `b`

Operadores lógicos

Son los utilizados para concatenar o negar distintas expresiones booleanas.

Expresión	Significado
`a and b`	El resultado es `True` solamente si `a` es `True` y `b` es `True` de lo contrario el resultado es `False`
`a or b`	El resultado es `True` si `a` es `True` o `b` es `True` (o ambos) de lo contrario el resultado es `False`
`not a`	El resultado es `True` si `a` es `False` de lo contrario el resultado es `False`

4.6 Ejercicios

Ejercicio 4.6.1. Escribir dos funciones que resuelvan los siguientes problemas:

a) Dado un número entero n, indicar si es par o no.

b) Dado un número entero n, indicar si es primo o no.

Ejercicio 4.6.2. Escribir una implementación propia de la función `abs`, que devuelva el valor absoluto de cualquier valor que reciba.

Ejercicio 4.6.3. Escribir una función que reciba por parámetro una dimensión n, e imprima la matriz identidad correspondiente a esa dimensión.

Ejercicio 4.6.4. Escribir funciones que permitan encontrar:

a) El máximo o mínimo de un polinomio de segundo grado (dados los coeficientes `a`, `b` y `c`), indicando si es un máximo o un mínimo.

b) Las raíces (reales o complejas) de un polinomio de segundo grado.
Nota: validar que las operaciones puedan efectuarse antes de realizarlas (no dividir por cero, ni calcular la raíz de un número negativo).

c) La intersección de dos rectas (dadas las pendientes y ordenada al origen de cada recta).
Nota: validar que no sean dos rectas con la misma pendiente, antes de efectuar la operación.

Ejercicio 4.6.5. Escribir funciones que resuelvan los siguientes problemas:

a) Dado un año indicar si es bisiesto.
Nota: un año es bisiesto si es un número divisible por 4, pero no si es divisible por 100, excepto que también sea divisible por 400.

b) Dado un mes, devolver la cantidad de días correspondientes.

c) Dada una fecha (día, mes, año), indicar si es válida o no.

d) Dada una fecha, indicar los días que faltan hasta fin de mes.

e) Dada una fecha, indicar los días que faltan hasta fin de año.

f) Dada una fecha, indicar la cantidad de días transcurridos en ese año hasta esa fecha.

g) Dadas dos fechas (día1, mes1, año1, día2, mes2, año2), indicar el tiempo transcurrido entre ambas, en años, meses y días.

Nota: en todos los casos, invocar las funciones escritas previamente cuando sea posible.

Ejercicio 4.6.6. Suponiendo que el primer día del año fue lunes, escribir una función que reciba un número con el día del año (de 1 a 366) y devuelva el día de la semana que le toca. Por ejemplo: si recibe `'3'` debe devolver `'miércoles'`, si recibe `'9'` debe devolver `'martes'`.

Ejercicio 4.6.7. ⋆ Escribir un programa que reciba como entrada un entero representando un año (por ejemplo 751, 1999, o 2158), y muestre por pantalla el mismo año escrito en números romanos.

Ejercicio 4.6.8. Programa de astrología: el usuario debe ingresar el día y mes de su cumpleaños y el programa le debe decir a qué signo corresponde.

Aries: 21 de marzo al 20 de abril.	Tauro: 21 de abril al 20 de mayo.
Geminis: 21 de mayo al 21 de junio.	Cancer: 22 de junio al 23 de julio.
Leo: 24 de julio al 23 de agosto.	Virgo: 24 de agosto al 23 de septiembre.
Libra: 24 de septiembre al 22 de octubre.	Escorpio: 23 de octubre al 22 de noviembre.
Sagitario: 23 de noviembre al 21 de diciembre.	Capricornio: 22 de diciembre al 20 de enero.
Acuario: 21 de enero al 19 de febrero.	Piscis: 20 de febrero al 20 de marzo.

Unidad 5

Más sobre ciclos

El último problema analizado en la unidad anterior decía:

> Leer un número. Si el número es positivo escribir un mensaje "Numero positivo", si el número es igual a 0 un mensaje "Igual a 0", y si el número es negativo escribir un mensaje "Numero negativo".

Se nos plantea a continuación un nuevo problema, similar al anterior:

Problema 5.1. El usuario debe poder ingresar muchos números y cada vez que se ingresa uno debemos informar si es positivo, cero o negativo.

Utilizando los ciclos definidos vistos en las primeras unidades, es posible preguntarle al usuario cada vez, al inicio del programa, cuántos números va a ingresar para consultar. La solución propuesta resulta:

```
def muchos_pcn():
    i = int(input("Cuantos numeros quiere procesar?: "))
    for j in range(0, i):
        x = int(input("Ingrese un numero: "))
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")
```

Su ejecución es exitosa:

```
>>> muchos_pcn()
Cuantos numeros quiere procesar: 3
Ingrese un numero: 25
Numero positivo
Ingrese un numero: 0
Igual a 0
Ingrese un numero: -5
Numero negativo
>>>
```

Sin embargo, el uso de este programa no resulta muy intuitivo, porque obliga al usuario a contar de antemano cuántos números va a querer procesar, sin equivocarse, en lugar de ingresar uno a uno los números hasta procesarlos a todos.

5.1 Ciclos indefinidos

Para poder resolver este problema sin averiguar primero la cantidad de números a procesar, debemos introducir una instrucción que nos permita construir ciclos que no requieran que se informe de antemano la cantidad de veces que se repetirá el cálculo del cuerpo. Se trata de *ciclos indefinidos* en los cuales se repite el cálculo del cuerpo mientras una cierta condición es verdadera.

Un ciclo indefinido es de la forma

```
while <condición>:
    <hacer algo>
```

Donde `while` es una palabra reservada, y la condición es una expresión booleana, igual que en las instrucciones `if`. El cuerpo es, como siempre, una o más instrucciones de Python.

El funcionamiento de esta instrucción es el siguiente:

1. Evaluar la condición.
2. Si la condición es falsa, salir del ciclo.
3. Si la condición es verdadera, ejecutar el cuerpo.
4. Volver a 1.

5.2 Ciclo interactivo

¿Cuál es la condición y cuál es el cuerpo del ciclo en nuestro problema? Claramente, el cuerpo del ciclo es el ingreso de datos y la verificación de si es positivo, negativo o cero. En cuanto a la condición, es que haya más datos para seguir calculando.

Definimos una variable `hay_mas_datos`, que valdrá "Si" mientras haya datos.

Se le debe preguntar al usuario, después de cada cálculo, si hay o no más datos. Cuando el usuario deje de responder "Si", dejaremos de ejecutar el cuerpo del ciclo.

Una primera aproximación al código necesario para resolver este problema podría ser:

```
def muchos_pcn():
    while hay_mas_datos == "Si":
        x = int(input("Ingrese un numero: "))
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        hay_mas_datos = input("¿Quiere seguir? <Si-No>: ")
```

Veamos qué pasa si ejecutamos la función tal como fue presentada:

```
>>> muchos_pcn()
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    muchos_pcn()
  File "<pyshell#24>", line 2, in muchos_pcn
    while hay_mas_datos == "Si":
UnboundLocalError: local variable 'hay_mas_datos' referenced before assignment
```

El problema que se presentó en este caso, es que `hay_mas_datos` no tiene un valor asignado en el momento de evaluar la condición del ciclo por primera vez.

> Es importante prestar atención a cuáles son las variables que hay que inicializar antes de ejecutar un ciclo, para asegurar que la expresión booleana que lo controla sea evaluable.

Una posibilidad es preguntarle al usario, antes de evaluar la condición, si tiene datos; otra posibilidad es suponer que si llamó a este programa es porque tenía algún dato para calcular, y darle el valor inicial "Si" a `hay_mas_datos`.

Encararemos la segunda opción:

```
def muchos_pcn():
    hay_mas_datos = "Si"
    while hay_mas_datos == "Si":
        x = int(input("Ingrese un numero: "))
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        hay_mas_datos = input("Quiere seguir? <Si-No>: ")
```

El esquema del ciclo interactivo es el siguiente:

```
hay_mas_datos hace referencia a "Si"
Mientras hay_mas_datos haga referencia a "Si":
    Pedir datos
    Realizar cálculos
    Preguntar al usuario si hay más datos ("Si" cuando los hay)
    hay_mas_datos hace referencia al valor ingresado
```

Ésta es una ejecución:

```
>>> muchos_pcn()
Ingrese un numero: 25
Numero positivo
Quiere seguir? <Si-No>: Si
Ingrese un numero: 0
Igual a 0
Quiere seguir? <Si-No>: Si
Ingrese un numero: -5
Numero negativo
Quiere seguir? <Si-No>: No
```

5.3 Ciclo con centinela

Un problema que tiene nuestra primera solución es que resulta poco amigable preguntarle al usuario después de cada cálculo si desea continuar. Para evitar esto, se puede usar el método del *centinela*: un valor arbitrario que, si se lee, le indica al programa que el usuario desea salir del ciclo. En este caso, podemos suponer que si el usuario ingresa el caracter *, es una indicación de que desea terminar.

El esquema del ciclo con centinela es el siguiente:

```
Pedir datos
Mientras el dato pedido no coincida con el centinela:
    Realizar cálculos
    Pedir datos
```

El programa resultante es el siguiente:

```
def muchos_pcn():
    centinela = input("Ingrese un numero (* para terminar): ") ❶

    while centinela != "*":
        x = int(centinela)
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        centinela = input("Ingrese un numero (* para terminar): ") ❷
```

Notar que no podemos hacer `centinela = int(input(...))` porque cuando el usuario ingrese `'*'` la llamada a `int` fallaría (al no poder convertir `'*'` a un valor entero). Por eso es que por un lado hacemos la llamada a `input`, y una vez que sabemos que el valor `centinela` no es un `'*'`, lo convertimos a entero llamando a `int`.

Y ahora lo ejecutamos:

```
>>> muchos_pcn()
Ingrese un numero (* para terminar): 25
Numero positivo
Ingrese un numero (* para terminar): 0
Igual a 0
Ingrese un numero (* para terminar): -5
Numero negativo
Ingrese un numero (* para terminar): *
```

El ciclo con centinela es muy claro pero tiene un problema: hay una línea de código repetida, marcada con ❶ y ❷.

Si en la etapa de mantenimiento tuviéramos que realizar un cambio en el ingreso del dato (por ejemplo, cambiar el mensaje) deberíamos estar atentos y corregir ambas líneas. En principio no parece ser un problema muy grave, pero a medida que el programa y el código se hacen más complejos, se hace mucho más difícil llevar la cuenta de todas las líneas de código duplicadas, y por lo tanto se hace mucho más fácil cometer el error de cambiar una de las líneas y olvidar hacer el cambio en la línea duplidada.

> El código duplicado suele incrementar el esfuerzo necesario para hacer modificaciones en la etapa de mantenimiento. Es conveniente prestar atención en a etapa de implementación, y modificar el código para eliminar la duplicación.

Veamos cómo eliminar el código duplicado en nuestro ejemplo. Lo ideal sería leer el dato `centinela` en un único punto del programa. Una opción es *extraer* el código duplicado en una función:

```
def leer_centinela():
    return input("Ingrese un numero (* para terminar): ")
```

```
def muchos_pcn():
    centinela = leer_centinela()
    while centinela != "*":
        x = int(centinela)
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        centinela = leer_centinela()
```

Esta implementación es mejor que la anterior: si tuviéramos que cambiar el mensaje sólo tendríamos que modificar una línea de código. Pero aun tenemos que recordar inicializar la variable `centinela` antes de comenzar el ciclo. ¿Habrá alguna manera de evitarlo?

5.4 Cómo cortar un ciclo

Un ciclo que no puede depender de valores leídos o calculados dentro de él será de la forma:

```
Repetir indefinidamente:
    Hacer algo
```

Y esto se traduce a Python como:

```
while True:
    <hacer algo>
```

Un ciclo cuya condición es `True` parece ser un ciclo infinito (o sea que nunca va a terminar). ¡Pero eso es gravísimo! ¡Nuestros programas tienen que terminar!

Afortunadamente hay una instrucción de Python, `break`, que nos permite salir de adentro de un ciclo (tanto sea `for` como `while`) en medio de su ejecución.

En esta construcción

```
while <condicion_while>:
    <hacer algo_1>
    if <condicion_if>:
        break
    <hacer algo_2>
```

El funcionamiento de la instrucción `break` es el siguiente:

1. Se evalúa la `<condicion_while>` y si es falsa se sale del ciclo.
2. Se ejecuta `<hacer_algo_1>`.
3. Se evalúa la `<condicion_if>` y si es verdadera se sale del ciclo (al llegar a la sintrucción `break`).
4. Se ejecuta `<hacer_algo_2>`.
5. Se vuelve al paso 1.

Diseñamos entonces:

```
Repetir indefinidamente:
    Pedir dato
    Si el dato ingresado es el centinela, salir del ciclo
    Operar con el dato
```

Codificamos en Python la solución al problema de los números usando ese esquema:

```
def muchos_pcn():
    while True:
        centinela = input("Ingrese un numero ('*' para terminar): ")
        if centinela == '*':
            break

        x = int(centinela)
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")
```

5.4.1 Otras formas de controlar el flujo de un ciclo

Ya vimos que la instrucción `break` permite cortar la ejecución de un ciclo:

Cuando la ejecución del cuerpo de un ciclo `for` o `while` llega a una instrucción `break`, la ejecución "sale" del ciclo inmediatamente.

```
def f():
    ...
    while ...:
        ...
        if ...:
            break
        ...
    ...
```

Hay otras dos instrucciones que permiten modificar el flujo del ciclo. Una de ellas es la instrucción `continue`:

Cuando la ejecución del cuerpo de un ciclo `while` llega a una instrucción `continue`, la ejecución "saltea" la iteración actual del ciclo y pasa a la siguiente.

```
def f():
    ...
    while ...:
        ...
        if ...:
            continue
        ...
    ...
```

Si la instrucción `continue` aparece dentro de un ciclo `while`, la condición del ciclo vuelve a evaluarse, y se sale del ciclo si es falsa.

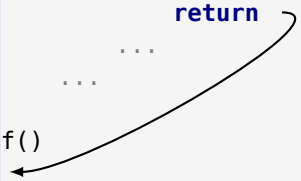
Si la instrucción `continue` aparece dentro de un ciclo `for`, la variable de control del ciclo tomará el siguiente valor del rango de iteración.

```
for x in range(10): # ← luego del continue, la ejecución continúa aquí,
                    # y x tomará el siguiente valor del rango
    print("x vale", x)
    if x % 2 == 0:
        continue
    print("x es impar")
```

Otra instrucción que puede cortar la ejecución del ciclo es `return`:

La instrucción `return` puede aparecer únicamente dentro de una función. Si además está dentro del cuerpo de un bucle `while` o `for`, cortará la ejecución del ciclo y de la función.

```
def f():
    ...
    while ...:
        ...
        if ...:
            return
        ...
    ...

f()
```



Sabías que…

Desde hace mucho tiempo los ciclos infinitos vienen provocando dolores de cabeza a los programadores. Cuando un programa deja de responder y se utiliza todos los recursos de la computadora, suele deberse a que entró en un ciclo del que no puede salir.

Estos bucles pueden aparecer por una gran variedad de causas. A continuación algunos ejemplos de ciclos de los que no se puede salir, siempre o para ciertos parámetros. Queda como ejercicio encontrar el error en cada uno.

```
def menor_factor_primo(x):
    """Devuelve el menor factor primo del número x."""
    n = 2
    while n <= x:
        if x % n == 0:
            return n
```

```
def buscar_impar(x):
    """Divide el número recibido por 2 hasta que sea impar."""
    while x % 2 == 0:
        x = x / 2
    return x
```

5.5 Ejercicios

Ejercicio 5.1. Nuevamente, se desea facturar el uso de un teléfono. Para ello se informa la tarifa por segundo y la duración de cada comunicación expresada en horas, minutos y segundos. Como resultado se informa la duración en segundos de cada llamado y su costo. Resolver este problema usando

1. Un ciclo definido.
2. Un ciclo interactivo.
3. Un ciclo con centinela.
4. Un ciclo “infinito” que se corta.

Ejercicio 5.2. Mantenimiento del tarifador: al final del día se debe informar cuántas llamadas hubo y el total facturado. Hacerlo con todos los esquemas anteriores.

Ejercicio 5.3. Nos piden que escribamos una función que le pida al usuario que ingrese un número positivo. Si el usuario ingresa cualquier cosa que no sea lo pedido se le debe informar de su error mediante un mensaje y volverle a pedir el número.

Resolver este problema usando

1. Un ciclo interactivo.
2. Un ciclo con centinela.
3. Un ciclo “infinito” que se corta.

¿Tendría sentido hacerlo con ciclo definido? Justificar.

5.6 Resumen

- Además de los ciclos definidos, en los que se sabe cuáles son los posibles valores que tomará una determinada variable, existen los ciclos indefinidos, que se terminan cuando no se cumple una determinada condición.
- La condición que termina el ciclo puede estar relacionada con una entrada de usuario o depender del procesamiento de los datos.
- Se puede utilizar el método del *centinela* cuando se quiere que un ciclo se repita hasta que el usuario indique que no quiere continuar.
- Además de la condición que hace que el ciclo se termine, es posible interrumpir su ejecución con código específico dentro del ciclo.

Referencia Python

`while <condicion>:`

Introduce un ciclo indefinido, que se termina cuando la condición sea falsa.

```
while <condición>:
    # acciones a ejecutar mientras condición sea verdadera
```

`break`

Interrumpe la ejecución del ciclo actual. Puede utilizarse tanto para ciclos definidos como indefinidos.

`continue`

"Saltea" la ejecución del ciclo a la siguiente iteración. Puede utilizarse tanto para ciclos definidos como indefinidos.

`return [valor]`

Finaliza la ejecución de una función, y además corta la ejecución del ciclo actual, en caso de estar dentro del cuerpo del ciclo.

5.7 Ejercicios

Ejercicio 5.7.1. Escribir un programa que permita al usuario ingresar un conjunto de notas, preguntando a cada paso si desea ingresar más notas, e imprimiendo el promedio correspondiente.

Ejercicio 5.7.2. Escribir una función que reciba un número entero k e imprima su descomposición en factores primos.

Ejercicio 5.7.3. Manejo de contraseñas

a) Escribir un programa que contenga una contraseña inventada, que le pregunte al usuario la contraseña, y no le permita continuar hasta que la haya ingresado correctamente.

b) Modificar el programa anterior para que solamente permita una cantidad fija de intentos.

c) Modificar el programa anterior para que después de cada intento agregue una pausa cada vez mayor, utilizando la función `sleep` del módulo `time`.

d) Modificar el programa anterior para que sea una función que devuelva si el usuario ingresó o no la contraseña correctamente, mediante un valor booleano (`True` o `False`).

Ejercicio 5.7.4. Utilizando la función `randrange` del módulo `random`, escribir un programa que obtenga un número aleatorio secreto, y luego permita al usuario ingresar números y le indique si son menores o mayores que el número a adivinar, hasta que el usuario ingrese el número correcto.

Ejercicio 5.7.5. Algoritmo de Euclides

a) Implementar el algoritmo de Euclides para calcular el máximo común divisor de dos números n y m, dado por los siguientes pasos.

1. Teniendo n y m, se obtiene r, el resto de la división entera de m/n.
2. Si r es cero, n es el mcd de los valores iniciales.
3. Se reemplaza $m \leftarrow n$, $n \leftarrow r$, y se vuelve al primer paso.

b) Hacer un seguimiento del algoritmo implementado para los siguientes pares de números: $(15, 9)$; $(9, 15)$; $(10, 8)$; $(12, 6)$.

Ejercicio 5.7.6. Potencias de dos.

a) Escribir una función `es_potencia_de_dos` que reciba como parámetro un número natural, y devuelva `True` si el número es una potencia de 2, y `False` en caso contrario.

b) Escribir una función que, dados dos números naturales pasados como parámetros, devuelva la suma de todas las potencias de 2 que hay en el rango formado por esos números (0 si no hay ninguna potencia de 2 entre los dos). Utilizar la función `es_potencia_de_dos`, descripta en el punto anterior.

Ejercicio 5.7.7. Números perfectos y números amigos

a) Escribir una función que devuelva la suma de todos los divisores de un número n, sin incluirlo.

b) Usando la función anterior, escribir una función que imprima los primeros m números tales que la suma de sus divisores sea igual a sí mismo (es decir los primeros m números *perfectos*).

c) Usando la primera función, escribir una función que imprima las primeras m parejas de números (a, b), tales que la suma de los divisores de a es igual a b y la suma de los divisores de b es igual a a (es decir las primeras m parejas de números *amigos*).

d) Proponer optimizaciones a las funciones anteriores para disminuir el tiempo de ejecución.

Ejercicio 5.7.8. Escribir un programa que le pida al usuario que ingrese una sucesión de números naturales (primero uno, luego otro, y así hasta que el usuario ingrese '-1' como condición de salida). Al final, el programa debe imprimir cuántos números fueron ingresados, la suma total de los valores y el promedio.

Ejercicio 5.7.9. Escribir una función que reciba dos números como parámetros, y devuelva cuántos múltiplos del primero hay, que sean menores que el segundo.

a) Implementarla utilizando un ciclo `for`, desde el primer número hasta el segundo.

b) Implementarla utilizando un ciclo `while`, que multiplique el primer número hasta que sea mayor que el segundo.

c) Comparar ambas implementaciones: ¿Cuál es más clara? ¿Cuál realiza menos operaciones?

Ejercicio 5.7.10. Escribir una función que reciba un número natural e imprima todos los números primos que hay hasta ese número.

Ejercicio 5.7.11. Escribir una función que reciba un dígito y un número natural, y decida numéricamente si el dígito se encuentra en la notación decimal del segundo.

Ejercicio 5.7.12. Escribir una función que dada la cantidad de ejercicios de un examen, y el porcentaje necesario de ejercicios bien resueltos necesario para aprobar dicho examen, revise un grupo de examenes. Para ello, en cada paso debe preguntar la cantidad de ejercicios resueltos por el alumno, indicando con un valor centinela que no hay más examenes a revisar. Debe mostrar por pantalla el porcentaje correspondiente a la cantidad de ejercicios resueltos respecto a la cantidad de ejercicios del examen y una leyenda que indique si aprobó o no.

Unidad 6

Cadenas de caracteres

Una cadena es una secuencia de caracteres. Ya las hemos usado para mostrar mensajes, pero sus usos son mucho más amplios que sólo ése: los textos que manipulamos mediante los editores de texto, los textos de Internet que analizan los buscadores, los mensajes enviados mediante correo electrónico, son todos ejemplos de cadenas de caracteres. Pero para poder programar este tipo de aplicaciones debemos aprender a manipularlas. Comenzaremos a ver ahora cómo hacer cálculos con cadenas.

Sabías que…

En Python todos los valores tienen asignado un *tipo*. La función `type` de Python nos permite averiguar de qué tipo es un valor. Las cadenas son de tipo `str`:

```
>>> type(12)
<class 'int'>
>>> type(12.0)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type("Hola")
<class 'str'>
```

6.1 Operaciones con cadenas

Ya vimos en la sección 2.3.3 que es posible:

- Sumar cadenas entre sí (y el resultado es la concatenación de todas las cadenas dadas):

```
>>> "Un divertido " + "programa " + "de " + "radio"
'Un divertido programa de radio'
```

- Multiplicar una cadena `s` por un número `k` (y el resultado es la concatenación de `s` consigo misma, `k` veces):

```
>>> 3 * "programas "
'programas programas programas '
```

```
>>> "programas " * 3
'programas programas programas '
```

A continuación, otras operaciones y particularidades de las cadenas.

6.1.1 Obtener la longitud de una cadena

Se puede averiguar la cantidad de caracteres que conforman una cadena utilizando una función provista por Python: `len`.

```
>>> len("programas ")
10
```

Existe una cadena especial, que llamaremos *cadena vacía*, que es la cadena que no contiene ningún carácter (se la indica sólo con un apóstrofe o comilla que abre, y un apóstrofe o comilla que cierra), y que por lo tanto tiene longitud cero:

```
>>> s = ""
>>> s
''
>>> len(s)
0
```

6.1.2 Una operación para recorrer todos los caracteres de una cadena

Python nos permite recorrer todos los caracteres de una cadena de manera muy sencilla, usando directamente un ciclo definido:

```
>>> for c in "programas ":
...     print(c)
...
p
r
o
g
r
a
m
a
s

>>>
```

6.1.3 Preguntar si una cadena contiene una subcadena

El operador `in` nos permite preguntar si una cadena contiene una subcadena. `a in b` es una expresión que se evalúa a `True` si la cadena `b` contiene la subcadena `a`.

```
>>> 'qué' in 'Hola, ¿qué tal?'
True
>>> '7' in '2468'
False
```

Al ser una expresión booleana, podemos utilizarlo como condición de un `if` o un `while`:

```
if "Hola" in s:
    print("Al parecer la cadena s es un saludo")
```

6.1.4 Acceder a una posición de la cadena

Queremos averiguar cuál es el carácter que está en la posición i-ésima de una cadena. Para ello Python nos provee de una notación con corchetes: escribiremos `s[i]` para hablar de la posición i-ésima de la cadena `s`.

Trataremos de averiguar con qué letra empieza una cadena.

```
>>> s = "Veronica"
>>> s[1]
'e'
```

Algo falló: ¡`s[1]` nos muestra la segunda letra, no la primera! Lo que sucede es que en Python las posiciones se cuentan desde 0.

```
>>> s[0]
'V'
```

Ahora sí hemos conseguido averiguar en Python cuál es el primer carácter de `s`.

Atención

Las distintas posiciones de una cadena `s` se llaman *índices*. Los índices son números enteros que pueden tomar valores entre `-len(s)` y `len(s) - 1`.

Los índices entre `0` y `len(s) - 1` son lo que ya vimos: los caracteres de la cadena del primero al útimo. Los índices negativos proveen una notación que hace más fácil indicar cuál es el último carácter de la cadena: `s[-1]` es el último carácter de `s`, `s[-2]` es el penúltimo carácter de `s`, `s[-len(s)]` es el primer carácter de `s`.

Algunos ejemplos de acceso a distintas posiciones en una cadena.

```
>>> s = "Veronica"
>>> len(s)
8
>>> s[0]
'V'
>>> s[7]
'a'
>>> s[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s[-1]
'a'
>>> s[-8]
'V'
>>> s[-9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Ejercicio 6.1. Escribir un ciclo que permita mostrar los caracteres de una cadena del final al principio.

6.2 Segmentos de cadenas

Python ofrece también una notación para identificar segmentos de una cadena. La notación es similar a la de los rangos que vimos en los ciclos definidos: `s[0:2]` se refiere a la subcadena formada por los caracteres cuyos índices están en el rango `[0,2)`:

```
>>> s[0:2]
'Ve'
>>> s[-4:-2]
'ni'
>>> s[0:8]
'Veronica'
```

Si `j` es un entero no negativo, se puede usar la notación `s[:j]` para representar al segmento `s[0:j]`; también se puede usar la notación `s[j:]` para representar al segmento `s[j:len(s)]`.

```
>>> s[:3]
'Ver'
>>> s[3:]
'onica'
```

Pero hay que tener cuidado con salirse del rango (en particular hay que tener cuidado con la cadena vacía):

```
>>> s[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s = ""
>>> s
''
>>> len(s)
0
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Sin embargo `s[0:0]` no da error. ¿Por qué?

```
>>> s[0:0]
''
```

Ejercicio 6.2. Investigar qué significa la notación `s[:]`.

Ejercicio 6.3. Investigar qué significan las notaciones `s[:j]` y `s[j:]` si `j` es un número negativo.

6.3 Las cadenas son inmutables

Resulta que la persona sobre la que estamos hablando en realidad se llama Veronika, con "k". Como conocemos la notación de corchetes, tratamos de corregir sólo el carácter correspondiente de la variable `s`:

```
>>> s[6] = "k"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

El error que se despliega nos dice que la cadena no soporta la modificación de un carácter. Decimos que *las cadenas son inmutables*.

Si queremos corregir la ortografía de una cadena, debemos hacer que la variable `s` se refiera a una nueva cadena:

```
>>> s = "Veronika"
>>> s
'Veronika'
```

6.4 Procesamiento sencillo de cadenas

Problema 6.1. Nuestro primer problema es muy simple: Queremos contar cuántas letras "A" hay en una cadena `s`.

1. **Especificación:** Dada una cadena `s`, la función retorna un valor `contador` que representa cuántas letras "A" tiene `s`.

2. **Diseño:**

   ¿Se parece a algo que ya conocemos?

   Ante todo es claro que se trata de un ciclo definido, porque lo que hay que tratar es cada uno de los caracteres de la cadena `s`, o sea que estamos frente a un esquema:

   ```
   para cada letra de s
       averiguar si la letra es 'A'
       y tratarla en consecuencia
   ```

   Nos dice la especificación que se necesita una variable `contador` que cuenta la cantidad de letras "A" que contiene `s`. Y por lo tanto sabemos que el tratamiento es: si la letra es "A" se incrementa el contador en 1, y si la letra no es "A" no se lo incrementa, o sea que nos quedamos con un esquema de la forma:

   ```
   para cada letra de s
       averiguar si la letra es 'A'
       y si lo es, incrementar en 1 el contador
   ```

   ¿Estará todo completo? Alicia Hacker nos hace notar que en el diseño no planteamos el retorno del valor del contador. Lo completamos entonces:

   ```
   para cada letra de s
       averiguar si la letra es 'A'
       y si lo es, incrementar en 1 el contador
   retornar el valor del contador
   ```

   ¿Y ahora estará todo completo? E. Lapurado, nuestro alumno impaciente nos induce a poner manos a la obra y a programar esta solución, y el resto del curso está de acuerdo.

3. **Implementación**

   Ya vimos que Python nos provee de un mecanismo para recorrer una cadena: una instrucción `for` que nos brinda un carácter por vez, del primero al último.

   Proponemos la siguiente solución:

```
def contarA(s):
    for letra in s:
        if letra == "A":
            contador = contador + 1
    return contador
```

Y la probamos

```
>>> contarA("Ana")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in contarA
UnboundLocalError: local variable 'contador' referenced before assignment
```

¿Qué es lo que falló? ¡Falló el diseño! Evidentemente la variable `contador` debe tomar un valor inicial antes de empezar a contar las apariciones del caracter "A". Volvamos al diseño entonces.

> Es muy tentador quedarse arreglando la implementación, sin volver al diseño, pero eso es de muy mala práctica, porque el diseño queda mal documentado, y además podemos estar dejando de tener en cuenta otras situaciones erróneas.

4. **Diseño (revisado)** Habíamos llegado a un esquema de la forma

```
para cada letra de s
    averiguar si la letra es 'A'
    y si lo es, incrementar en 1 el contador
retornar el valor del contador
```

¿Cuál es el valor inicial que debe tomar `contador`? Como nos dice la especificación `contador` cuenta la cantidad de letras "A" que tiene la cadena `s`. Pero si nos detenemos en medio de la computación, cuando aún no se recorrió toda la cadena sino sólo los primeros 10 caracteres, por ejemplo, el valor de `contador` refleja la cantidad de "A" que hay en los primeros 10 caracteres de `s`.

Si llamamos *parte izquierda de* `s` al segmento de `s` que ya se recorrió, diremos que cuando leímos los primeros 10 caracteres de `s`, su parte izquierda es el segmento `s[0:10]`.

El valor inicial que debemos darle a `contador` debe reflejar la cantidad de "A" que contiene la parte izquierda de `s` cuando aún no iniciamos el recorrido, es decir cuando esta parte izquierda es `s[0:0]` (o sea la cadena vacía). Pero la cantidad de caracteres iguales a "A" de la cadena vacía es 0.

Por lo tanto el diseño será:

```
inicializar el contador en 0
para cada letra de s
    averiguar si la letra es 'A'
    y si lo es, incrementar en 1 el contador
retornar el valor del contador
```

Lo identificaremos como el esquema *Inicialización - Ciclo de tratamiento - Retorno de valor*. Pasamos ahora a implementar este diseño:

5. **Implementación (del diseño revisado)**

```
def contarA(s):
    """Devuelve cuántas letras "A" aparecen en la cadena s."""
    contador = 0
    for letra in s:
        if letra == "A":
            contador = contador + 1
    return contador
```

6. Prueba

```
>>> contarA("banana")
0
>>> contarA("Ana")
1
>>> contarA("lAn")
1
>>> contarA("lAAn")
2
>>> contarA("lAnA")
2
```

Sabías que…

La instrucción `contador = contador + 1` puede reemplazarse por `contador += 1`.

En general, la mayoría de los operadores tienen versiones abreviadas para cuando la variable que queremos asignar es la misma que el primer operando:

Asignación	Asignación abreviada
`x = x + n`	`x += n`
`x = x - n`	`x -= n`
`x = x * n`	`x *= n`
`x = x / n`	`x /= n`

7. **Mantenimiento:**

   Esta función resulta un poco limitada. Cuando necesitemos contar cuántas letras "E" hay en una cadena tendremos que hacer otra función. Tiene sentido hacer una función más general que nos permita contar cuántas veces aparece un carácter dado en una cadena.

   Ejercicio 6.4. Escribir una función `contar(c, s)` que cuente cuántas veces aparece un carácter `c` dado en una cadena `s`.

   Ejercicio 6.5. ¿Hay más letras "A" o más letras "E" en una cadena? Escribir un programa que lo decida.

   Ejercicio 6.6. Escribir un programa que cuente cúantas veces aparecen cada una de las vocales en una cadena. No importa si la vocal aparece en mayúscula o en minúscula.

6.5 Darle formato a las cadenas

Muchas veces es necesario darle un formato determinado a las cadenas, o dicho de otro modo, procesar los datos de entrada para que las cadenas resultantes se vean de una manera en

particular. Además, separar el formato del texto de los datos a mostrar nos permite enfocarnos en la presentación cuando eso es lo que queremos.

Por ejemplo, para incluir un número en medio de un mensaje, podemos usar la siguiente cadena de formato:

```
>>> no_leidos = 8
>>> "Hay {} mensajes sin leer".format(no_leidos)
'Hay 8 mensajes sin leer'
```

Equivalente a:

```
>>> "Hay " + str(no_leidos) + " mensajes sin leer"
'Hay 8 mensajes sin leer'
```

La función `format` devuelve una cadena en donde se reemplazan todas las marcas `{}` por los valores indicados, que automáticamente se convierten a cadenas de texto.

Al usar la cadenas de formato de esta manera, podemos ver claramente cuál es el contenido del mensaje, evitando errores con respecto a espacios de más o de menos, interrupciones en el texto que complican su lectura, etc.

Sabías que…

La función `format` se utiliza con una sintaxis que hasta ahora no habíamos visto: `cadena.format(valor)` en lugar de la notación usual `format(cadena, valor)`.

Por ahora es suficiente con entender que en la llamada `cadena.format(valor)`, es como si la función `format` recibiera dos parámetros, `cadena` y `valor`.

Esta notación es propia de la *orientación a objetos*. Aprenderemos más sobre objetos en la unidad 14.

Otro ejemplo, utilizando varios valores, en este caso cadenas:

```
>>> "Artista: {}. Disco: {}.".format(artista, disco)
```

Equivalente a:

```
>>> "Artista: " + artista + ". Disco: " + disco + "."
```

En el caso de los valores numéricos, es posible modificar la forma en la que el número es presentado. Por ejemplo, si se trata de un monto monetario, usualmente queremos mostrarlo con dos dígitos decimales, para ello utilizaremos la marca `{:.2}` para indicar dos dígitos luego del separador decimal.

```
>>> precio = 205.5
>>> 'Sin IVA: ${:.2}. Con IVA: ${:.2}'.format(precio, precio * 1.21)
'Sin IVA: $205.50. Con IVA: $248.66'
```

En otras situaciones, como el caso de un valor en un estudio médico, podemos querer mostrar el número en notación científica. En este caso utilizaremos la marca `{:.1e}`, indicando que queremos un dígito significativo luego del separador decimal.

```
>>> rojos = 4640000
>>> 'Glóbulos rojos: {:.1e}/uL'
'Glóbulos rojos: 4.6e+06/uL'
```

6.6 Nuestro primer juego

Con todo esto ya estamos en condiciones de escribir un programa para jugar con la computadora: el *Mastermind*. El Mastermind es un juego que consiste en deducir un código numérico de (por ejemplo) cuatro cifras.

1. **Análisis** (explicación del juego):

   Cada vez que se empieza un partido, el programa debe "eligir" un número de cuatro cifras (sin cifras repetidas), que será el código que el jugador debe adivinar en la menor cantidad de intentos posibles. Cada intento consiste en una propuesta de un código posible que tipea el jugador, y una respuesta del programa. Las respuestas le darán pistas al jugador para que pueda deducir el código.

   Estas pistas indican cuán cerca estuvo el número propuesto de la solución a través de dos valores: la cantidad de *aciertos* es la cantidad de dígitos que propuso el jugador que también están en el código *en la misma posición*. La cantidad de *coincidencias* es la cantidad de digitos que propuso el jugador que también están en el código pero *en una posición distinta*.

   Por ejemplo, si el código que eligió el programa es el `2607`, y el jugador propone el `1406`, el programa le debe responder un acierto (el `0`, que está en el código original en el mismo lugar, el tercero), y una coincidencia (el `6`, que también está en el código original, pero en la segunda posición, no en el cuarto como fue propuesto). Si el jugador hubiera propuesto el `3591`, habría obtenido como respuesta ningún acierto y ninguna coincidencia, ya que no hay números en común con el código original, y si se obtienen cuatro aciertos es porque el jugador adivinó el código y ganó el juego.

2. **Especificación:** El programa, entonces, debe generar un número que el jugador no pueda predecir. A continuación, debe pedirle al usuario que introduzca un número de cuatro cifras distintas, y cuando éste lo ingresa, procesar la propuesta y evaluar el número de aciertos y de coincidencias que tiene de acuerdo al código elegido. Si es el código original, se termina el programa con un mensaje de felicitación. En caso contrario, se informa al jugador la cantidad de aciertos y la de coincidencias, y se le pide una nueva propuesta. Este proceso se repite hasta que el jugador adivine el código.

3. **Diseño:**

   Lo primero que tenemos que hacer es indicarle al programa que tiene que "elegir" un número de cuatro cifras al azar. Esto lo hacemos a través del módulo `random`. Este módulo provee funciones para hacer elecciones aleatorias[1].

   La función del módulo que vamos a usar se llama `choice`. Esta función recibe una secuencia de valores, y devuelve un valor de la secuencia elegido al azar. Una n-upla (ver sección 3.6) es un ejemplo de una secuencia, por lo que podemos hacer algo como:

```
>>> import random
>>> digitos = ('0','1','2','3','4','5','6','7','8','9')
>>> random.choice(digitos)
'4'
>>> random.choice(digitos)
'1'
```

[1] En realidad, la computadora nunca puede hacer elecciones *completamente* aleatorias. Por eso los números "al azar" que puede elegir se llaman *pseudoaleatorios*.

Como están entre comillas, los dígitos son tratados como cadenas de caracteres de longitud uno. Sin las comillas, habrían sido considerados números enteros. En este caso elegimos verlos como cadenas de caracteres porque lo que nos interesa hacer con ellos no son cuentas sino comparaciones, concatenaciones, contar cuántas veces aparece o donde está en una cadena de mayor longitud, es decir, las operaciones que se aplican a cadenas de texto. Entonces que sean variables de tipo cadena de caracteres es lo que mejor se adapta a nuestro problema.

Ahora tenemos que generar el número al azar, asegurándonos de que no haya cifras repetidas. Esto lo podemos modelar así:

```
Tomar una cadena vacía
Repetir cuatro veces:
    1. Elegir un elemento al azar de la lista de dígitos
    2. Si el elemento no está en la cadena, agregarlo
    3. En caso contrario, volver al punto 1
```

Una vez elegido el número, hay que interactuar con el usuario y pedirle su primera propuesta. Si el número no coincide con el código, hay que buscar la cantidad de aciertos y de coincidencias y repetir el pedido de propuestas, hasta que el jugador adivine el código.

Para verificar la cantidad de aciertos se pueden recorrer las cuatro posiciones de la propuesta: si alguna coincide con los dígitos en el código en esa posición, se incrementa en uno la cantidad de aciertos. En caso contrario, se verifica si el dígito está en alguna otra posición del código, y en ese caso se incrementa la cantidad de coincidencias. En cualquier caso, hay que incrementar en uno también la cantidad de intentos que lleva el jugador.

Finalmente, cuando el jugador acierta el código elegido, hay que dejar de pedir propuestas, informar al usuario que ha ganado y terminar el programa.

4. **Implementación:** Entonces, de acuerdo a lo diseñado en 3, se muestra una implementación en el Código 6.1.

5. **Pruebas:** La forma más directa de probar el programa es jugándolo, y verificando manualmente que las respuestas que da son correctas, por ejemplo:

```
$ python mastermind.py
Bienvenido/a al Mastermind!
Tenes que adivinar un numero de 4 cifras distintas
Que codigo propones?: 1234
Tu propuesta (1234) tiene 0 aciertos y  1 coincidencias.
Propone otro codigo: 5678
Tu propuesta (5678) tiene 0 aciertos y  1 coincidencias.
Propone otro codigo: 1590
Tu propuesta (1590) tiene 1 aciertos y  1 coincidencias.
Propone otro codigo: 2960
Tu propuesta (2960) tiene 2 aciertos y  1 coincidencias.
Propone otro codigo: 0963
Tu propuesta (0963) tiene 1 aciertos y  2 coincidencias.
Propone otro codigo: 9460
Tu propuesta (9460) tiene 1 aciertos y  3 coincidencias.
Propone otro codigo: 6940
Felicitaciones! Adivinaste el codigo en 7 intentos.
```

Código 6.1 `mastermind.py`: Juego Mastermind

```
import random

def mastermind():
    """Funcion principal del juego Mastermind"""
    print("Bienvenido/a al Mastermind!")
    print("Tienes que adivinar un numero de cuatro cifras distintas")

    codigo = elegir_codigo()
    intentos = 1
    propuesta = input("Que codigo propones?: ")

    while propuesta != codigo:
        intentos += 1
        aciertos, coincidencias = analizar_propuesta(propuesta, codigo)
        print("Tu propuesta ({}) tiene {} aciertos y {} coincidencias.".format(
            propuesta,
            aciertos,
            coincidencias
        ))
        propuesta = input("Propone otro codigo: ")

    print("Felicitaciones! Adivinaste el codigo en {} intentos.".format(intentos))

def elegir_codigo():
    """Devuelve un codigo de 4 digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(4):
        candidato = random.choice(digitos)
        # Debemos asegurarnos de no repetir digitos
        while candidato in codigo:
            candidato = random.choice(digitos)
        codigo = codigo + candidato
    return codigo

def analizar_propuesta(propuesta, codigo):
    """Determina la cantidad de aciertos y coincidencias"""
    aciertos = 0
    coincidencias = 0
    for i in range(4):
        if propuesta[i] == codigo[i]:
            aciertos = aciertos + 1
        elif propuesta[i] in codigo:
            coincidencias = coincidencias + 1
    return aciertos, coincidencias

mastermind()
```

Podemos ver que para este caso el programa parece haberse comportado bien. ¿Pero cómo podemos saber que el código final era realmente el que eligió originalmente el programa? ¿O qué habría pasado si no encontrábamos la solución?

Para probar estas cosas recurrimos a la depuración del programa. Una forma de hacerlo es simplemente agregar algunas líneas en el código que nos informen lo que está sucediendo que no podemos ver. Por ejemplo, los números que va eligiendo al azar y el código que queda al final. Así podremos verificar si las respuestas son correctas a medida que las hacemos y podremos elegir mejor las propuestas en las pruebas.

```
def elegir_codigo():
    """Devuelve un codigo de 4 digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(4):
        candidato = random.choice(digitos)
        print('[DEBUG] candidato:', candidato)
        # Debemos asegurarnos de no repetir digitos
        while candidato in codigo:
            candidato = random.choice(digitos)
            print('[DEBUG] otro candidato:', candidato)
        codigo = codigo + candidato
        print('[DEBUG] el codigo va siendo:', codigo)
    return codigo
```

De esta manera podemos monitorear cómo se va formando el código que hay que adivinar, y los candidatos que van apareciendo pero se rechazan por estar repetidos:

```
$ python master_debug.py
Bienvenido/a al Mastermind!
Tienes que adivinar un numero de cuatro cifras distintas
[DEBUG] candidato: 8
[DEBUG] el codigo va siendo: 8
[DEBUG] candidato: 0
[DEBUG] el codigo va siendo: 80
[DEBUG] candidato: 2
[DEBUG] el codigo va siendo: 802
[DEBUG] candidato: 8
[DEBUG] otro candidato: 2
[DEBUG] otro candidato: 7
[DEBUG] el codigo va siendo: 8027
Que codigo propones?:
```

6. **Mantenimiento:** Supongamos que queremos jugar el mismo juego, pero en lugar de hacerlo con un número de cuatro cifras, adivinar uno de cinco. ¿Qué tendríamos que hacer para cambiarlo?

   Para empezar, habría que reemplazar el 4 en la línea 28 del programa por un 5, indicando que hay que elegir 5 dígitos al azar. Pero además, el ciclo en la línea 40 también necesita cambiar la cantidad de veces que se va a ejecutar, 5 en lugar de 4. Y hay un lugar más, adentro del mensaje al usuario que indica las instrucciones del juego en la línea 6.

   El problema de ir cambiando estos números de a uno es que si quisiéramos volver al programa de los 4 dígitos o quisiéramos cambiarlo por uno que juegue con 3, tenemos que

volver a hacer los reemplazos en todos lados cada vez que lo queremos cambiar, y corremos el riesgo de olvidarnos de alguno e introducir errores en el código.

Una forma de evitar esto es fijar la cantidad de cifras en una variable:

```
import random

CANT_DIGITOS = 5

def mastermind():
    """Funcion principal del juego Mastermind"""
    print("Bienvenido/a al Mastermind!")
    print("Tienes que adivinar un numero de {} cifras distintas".format(
    ↪ CANT_DIGITOS))
```

Por convención, el nombre de la variable se escribe en mayúsculas, para indicar que el valor asignado es constante a lo largo de la ejecución del programa.

En la función `elegir_codigo`:

```
def elegir_codigo():
    """Devuelve un codigo de CANT_DIGITOS digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(CANT_DIGITOS):
```

Y el chequeo de aciertos y coincidencias:

```
def analizar_propuesta(propuesta, codigo):
    """Determina la cantidad de aciertos y coincidencias"""
    aciertos = 0
    coincidencias = 0
    for i in range(CANT_DIGITOS):
```

Con 5 dígitos, el juego se pone más difícil. Nos damos cuenta que si el jugador no logra adivinar el código, el programa no termina: se queda preguntando códigos y respondiendo aciertos y coincidencias para siempre. Entonces queremos darle al usuario la posibilidad de rendirse y saber cuál era la respuesta y terminar el programa.

Para esto agregamos en el ciclo `while` principal una condición extra: para seguir preguntando, la propuesta tiene que ser distinta al código pero además tiene que ser distinta del texto `"Me doy"`.

```
def mastermind():
    ...
    propuesta = input("Que codigo propones?: ")
    ME_DOY = "Me doy"

    while propuesta != codigo and propuesta != ME_DOY:
```

Entonces, ahora no sólamente sale del `while` si acierta el código, sino además si se rinde y quiere saber cuál era el código. Entonces afuera del while tenemos que separar las dos posibilidades, y dar distintos mensajes:

```
    if propuesta == ME_DOY:
        print("Mala suerte! El código era: {}".format(codigo))
    else:
        print("Felicitaciones! Adivinaste el codigo en {} intentos.".format(
    ↪ intentos))
```

En el Código 6.2 se muestra el código completo luego de aplicar las mejoras.

Ejercicio 6.7. En el punto 6 (Mantenimiento) usamos una variable que guardara el valor de la cantidad de dígitos para no tener que cambiarlo todas las veces. ¿Cómo harían para evitar esta variable usando la función `len(cadena)`?

Ejercicio 6.8. Modificar el programa para permitir repeticiones de dígitos. ¡Cuidado con el cómputo de aciertos y coincidencias!

Código 6.2 `mastermind.py`: Juego Mastermind de 5 dígitos y con posibilidad de rendirse

```
import random

CANT_DIGITOS = 5

def mastermind():
    """Funcion principal del juego Mastermind"""
    print("Bienvenido/a al Mastermind!")
    print("Tienes que adivinar un numero de {} cifras distintas".format(
    ↪ CANT_DIGITOS))

    codigo = elegir_codigo()
    intentos = 1
    propuesta = input("Que codigo propones?: ")
    ME_DOY = "Me doy"

    while propuesta != codigo and propuesta != ME_DOY:
        intentos += 1
        aciertos, coincidencias = analizar_propuesta(propuesta, codigo)
        print("Tu propuesta ({}) tiene {} aciertos y {} coincidencias.".format(
            propuesta,
            aciertos,
            coincidencias
        ))
        propuesta = input("Propone otro codigo: ")

    if propuesta == ME_DOY:
        print("Mala suerte! El código era: {}".format(codigo))
    else:
        print("Felicitaciones! Adivinaste el codigo en {} intentos.".format(
    ↪ intentos))

def elegir_codigo():
    """Devuelve un codigo de CANT_DIGITOS digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(CANT_DIGITOS):
        candidato = random.choice(digitos)
        # Debemos asegurarnos de no repetir digitos
        while candidato in codigo:
            candidato = random.choice(digitos)
        codigo = codigo + candidato
    return codigo

def analizar_propuesta(propuesta, codigo):
    """Determina la cantidad de aciertos y coincidencias"""
    aciertos = 0
    coincidencias = 0
    for i in range(CANT_DIGITOS):
        if propuesta[i] == codigo[i]:
            aciertos = aciertos + 1
        elif propuesta[i] in codigo:
            coincidencias = coincidencias + 1
    return aciertos, coincidencias

mastermind()
```

6.7 Resumen

- Las cadenas de caracteres nos sirven para operar con todo tipo de textos. Contamos con funciones para ver su longitud, sus elementos uno a uno, o por segmentos, comparar estos elementos con otros, etc.

Referencia Python

`len(cadena)`

Devuelve el largo de una cadena, 0 si se trata de una cadena vacía.

`for caracter in cadena`

Permite realizar una acción para cada una de las letras de una cadena.

`subcadena in cadena`

Evalúa a `True` si la `cadena` contiene a la `subcadena`.

`cadena[i]`

Corresponde al valor de la cadena en la posición `i`, comenzando desde 0.
Si se utilizan números negativos, se puede acceder a los elementos desde el último (`-1`) hasta el primero (`-len(cadena)`).

`cadena[i:j]`

Permite obtener un segmento de la cadena, desde la posición `i` inclusive, hasta la posición `j` exclusive.
En el caso de que se omita `i`, se asume `0`. En el caso de que se omita `j`, se asume `len(cadena)`. Si se omiten ambos, se obtiene la cadena completa.

`cadena.format(valores)`

Genera otra cadena reemplazando las marcas de formato por el valor correspondiente, luego de convertirlo de acuerdo a lo especificado en la marca de formato.
Las marcas de formato se indican con `{}`, incluyendo opcionalmente diversos parámetros de conversión.
A continuación una referencia de algunos tipos de conversión utilizados frecuentemente:

Tipo	Significado	Ejemplo
`{:d}`	Valor entero en decimal	`'{:d}'.format(10.5)` → `'10'`
`{:o}`	Valor entero en octal	`'{:o}'.format(8)` → `'10'`
`{:x}`	Valor entero en hexadecimal	`'{:x}'.format(16)` → `'10'`
`{:f}`	Valor de punto flotante, en decimal	`'{:f}'.format(0.1**5)` → `'0.000010'`
`{:.2f}`	Punto flotante, con dos dígitos de precisión	`'{:.2f}'.format(0.1**5)` → `'0.00'`
`{:e}`	Punto flotante, en notación exponencial	`'{:e}'.format(0.1**5)` → `'1.000000e-05'`
`{:g}`	Punto flotante, lo que sea más corto	`'{:g}'.format(0.1**5)` → `'0.00001'`
`{:.2s}`	Cadena recortada en 2 caracteres	`'{:.2s}'.format('Python')` → `'Py'`
`{:<6s}`	Cadena alineada a la izquierda, ocupando 6 caracteres	`'{:<6s}'.format('Py')` → `'Py    '`
`{:^6s}`	Cadena centrada	`'{:^6s}'.format('Py')` → `'  Py  '`
`{:>6s}`	Cadena alineada a la derecha	`'{:>6s}'.format('Py')` → `'    Py'`
`{{`	Un {	
`}}`	Un }	

`cadena.isdigit()`

Devuelve `True` si todos los caracteres de la cadena son dígitos, `False` en caso contrario.

`cadena.isalpha()`

Devuelve `True` si todos los caracteres de la cadena son alfabéticos, `False` en caso contrario.

`cadena.isalnum()`

Devuelve `True` si todos los caracteres de la cadena son alfanuméricos, `False` en caso contrario.

`cadena.capitalize()`

Devuelve `True` si todos los caracteres de la cadena son alfanuméricos, `False` en caso contrario.

`cadena.upper()`

Devuelve una copia de la cadena convertida a mayúsculas.

`cadena.lower()`

Devuelve una copia de la cadena convertida a minúsculas.

6.8 Ejercicios

Ejercicio 6.8.1. Escribir funciones que dada una cadena de caracteres:

a) Imprima los dos primeros caracteres.

b) Imprima los tres últimos caracteres.

c) Imprima dicha cadena cada dos caracteres. Ej.: `'recta'` debería imprimir `'rca'`

d) Dicha cadena en sentido inverso. Ej.: `'hola mundo!'` debe imprimir `'!odnum aloh'`

e) Imprima la cadena en un sentido y en sentido inverso. Ej: `'reflejo'` imprime `'reflejoojelfer'`.

Ejercicio 6.8.2. Escribir funciones que dada una cadena y un caracter:

a) Inserte el caracter entre cada letra de la cadena. Ej: `'separar'` y `','` debería devolver `'s,e,p,a,r,a,r'`

b) Reemplace todos los espacios por el caracter. Ej: `'mi archivo de texto.txt'` y `'_'` debería devolver `'mi_archivo_de_texto.txt'`

c) Reemplace todos los dígitos en la cadena por el caracter. Ej: `'su clave es: 1540'` y `'X'` debería devolver `'su clave es: XXXX'`

d) Inserte el caracter cada 3 dígitos en la cadena. Ej. `'2552552550'` y `'.'` debería devolver `'255.255.255.0'`

Ejercicio 6.8.3. Modificar las funciones anteriores, para que reciban un parámetro que indique la cantidad máxima de reemplazos o inserciones a realizar.

Ejercicio 6.8.4. Escribir una función que reciba una cadena que contiene un largo número entero y devuelva una cadena con el número y las separaciones de miles. Por ejemplo, si recibe `'1234567890'`, debe devolver `'1.234.567.890'`.

Ejercicio 6.8.5. Escribir una función que dada una cadena de caracteres, devuelva:

a) La primera letra de cada palabra. Por ejemplo, si recibe `'Universal Serial Bus'` debe devolver `'USB'`.

b) Dicha cadena con la primera letra de cada palabra en mayúsculas. Por ejemplo, si recibe `'república argentina'` debe devolver `'República Argentina'`.

c) Las palabras que comiencen con la letra 'A'. Por ejemplo, si recibe `'Antes de ayer'` debe devolver `'Antes ayer'`

Ejercicio 6.8.6. Escribir funciones que dada una cadena de caracteres:

a) Devuelva solamente las letras consonantes. Por ejemplo, si recibe `'algoritmos'` o `'logaritmos'` debe devolver `'lgrtms'`.

b) Devuelva solamente las letras vocales. Por ejemplo, si recibe `'sin consonantes'` debe devolver `'i ooae'`.

c) Reemplace cada vocal por su siguiente vocal. Por ejemplo, si recibe `'vestuario'` debe devolver `'vistaerou'`.

d) Indique si se trata de un palíndromo. Por ejemplo, `'anita lava la tina'` es un palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda).

Ejercicio 6.8.7. Escribir funciones que dadas dos cadenas de caracteres:

a) Indique si la segunda cadena es una subcadena de la primera. Por ejemplo, `'cadena'` es una subcadena de `'subcadena'`.

b) Devuelva la que sea anterior en orden alfábetico. Por ejemplo, si recibe `'kde'` y `'gnome'` debe devolver `'gnome'`.

Ejercicio 6.8.8. Escribir una función que reciba una cadena de unos y ceros (es decir, un número en representación binaria) y devuelva el valor decimal correspondiente.

Ejercicio 6.8.9. Implementar la función `pedir_entero(mensaje, min, max)`, que debe imprimir el `mensaje` y luego esperar a que el usuario ingrese un valor. Si el valor ingresado no es un número entero, o no es un número entre `min` y `max` (inclusive), se le debe avisar al usuario y pedir el ingreso de otro valor. Una vez que el usuario ingresa un valor válido, la función lo debe devolver.

Ejemplo:

```
>>> edad = pedir_entero("¿Cuántos años tenés?", 0, 150)
¿Cuántos años tenés? [0-150]:
Por favor ingresa un número entre 0 y 150.
¿Cuántos años tenés? [0-150]: hola
Por favor ingresa un número entre 0 y 150.
¿Cuántos años tenés? [0-150]: -3
Por favor ingresa un número entre 0 y 150.
¿Cuántos años tenés? [0-150]: 151
Por favor ingresa un número entre 0 y 150.
¿Cuántos años tenés? [0-150]: 25
>>> edad
25
```

Unidad 7

Tuplas y listas

Python cuenta con una gran variedad de tipos de datos que permiten representar la información según cómo esté estructurada. En esta unidad se estudian las tuplas y las listas, que son tipos de datos utilizados cuando se quiere agrupar elementos.

7.1 Tuplas

En la conversión de un tiempo a horas, minutos y segundos, en la sección 3.6, usamos n-uplas (o *tuplas*) como una construcción que nos permitía que una función devolviera múltiples valores.

En programación, en general, al querer modelar objetos de la vida real, es muy común que queramos describir un objeto como un agrupamiento de datos de distintos tipos. Veamos algunos ejemplos:

- Una fecha la podemos querer representar como la terna día (un número entero), mes (una cadena de caracteres), y año (un número entero), y tendremos por ejemplo: `(25, "Mayo", 1810)`.
- Como datos de los alumnos queremos guardar número de padrón, nombre y apellido, como por ejemplo `(89766, "Alicia", "Hacker")`.
- **Es posible anidar tuplas:** como datos de los alumnos queremos guardar número de padrón, nombre, apellido y fecha de nacimiento, como por ejemplo: `(89766, "Alicia", "Hacker", (9, "Julio", 1988))`.

En Python el tipo de dato asociado a las tuplas se llama `tuple`:

```
>>> type((25, "Mayo", 1810))
<class 'tuple'>
```

7.1.1 Elementos y segmentos de tuplas

Las tuplas son *secuencias*, igual que las cadenas, y se puede utilizar la misma notación de índices que en las cadenas para obtener cada una de sus componentes.

- El primer elemento de `(25, "Mayo", 1810)` es 25.

- El segundo elemento de `(25, "Mayo", 1810)` es "Mayo".
- El tercer elemento de `(25, "Mayo", 1810)` es 1810.

```
>>> t = (25, "Mayo", 1810)
>>> t[0]
25
>>> t[1]
'Mayo'
>>> t[2]
1810
```

Atención

Todas las secuencias en Python comienzan a numerarse desde 0. Es por eso que se produce un error si se quiere acceder al n-ésimo elemento de un tupla:

```
>>> t[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

También se puede utilizar la notación de rangos, que se vio aplicada a cadenas para obtener una nueva tupla, con un subconjunto de componentes. Si en el ejemplo de la fecha queremos quedarnos con un par que sólo contenga día y mes podremos tomar el rango `[:2]` de la misma:

```
>>> t[:2]
(25, 'Mayo')
```

Ejercicio 7.1. ¿Cuál es el cuarto elemento de la tupla `(89766, "Alicia", "Hacker", (9, "Julio", 1988))`?

7.1.2 Las tuplas son inmutables

Al igual que con las cadenas, las componentes de las tuplas no pueden ser modificadas:

```
>>> t[2] = 2008
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

7.1.3 Longitud de tuplas

A las tuplas también se les puede aplicar la función `len()` para calcular su longitud. El valor de esta función aplicada a una tupla nos indica cuántas componentes tiene esa tupla.

```
>>> len(t)
3
```

Ejercicio 7.2. ¿Cuál es la longitud de la tupla `(89766, "Alicia", "Hacker", (9, "Julio", 1988))`?

- Una *tupla vacía* es una tupla con 0 componentes, y se la indica como `()`.

```
>>> z = ()
>>> len(z)
0
>>> z[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

- Una *tupla unitaria* es una tupla con una componente. Para distinguir la tupla unitaria de la componente que contiene, Python exige que a la componente no sólo se la encierre entre paréntesis sino que se le ponga una coma a continuación del valor de la componente (así `(1810)` es un número, pero `(1810,)` es la tupla unitaria cuya única componente vale 1810).

```
>>> u = (1810)
>>> len(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
>>> u = (1810,)
>>> len(u)
1
>>> u[0]
1810
```

7.1.4 Empaquetado y desempaquetado de tuplas

Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por todos los valores asignados. A esta operación se la denomina *empaquetado de tuplas*.

```
>>> a = 125
>>> b = "#"
>>> c = "Ana"
>>> d = a, b, c
>>> len(d)
3
>>> d
(125, '#', 'Ana')
```

Si se tiene una tupla de longitud `k`, se puede asignar la tupla a `k` variables distintas y en cada variable quedará una de las componentes de la tupla. A esta operación se la denomina *desempaquetado de tuplas*.

```
>>> x, y, z = d
>>> x
125
>>> y
'#'
>>> z
'Ana'
```

Atención

Si las variables no son distintas, se pierden valores. Y si las variables no son exactamente k se produce un error.

```
>>> p, p, p = d
>>> p
'Ana'
>>> m, n = d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> m, n, o, p = d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
```

Sabías que…

En la implementación de algunos programas suele ser necesario intercambiar el valor de dos variables. Si queremos intercambiar el valor de `a` y `b`, una posibilidad es utilizar una variable auxiliar:

```
aux = a
a = b
b = aux
```

Pero un "truco" que permiten algunos lenguajes, entre ellos Python, es la posibilidad de empaquetar y desempaquetar una tupla en una única operación:

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

7.1.5 Ejercicios con tuplas

Ejercicio 7.3. Cartas como tuplas.

a) Proponer una representación con tuplas para las cartas de la baraja francesa.

b) Escribir una función `poker` que reciba cinco cartas de la baraja francesa e informe (devuelva el valor lógico correspondiente) si esas cartas forman o no un *poker* (es decir que hay 4 cartas con el mismo número).

Ejercicio 7.4. El tiempo como tuplas.

a) Proponer una representación con tuplas para representar el tiempo.

b) Escribir una función `sumar_tiempos` que reciba dos tiempos dados y devuelva su suma.

Ejercicio 7.5. Escribir una función `dia_siguiente` que dada una fecha expresada como la terna `(Día, Mes, Año)` (donde `Día`, `Mes` y `Año` son números enteros) calcule el día siguiente al dado, en el mismo formato.

Ejercicio 7.6. Escribir una función `dia_siguiente_m` que dada una fecha expresada como la terna `(Día, Mes, Año)` (donde `Día` y `Año` son números enteros, y `Mes` es el texto `"Ene"`, `"Feb"`, ..., `"Dic"`, según corresponda) calcule el día siguiente al dado, en el mismo formato.

7.2 Listas

Presentaremos ahora una nueva estructura de datos: la *lista*. Usaremos listas para poder modelar datos compuestos, pero cuya cantidad y valor varían a lo largo del tiempo. Son secuencias *mutables* y vienen dotadas de una variedad de operaciones muy útiles.

La notación para lista es una secuencia de valores encerrados entre corchetes y separados por comas. Por ejemplo, si representamos a los alumnos mediante su número de padrón, se puede tener una lista de inscriptos en la materia como la siguiente: `[78455, 89211, 66540, 45750]`. Al abrirse la inscripción, antes de que hubiera inscriptos, la lista de inscriptos se representará por una lista vacía: `[]`.

En Python el tipo de dato asociado a las listas se llama `list`:

```
>>> type([78455, 89211, 66540, 45750])
<class 'list'>
```

7.2.1 Longitud de la lista. Elementos y segmentos de listas

- Como a las secuencias ya vistas, a las listas también se les puede aplicar la función `len()` para conocer su longitud.
- Para acceder a los distintos elementos de la lista se utilizará la misma notación de índices de cadenas y tuplas, con valores que van de 0 a la longitud de la lista -1.

```
>>> padrones = [78455, 89211, 66540, 45750]
>>> padrones[0]
78455
>>> len(padrones)
4
>>> padrones[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> padrones[3]
45750
```

- Para obtener una sublista a partir de la lista original, se utiliza la notación de rangos, como en las otras secuencias.

  Para obtener la lista que contiene sólo a quién se inscribió en segundo lugar podemos escribir:

```
>>> padrones[1:2]
[89211]
```

  Para obtener la lista que contiene al segundo y tercer inscriptos podemos escribir:

```
>>> padrones[1:3]
[89211, 66540]
```

  Para obtener la lista que contiene al primero y segundo inscriptos podemos escribir:

```
>>> padrones[:2]
[78455, 89211]
```

7.2.2 Cómo mutar listas

Dijimos antes que las listas son secuencias mutables. Para lograr la mutabilidad Python provee operaciones que nos permiten cambiarle valores, agregarle valores y quitarle valores.

- Para cambiar una componente de una lista, se selecciona la componente mediante su índice y se le asigna el nuevo valor:

```
>>> padrones[1] = 79211
>>> padrones
[78455, 79211, 66540, 45750]
```

- Para agregar un nuevo valor al final de la lista se utiliza la operación `append()`. Escribimos `padrones.append(47890)` para agregar el padrón 47890 al final de `padrones`.

```
>>> padrones.append(47890)
>>> padrones
[78455, 79211, 66540, 45750, 47890]
```

- Para insertar un nuevo valor en la posición cuyo índice es `k` (y desplazar un lugar el resto de la lista) se utiliza la operación `insert()`.

  Escribimos `padrones.insert(2, 54988)` para insertar el padrón 54988 en la tercera posición de `padrones`.

```
>>> padrones.insert(2, 54988)
>>> padrones
[78455, 79211, 54988, 66540, 45750, 47890]
```

> Las listas no controlan si se insertan elementos repetidos. Si necesitamos exigir unicidad, debemos hacerlo mediante el código de nuestros programas.

```
>>> padrones.insert(1,78455)
>>> padrones
[78455, 78455, 79211, 54988, 66540, 45750, 47890]
```

- Para eliminar un valor de una lista se utiliza la operación `remove()`.

  Escribimos `padrones.remove(45750)` para borrar el padrón 45750 de la lista de inscriptos:

```
>>> padrones.remove(45750)
>>> padrones
[78455, 78455, 79211, 54988, 66540, 47890]
```

  Si el valor a borrar está repetido, se borra sólo su primera aparición:

```
>>> padrones.remove(78455)
>>> padrones
[78455, 79211, 54988, 66540, 47890]
```

Atención

Si el valor a borrar no existe, se produce un error:

```
>>> padrones.remove(78)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

7.2.3 Cómo buscar dentro de las listas

Queremos poder formular dos preguntas más respecto de la lista de inscriptos:

- ¿Está la persona cuyo padrón es v inscripta en esta materia?
- ¿En qué orden se inscribió la persona cuyo padrón es v?.

Veamos qué operaciones sobre listas se pueden usar para lograr esos dos objetivos:

- Para preguntar si un valor determinado es un elemento de una lista usaremos la operación `in`:

```
>>> padrones
[78455, 79211, 54988, 66540, 47890]
>>> 78 in padrones
False
>>> 66540 in padrones
True
```

El operador `in` se puede utilizar para todas las secuencias, incluyendo tuplas y cadenas.

- Para averiguar la posición de un valor dentro de una lista usaremos la operación `index()`.

```
>>> padrones.index(78455)
0
>>> padrones.index(47890)
4
```

Atención

Si el valor no se encuentra en la lista, se producirá un error:

```
>>> padrones.index(78)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

Si el valor está repetido, el índice que devuelve es el de la primera aparición:

```
>>> [10, 20, 10].index(10)
0
```

La función `index` también se puede utilizar con cadenas y tuplas.

- Para iterar sobre todos los elementos de una lista usaremos una construcción `for`:

```
>>> for p in padrones:
...     print(p)
...
78455
79211
54988
66540
47890
```

El ciclo `for <variable> in <secuencia>:` se puede utilizar sobre cualquier secuencia, incluyendo tuplas y cadenas.

- Muchas veces, dentro del cuerpo del ciclo `for` es necesario contar con la posición de cada elemento de la lista. Para esto es posible utilizar la función `enumerate`:

```
>>> for i, p in enumerate(padrones):
...     print(i, p)
...
0 78455
1 79211
2 54988
3 66540
4 47890
```

Problema 7.1. Queremos escribir un programa que nos permita armar la lista de los inscriptos de una materia.

1. **Análisis:** El usuario ingresa datos de padrones que se van guardando en una lista.
2. **Especificación:** El programa solicitará al usuario que ingrese uno a uno los padrones de los inscriptos. Con esos números construirá una lista, que al final se mostrará.
3. **Diseño:**
   - ¿Qué estructura tiene este programa? ¿Se parece a algo conocido?
     Es claramente un ciclo en el cual se le pide al usuario que ingrese uno a uno los padrones de los inscriptos, y estos números se agregan a una lista. Y en algún momento, cuando se terminaron los inscriptos, el usuario deja de cargar.
   - ¿El ciclo es definido o indefinido?
     Para que fuera un ciclo definido deberíamos contar de antemano cuántos inscriptos tenemos, y luego cargar exactamente esa cantidad, pero eso no parece muy útil.
     Estamos frente a una situación parecida al problema de la lectura de los números, en el sentido de que no sabemos cuántos elementos queremos cargar de antemano. Para ese problema, en la sección 5.3, vimos una solución muy sencilla y cómoda: se le piden datos al usuario y, cuando se cargaron todos los datos se ingresa un valor arbitrario (que se usa sólo para indicar que no hay más información). A ese diseño lo hemos llamado ciclo con centinela y tiene el siguiente esquema:

Sabías que…

En Python, las listas, las tuplas y las cadenas son parte del conjunto de las *secuencias*. Todas las secuencias cuentan con las siguientes operaciones:

Operación	Resultado
`x in s`	Indica si el valor `x` se encuentra en `s`
`s + t`	Concantena las secuencias `s` y `t`
`s * n`	Concatena `n` copias de `s`
`s[i]`	Elemento `i` de `s`, empezando por 0
`s[i:j]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive)
`s[i:j:k]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive), con paso `k`
`len(s)`	Cantidad de elementos de la secuencia `s`
`min(s)`	Mínimo elemento de la secuencia `s`
`max(s)`	Máximo elemento de la secuencia `s`
`sum(s)`	Suma de los elementos de la secuencia `s`
`enumerate(s)`	Enumerar los elementos de `s` junto con sus posiciones

Además, es posible crear una lista o una tupla a partir de cualquier otra secuencia, utilizando las funciones `list` y `tuple`, respectivamente:

```
>>> list("Hola")
['H', 'o', 'l', 'a']
>>> tuple("Hola")
('H', 'o', 'l', 'a')
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]
```

```
Repetir indefinidamente:
    Pedir datos
    Si el dato pedido coincide con el centinela:
        Salir del ciclo
    Realizar cálculos
```

Como sabemos que los números de padrón son siempre enteros positivos, podemos considerar que el centinela puede ser cualquier número menor o igual a cero. También sabemos que en nuestro caso tenemos que ir armando una lista que inicialmente no tiene ningún inscripto.

Modificamos el esquema anterior para ajustarnos a nuestra situación:

```
La lista de inscriptos es vacía
Repetir indefinidamente:
    Pedir padrón
    Si el padrón no es positivo:
        Salir del ciclo
    Agregar el padrón a la lista
Devolver la lista de inscriptos
```

4. **Implementación:** De acuerdo a lo diseñado, el programa quedaría como se muestra en el Código 7.1.

> Para entender mejor la implementación propuesta, es una buena idea repasar los conceptos de ciclos, en especial las instrucciones `break` y `continue`, explicadas en la sección 5.6.

Código 7.1 `inscripcion.py`**:** Permite ingresar padrones de alumnos inscriptos

```
def inscribir_alumnos():
    """Permite inscribir alumnos al curso"""

    print("Inscripcion en el curso de Algoritmos y Programación I")
    inscriptos = []
    while True:
        padron = int(input("Ingresa un padrón (<=0 para terminar): "))
        if padron <= 0:
            break
        inscriptos.append(padron)
    return inscriptos

inscriptos = inscribir_alumnos()
print("La lista de inscriptos es:", inscriptos)
```

5. **Prueba:** Para probarlo lo ejecutamos con algunos lotes de prueba (inscripción de tres alumnos, inscripción de cero alumnos, inscripción de alumnos repetidos):

```
$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: []

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 40, 30, 50]
```

   Evidentemente el programa funciona de acuerdo a lo especificado, pero hay algo que no tuvimos en cuenta: permite inscribir a una misma persona más de una vez.

6. **Mantenimiento:** No permitir que haya padrones repetidos.

7. **Diseño revisado:** Para no permitir que haya padrones repetidos debemos revisar que no exista el padrón antes de agregarlo en la lista:

```
La lista de inscriptos es vacía
Repetir indefinidamente:
    Pedir padrón
    Si el padrón no es positivo:
        Salir del ciclo
    Si el padrón está en la lista:
        Avisar que el padrón ya está en la lista
        Saltear a la siguiente iteración del ciclo
    Agregar el padrón a la lista
Devolver la lista de inscriptos
```

8. **Nueva implementación:** De acuerdo a lo diseñado en el párrafo anterior, el programa ahora quedaría como se muestra en el Código 7.2.

Código 7.2 `inscripcion.py`**:** Permite ingresar padrones, sin repetir

```
1  def inscribir_alumnos():
2      """Permite inscribir alumnos al curso"""
3
4      print("Inscripcion en el curso de Algoritmos y Programación I")
5      inscriptos = []
6      while True:
7          padron = int(input("Ingresa un padrón (<=0 para terminar): "))
8          if padron <= 0:
9              break
10         if padron in inscriptos:
11             print("El padrón ya está en la lista de inscriptos.")
12             continue
13         inscriptos.append(padron)
14     return inscriptos
15
16 inscriptos = inscribir_alumnos()
17 print("La lista de inscriptos es:", inscriptos)
```

9. **Nueva prueba:** Para probarlo lo ejecutamos con los mismos lotes de prueba anteriores (inscripción de tres alumnos, inscripción de cero alumnos, inscripción de alumnos repetidos):

```
$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: []

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
```

```
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 40
El padrón ya está en la lista de inscriptos.
Ingresa un padrón (<=0 para terminar): 30
El padrón ya está en la lista de inscriptos.
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]
```

Ahora el resultado es satisfactorio: no tenemos inscriptos repetidos.

Ejercicio 7.7. Permitir que los alumnos se puedan inscribir o borrar.

Ejercicio 7.8. Inscribir y borrar alumnos como antes, pero registrar también el nombre y apellido de la persona inscripta, de modo de tener como lista de inscriptos: `[(20, "Ana", "García"), (10, "Juan", "Salas")]`.

7.3 Ordenar listas

Nos puede interesar que los elementos de una lista estén ordenados: una vez que finalizó la inscripción en un curso, tener a los padrones de los alumnos por orden de inscripción puede ser muy incómodo, siempre será preferible tenerlos ordenados por número para realizar cualquier comprobación.

Python provee dos operaciones para obtener una lista ordenada a partir de una lista desordenada.

- Para dejar la lista original intacta pero obtener una nueva lista ordenada a partir de ella, se usa la función `sorted`.

```
>>> bs = [5, 2, 4, 2]
>>> cs = sorted(bs)
>>> bs
[5, 2, 4, 2]
>>> cs
[2, 2, 4, 5]
```

- Para modificar directamente la lista original usaremos la operación `sort()`.

```
>>> ds = [5, 3, 4, 5]
>>> ds.sort()
>>> ds
[3, 4, 5, 5]
```

7.4 Listas y cadenas

A partir de una cadena de caracteres, podemos obtener una lista con sus componentes usando la función `split`.

Si queremos obtener las palabras (separadas entre sí por espacios) que componen la cadena `padrones` escribiremos simplemente `padrones.split()`:

```
>>> c = "   Una    cadena   con    espacios    "
>>> c.split()
['Una', 'cadena', 'con', 'espacios']
```

En este caso `split` elimina todos los blancos de más, y devuelve sólo las palabras que conforman la cadena.

Si en cambio el separador es otro carácter (por ejemplo la arroba, `"@"`), se lo debemos pasar como parámetro a la función `split`. En ese caso se considera una componente todo lo que se encuentra entre dos arrobas consecutivas. En el caso particular de que el texto contenga dos arrobas una a continuación de la otra, se devolverá una componente vacía:

```
>>> d="@@Una@@@cadena@@@con@@arrobas@"
>>> d.split("@")
['', '', 'Una', '', '', 'cadena', '', '', 'con', '', 'arrobas', '']
```

La "casi"–inversa de `split` es una función `join` que tiene la siguiente sintaxis:

```
<separador>.join(<lista de componentes a unir>)
```

y que devuelve la cadena que resulta de unir todas las componentes separadas entre sí por medio del *separador*:

```
>>> xs = ['aaa', 'bbb', 'cccc']
>>> " ".join(xs)
'aaa bbb cccc'
>>> ", ".join(xs)
'aaa, bbb, cccc'
>>> "@@".join(xs)
'aaa@@bbb@@cccc'
```

7.4.1 Ejercicios con listas y cadenas

Ejercicio 7.9. Escribir una función que reciba como parámetro una cadena de palabras separadas por espacios y devuelva, como resultado, cuántas palabras de más de cinco letras tiene la cadena dada.

Ejercicio 7.10. Procesamiento de telegramas. Un oficial de correos decide optimizar el trabajo de su oficina cortando todas las palabras de más de cinco letras a sólo cinco letras (e indicando que una palabra fue cortada con el agregado de una arroba). Además elimina todos los espacios en blanco de más.

Por ejemplo, al texto `"  Llego  mañana  alrededor  del  mediodía  "` se transcribe como `"Llego mañan@ alred@ del medio@"`.

Por otro lado cobra un valor para las palabras cortas y otro valor para las palabras largas (que deben ser cortadas).

a) Escribir una función que reciba un texto, la longitud máxima de las palabras, el costo de cada palabra corta, el costo de cada palabra larga, y devuelva como resultado el texto del telegrama y el costo del mismo.

b) Los puntos se reemplazan por la palabra especial "STOP", y el punto final (que puede faltar en el texto original) se indica como "STOPSTOP".

Al texto:

`"  Llego  mañana  alrededor  del  mediodía. Voy a almorzar "`

Se lo transcribe como:

`"Llego mañan@ alred@ del medio@ STOP Voy a almor@ STOPSTOP"`.

Extender la función anterior para agregar el tratamiento de los puntos.

7.5 Resumen

- Python nos provee con varias estructuras que nos permiten agrupar los datos que tenemos. En particular, las **tuplas** son estructuras **inmutables** que permiten agrupar valores al momento de crearlas, y las **listas** son estructuras **mutables** que permiten agrupar valores, con la posibilidad de agregar, quitar o reemplazar sus elementos.
- Las tuplas se utilizan para modelar situaciones en las cuales al momento de crearlas ya se sabe cuál va a ser la información a almacenar. Por ejemplo, para representar una fecha, una carta de la baraja, una ficha de dominó.
- Las listas se utilizan en las situaciones en las que los elementos a agrupar pueden ir variando a lo largo del tiempo. Por ejemplo, para representar un las notas de un alumno en diversas materias, los inscriptos para un evento o la clasificación de los equipos en una competencia.
- Las cadenas, tuplas y listas son tres tipos diferentes de **secuencias**. Las secuencias ofrecen un conjunto de operaciones básicas, como obtener la longitud y recorrer sus elementos, que se aplican de la misma manera sin importar qué tipo de secuencia es.

Referencia Python

`(valor1, valor2, valor3)`

Las tuplas se definen como una sucesión de valores encerrados entre paréntesis y separados por comas. Una vez definidas, no se pueden modificar los valores asignados.

Casos particulares:

```
tupla_vacia = ()
tupla_unitaria = (3459,)
```

`[valor1, valor2, valor3]`

Las listas se definen como una sucesión de valores encerrados entre corchetes y separados por comas. Se les puede agregar, quitar o cambiar los valores que contienen.

```
lista = [1, 2, 3]
lista[0] = 5
```

Caso particular:

```
lista_vacia = []
```

`x, y, z = secuencia`

Es posible *desempaquetar* una secuencia, asignando a la izquierda tantas variables como elementos tenga la secuencia. Cada variable tomará el valor del elemento que se encuentra en la misma posición.

`len(secuencia)`

Devuelve la cantidad de elementos que contiene la secuencia, 0 si está vacía.

`for elemento in secuencia:`

Itera uno a uno por los elementos de la secuencia.

`elemento in secuencia`

Indica si el elemento se encuentra o no en la secuencia

`secuencia[i]`

Corresponde al valor de la secuencia en la posición `i`, comenzando desde 0.

Si se utilizan números negativos, se puede acceder a los elementos desde el último (`-1`) hasta el primero (`-len(secuencia)`).

En el caso de las tuplas o cadenas (inmutables) sólo puede usarse para obtener el valor, mientra que en las listas (mutables) puede usarse también para modificar su valor.

`secuencia[i:j:k]`

Permite obtener un segmento de la secuencia, desde la posición `i` inclusive, hasta la posición `j` exclusive, con paso `k`.

En el caso de que se omita `i`, se asume `0`. En el caso de que se omita `j`, se asume `len(secuencia)`. En el caso de que se omita `k`, se asume 1. Si se omiten todos, se obtiene una copia completa de la secuencia.

`lista.append(valor)`

Agrega un elemento al final de la lista.

`lista.insert(posicion, valor)`

Agrega un elemento a la lista, en la posición `posicion`.

`lista.remove(valor)`

Quita de la lista la primera aparción de elemento, si se encuentra. De no encontrarse en la lista, se produce un error.

`lista.pop()`

Quita el elemento del final de la lista, y lo devuelve. Si la lista está vacía, se produce un error.

`lista.pop(posicion)`

Quita el elemento que está en la posición indicada, y lo devuelve. Si la lista tiene menos de `posicion + 1` elementos, se produce un error.

`lista.index(valor)`

Devuelve la posición de la primera aparición de valor. Si no se encuentra en la lista, se produce un error.

`sorted(secuencia)`

Devuelve una lista nueva, con los elementos de la secuencia ordenados.

`lista.sort()`

Ordena la misma lista.

`cadena.split(separador)`

Devuelve una lista con los elementos de cadena, utilizando `separador` como separador de elementos.

Si se omite el separador, toma todos los espacios en blanco como separadores.

`separador.join(lista)`

Genera una cadena a partir de los elementos de `lista`, utilizando `separador` como unión entre cada elemento y el siguiente.

7.6 Ejercicios

Ejercicio 7.6.1. Escribir una función que reciba una tupla de elementos e indique si se encuentran ordenados de menor a mayor o no.

Ejercicio 7.6.2. Dominó.

a) Escribir una función que indique si dos fichas de dominó *encajan* o no. Las fichas son recibidas en dos tuplas, por ejemplo: `(3,4)` y `(5,4)`

b) Escribir una función que indique si dos fichas de dominó *encajan* o no. Las fichas son recibidas en una cadena, por ejemplo: `3-4 2-5`. **Nota:** utilizar la función `split` de las cadenas.

Ejercicio 7.6.3. Campaña electoral

a) Escribir una función que reciba una tupla con nombres, y para cada nombre imprima el mensaje *Estimado <nombre>, vote por mí.*

b) Escribir una función que reciba una tupla con nombres, una posición de origen `p` y una cantidad `n`, e imprima el mensaje anterior para los `n` nombres que se encuentran a partir de la posición `p`.

c) Modificar las funciones anteriores para que tengan en cuenta el género del destinatario, para ello, deberán recibir una tupla de tuplas, conteniendo el nombre y el género.

Ejercicio 7.6.4. Vectores

a) Escribir una función que reciba dos vectores y devuelva su producto escalar.

b) Escribir una función que reciba dos vectores y devuelva si son o no ortogonales.

c) Escribir una función que reciba dos vectores y devuelva si son paralelos o no.

d) Escribir una función que reciba un vector y devuelva su norma.

Ejercicio 7.6.5. Dada una lista de números enteros, escribir una función que:

a) Devuelva una lista con todos los que sean primos.

b) Devuelva la sumatoria y el promedio de los valores.

c) Devuelva una lista con el factorial de cada uno de esos números.

Ejercicio 7.6.6. Dada una lista de números enteros y un entero k, escribir una función que:

a) Devuelva tres listas, una con los menores, otra con los mayores y otra con los iguales a k.

b) Devuelva una lista con aquellos que son múltiplos de k.

Ejercicio 7.6.7. Escribir una función que reciba una lista de tuplas (Apellido, Nombre, Inicial_segundo_nombre) y devuelva una lista de cadenas donde cada una contenga primero el nombre, luego la inicial con un punto, y luego el apellido.

Ejercicio 7.6.8. Inversión de listas

a) Realizar una función que, dada una lista, devuelva una nueva lista cuyo contenido sea igual a la original pero invertida. Así, dada la lista `['Di', 'buen', 'día', 'a', 'papa']`, deberá devolver `['papa', 'a', 'día', 'buen', 'Di']`.

b) Realizar otra función que invierta la lista, pero en lugar de devolver una nueva, modifique la lista dada para invertirla, **sin** usar listas auxiliares.

Ejercicio 7.6.9. Escribir una función `empaquetar` para una lista, donde epaquetar significa indicar la repetición de valores consecutivos mediante una tupla (valor, cantidad de repeticiones). Por ejemplo, `empaquetar([1, 1, 1, 3, 5, 1, 1, 3, 3])` debe devolver `[(1, 3), (3, 1), (5, 1), (1, 2), (3, 2)]`.

Ejercicio 7.6.10. Matrices.

a) Escribir una función que reciba dos matrices y devuelva la suma.

b) Escribir una función que reciba dos matrices y devuelva el producto.

c) ⋆ Escribir una función que opere sobre una matriz y mediante *eliminación gaussiana* devuelva una matriz triangular superior.

d) ⋆ Escribir una función que indique si un grupo de vectores, recibidos mediante una lista, son linealmente independientes o no.

Ejercicio 7.6.11. ⋆ **Plegado de un texto.** Escribir una función que reciba un texto y una longitud y devuelva una lista de cadenas de como máximo esa longitud. Las líneas deben ser cortadas correctamente en los espacios (sin cortar las palabras).

Ejercicio 7.6.12. Funciones que reciben funciones.

a) Escribir una funcion llamada **map**, que reciba una función y una lista y devuelva la lista que resulta de aplicar la función recibida a cada uno de los elementos de la lista recibida.

b) Escribir una función llamada **filter**, que reciba una función y una lista y devuelva una lista con los elementos de la lista recibida para los cuales la función recibida devuelve un valor verdadero.

c) ¿En qué ejercicios de esta guía podría haber utilizado estas funciones?

Apéndice 7.A Funciones de orden superior

En Python, se dice que *las funciones son ciudadanos de primera clase*, ya que una función es un valor (cuyo tipo de dato es `function`) que puede ser manipulado como cualquier otro valor (`int`, `str`, etc.):

```
>>> def exclamar(s):
...     return '¡' + s + '!'
>>> exclamar
<function exclamar at 0x10532ef28>
>>> type(exclamar)
<class 'function'>
>>> exclamar('Hola')
'¡Hola!'
>>> f = exclamar
>>> f('Hola')
'¡Hola!'
```

En el ejemplo, `exclamar` (sin los paréntesis) es una referencia a un valor de tipo `function` (la función llamada "exclamar"). Al hacer `f = exclamar`, el resultado es que ambas variables (`exclamar` y `f`) hacen referencia a *la misma* función. Por lo tanto, podemos llamar a la función con `f(...)` o con `exclamar(...)`.

Aprovechando esto, una función puede recibir otra función como parámetro:

```
>>> def aplicar(f, x, n):
...     """Devuelve el resultado de f(f(...f(f(x)))), aplicando en total n
...        veces la función f"""
...     for i in range(n):
...         x = f(x)
...     return x
>>> aplicar(exclamar, 'Hola', 5)
'¡¡¡¡¡Hola!!!!!'
>>> def alargar(s):
...     return s + s[-1]
>>> alargar('Hola')
'Holaa'
>>> aplicar(alargar, 'Hola', 5)
'Holaaaaaa'
```

Una función también puede devolver otra función:

```
>>> def puntuador(inicio, fin):
...     """Recibe dos cadenas de inicio y fin, y devuelve una función
...        que permite agregar signos de puntuación a una cadena."""
...     def puntuar(cadena):
...         return inicio + cadena + fin
...     return puntuar
>>> exclamar = puntuador('¡', '!')
>>> type(exclamar)
<class 'function'>
>>> exclamar('Hola')
'¡Hola!'
>>> preguntar = puntuador('¿', '?')
>>> preguntar('Qué')
'¿Qué?'
```

Se dice que una función es *de orden superior* cuando recibe una función por parámetro, o bien cuando devuelve una función.

En nuestros ejemplos, `aplicar` y `puntuador` son de orden superior. Las funciones de orden superior son características de un estilo de programación llamado *programación funcional*.

Funciones anónimas

Cuando trabajamos con funciones de orden superior, suele ser conveniente definir *funciones anónimas*. En Python, cualquier función que evalúa una única expresión y la devuelve puede ser definida en forma anónima usando la sintaxis `lambda`:

- `lambda x: x * 2` es "una función que dado un número x devuelve $2x$".
- `lambda x, y: x + y` es "una función que dados x e y devuelve la suma de ambos".

La sintaxis general es `lambda <parámetros>: <expresión>`. El valor de retorno de una función definida con `lambda` es el resultado de la `<expresión>`; el `return` es implícito.

Sabías que...

La sintaxis `lambda` hace referencia al *cálculo lambda* (un sistema formal introducido por Alonzo Church y Stephen Kleene en la década de 1930), en el que las funciones se escriben anteponiendo la letra griega λ. Por ejemplo, una función que eleva un número al cuadrado, en notación lambda se escribe $\lambda x.x^2$, mientras que en Python se escribe `lambda x: x ** 2`.

Las funciones `exclamar` y `alargar` podrían haber sido definidas usando la sintaxis `lambda`:

```
>>> exclamar = lambda s: 'i' + s + '!'
>>> type(exclamar)
<class 'function'>
>>> exclamar('Hola')
'¡Hola!'
>>> alargar = lambda s: s + s[-1]
>>> alargar('Hola')
'Holaa'
```

Pero las funciones anónimas son especialmente útiles cuando trabajamos con funciones de orden superior. Por ejemplo, podemos pasar un `lambda` a una función que recibe una función:

```
>>> aplicar(lambda x: 2 * x, 1, 4)
16[1]
```

Una función que devuelve una función también puede devolver un `lambda`:

```
>>> def puntuador(inicio, fin):
...     """Recibe dos cadenas de inicio y fin, y devuelve una función
...        que permite agregar signos de puntuación a una cadena."""
...     return lambda cadena: inicio + cadena + fin
>>> citar = puntuador('“', '”')
>>> citar('Del dicho al hecho hay mucho trecho')
```

[1] Ya que $2 \cdot (2 \cdot (2 \cdot (2 \cdot 1))) = 16$.

```
'“Del dicho al hecho hay mucho trecho”'
>>> def componer(f, g):
...     """Devuelve una función que dado x devuelve g(f(x))"""
...     return lambda x: g(f(x))
>>> gritar = componer(alargar, exclamar)
>>> gritar('Hola')
'¡Holaa!'
```

Funciones de orden superior y secuencias

Las funciones `sorted`, `map` y `filter` son ejemplos de funciones de orden superior disponibles en Python, que además permiten operar sobre secuencias.

- La función `sorted` ya fue mencionada en la Sección 7.3. Si recibe una lista de cadenas, devuelve una nueva lista ordenada:

  ```
  >>> L = ['zorro', '', '935', 'alpaca', '1312', '-----', 'gato']
  >>> sorted(L)
  ['', '-----', '1312', '935', 'alpaca', 'gato', 'zorro']
  ```

  El comportamiento por omisión es ordenar las cadenas en orden *lexicográfico* (comparando el primer caracter, luego el segundo, etc.). Este comportamiento es útil en la gran mayoría de los casos, pero ¿qué pasa si necesitamos ordenar las cadenas por otro criterio?

  Podemos pasarle a `sorted` un parámetro adicional, que es una función que determina para cada elemento el valor utilizado para ordenar. Por ejemplo, si queremos ordenar las cadenas según su longitud (de menor a mayor), tenemos que pasarle a `sorted` una función que recibe una cadena y devuelve su longitud. Esa función no es otra que la función `len`:

  ```
  >>> sorted(L, key=len)[2]
  ['', '935', '1312', 'gato', 'zorro', '-----', 'alpaca']
  ```

- La función `map` recibe una función y una secuencia, y devuelve la secuencia resultante de aplicar la función a cada uno de los elementos de la secuencia original.

  ```
  >>> map(lambda x: x * 2, [1, 2, 3, 4])
  <map object at 0x10edcba20>
  ```

  El valor de retorno de `map` es una secuencia, pero no es una tupla ni una lista. Por suerte, podemos convertir cualquier secuencia en una lista con la función `list`:

  ```
  >>> list(map(lambda x: x * 2, [1, 2, 3, 4, 5]))
  [2, 4, 6, 8, 10]
  >>> list(map(exclamar, ['Usando', 'la', 'función', 'map']))
  ['¡Usando!', '¡la!', '¡función!', '¡map!']
  >>> list(map(int, [8.3, '42', True]))
  [8, 42, 1]
  ```

- La función `filter` recibe una función `f` y una secuencia, y devuelve una secuencia que contiene los elementos de la secuencia original para los cuales el resultado de `f(x)` es `True` .

[2] `key=len` indica que el parámetro adicional es un *parámetro con nombre* (en inglés *keyword argument*): el nombre del parámetro es `key` y el valor es `len`.

```
>>> list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5]))
[2, 4]
>>> list(filter(lambda s: len(s) > 2, ['Usando', 'la', 'función', 'filter']))
['Usando', 'función', 'filter']
```

Unidad 8

Más sobre listas y secuencias

8.1 Búsqueda lineal

El problema de la búsqueda

Presentamos ahora uno de los problemas más clásicos de la computación, *el problema de la búsqueda*, que se puede enunciar de la siguiente manera:

Problema: Dada una lista xs y un valor x devolver el índice de x en xs si x está en xs, y -1 si x no está en xs.

Alicia Hacker afirma que este problema tiene una solución muy sencilla en Python: se puede usar directamente la poderosa función `index()` de lista.

Probamos esa solución para ver qué pasa:

```
>>> [1, 3, 5, 7].index(5)
2
>>> [1, 3, 5, 7].index(20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

Vemos que usar la función `index()` resuelve nuestro problema si el valor buscado está en la lista, pero si el valor no está no sólo no devuelve un -1, sino que se produce un error.

El problema es que para poder aplicar la función `index()` debemos estar seguros de que el valor está en la lista, y para averiguar eso Python nos provee del operador `in`:

```
>>> 5 in [1, 3, 5, 7]
True
>>> 20 in [1, 3, 5, 7]
False
```

O sea que si llamamos a la función `index()` sólo cuando el resultado de `in` es verdadero, y devolvemos -1 cuando el resultado de `in` es falso, estaremos resolviendo el problema planteado usando sólo funciones provistas por Python. La solución se plantea a continuación:

```
def busqueda_con_index(xs, x):
    """Busca un elemento x en una lista xs.

    Si x está en xs devuelve el índice,
    de lo contrario devuelve -1.
    """
    if x in xs:
```

```
        return xs.index(x)
    else:
        return -1
```

Probamos la función `busqueda_con_index()`:

```
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 1)
0
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], -1)
5
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 3)
3
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 44)
-1
>>> busqueda_con_index([], 0)
-1
```

¿Cuántas comparaciones hace este programa?

Es decir, ¿cuánto esfuerzo computacional requiere este programa? ¿Cuántas veces compara el valor que buscamos con los datos de la lista? No lo sabemos porque no sabemos cómo están implementadas las operaciones `in` e `index()`. La pregunta queda planteada por ahora pero daremos un método para averiguarlo más adelante en esta unidad.

Búsqueda lineal

Nos interesa ver qué sucede si programamos la búsqueda usando operaciones más elementales, y no las grandes primitivas `in` e `index()`. Esto nos permitirá estudiar una solución que puede portarse a otros lenguajes que no tienen instrucciones tan poderosas.

Supongamos entonces que en nuestra versión de Python no existen ni `in` ni `index()`. Podemos en cambio acceder a cada uno de los elementos de la lista a través de una construcción `for`, y también, por supuesto, podemos acceder a un elemento de la lista mediante un índice.

Diseñamos una solución: Podemos comparar uno a uno los elementos de la lista con el valor de `x`, y retornar el valor de la posición donde lo encontramos en caso de encontrarlo.

Si llegamos al final de la lista sin haber salido antes de la función es porque el valor de `x` no está en la lista, y en ese caso retornamos -1.

En esta solución necesitamos una variable `i` que cuente en cada momento en qué posición de la lista estamos parados. Esta variable se inicializa en 0 antes de entrar en el ciclo y se incrementa en 1 en cada paso.

El programa nos queda entonces como se muestra a continuación:

```
def busqueda_lineal(lista, x):
    """Si x está en lista devuelve su posición en lista, de lo
    contrario devuelve -1.
    """

    i = 0
    for z in lista: ❶
        if z == x: ❷
            return i ❸
        i += 1
    return -1
```

Y ahora lo probamos:

```
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 44)
-1
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 3)
3
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 0)
4
>>> busqueda_lineal([], 42)
-1
```

¿Cuántas comparaciones hace este programa?

Volvemos a preguntarnos lo mismo que en la sección anterior, pero con el nuevo programa: ¿cuánto esfuerzo computacional requiere este programa?, ¿cuántas veces compara el valor que buscamos con los datos de la lista? Ahora podemos analizar el código de `busqueda_lineal`:

- La línea ❶ es un ciclo que recorre uno a uno los elementos de la lista, y en el cuerpo de ese ciclo, en ❷ se compara cada elemento con el valor buscado. En el caso de encontrarlo (❸) se devuelve la posición.
- Si el valor no está en la lista se recorrerá la lista entera, haciendo una comparación por cada elemento.

O sea que si el valor está en la posición p de la lista se hacen p comparaciones, y si el valor no está se hacen tantas comparaciones como elementos tenga la lista.

Nuestra hipótesis es: **Si la lista crece, la cantidad de comparaciones para encontrar un valor arbitrario crecerá en forma proporcional al tamaño de la lista**. Por lo tanto diremos que:

> El algoritmo de búsqueda lineal tiene un comportamiento *proporcional a la longitud de la lista involucrada*, o que es un algoritmo *lineal*.

En la próxima sección veremos cómo probar esta hipótesis.

8.2 Búsqueda sobre listas ordenadas

Si podemos suponer que la lista está previamente ordenada, ¿podemos encontrar una manera más eficiente de buscar elementos sobre ella?

En principio hay una modificación muy simple que podemos hacer sobre el algoritmo de búsqueda lineal: si estamos buscando el elemento x en una lista que está ordenada de menor a mayor, en cuanto encontremos algún elemento mayor a x podemos estar seguros de que x no está en la lista, por lo que no es necesario continuar recorriendo el resto.

Ejercicio 8.1. Modificar la búsqueda lineal para el caso de listas ordenadas. En el peor caso, ¿cuál es nuestra nueva hipótesis sobre comportamiento del algoritmo? ¿Es realmente más eficiente?

Búsqueda binaria

¿Podemos hacer algo mejor? Trataremos de aprovechar el hecho de que la lista está ordenada y vamos a hacer algo distinto: nuestro espacio de búsqueda se irá achicando a segmentos cada vez menores de la lista original. La idea es descartar segmentos de la lista donde el valor seguro que no puede estar:

1. Consideramos como segmento inicial de búsqueda a la lista completa.
2. Analizamos el punto medio del segmento (el valor central); si es el valor buscado, devolvemos el índice del punto medio.
3. Si el valor central es mayor al buscado, podemos descartar el segmento que está desde el punto medio hacia la a derecha.
4. Si el valor central es menor al buscado, podemos descartar el segmento que está desde el punto medio hacia la izquierda.
5. Una vez descartado el segmento que no nos interesa, volvemos a analizar el segmento restante, de la misma forma.
6. Si en algún momento el segmento a analizar tiene longitud 0 o negativa significa que el valor buscado no se encuentra en la lista.

Para señalar la porción del segmento que se está analizando a cada paso, utilizaremos dos variables (`izq` y `der`) que contienen la posición de inicio y la posición de fin del segmento que se está considerando. De la misma manera usaremos la varible `medio` para contener la posición del punto medio del segmento.

En la Figura 8.1 vemos qué pasa cuando se busca el valor 18 en la lista [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23].

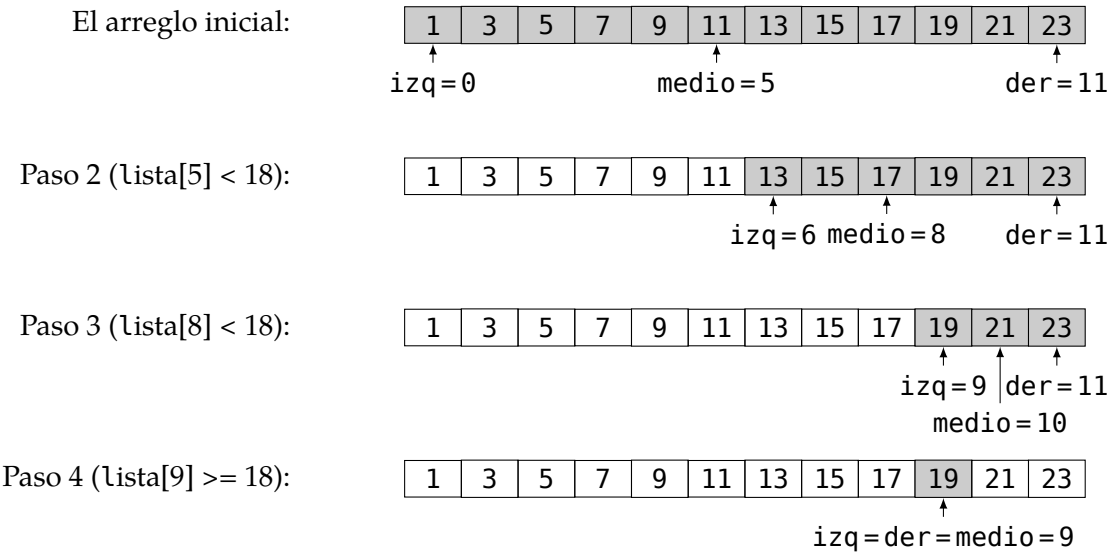


Figura 8.1: Ejemplo de una búsqueda usando el algoritmo de búsqueda binaria. Como no se encontró al valor buscado, devuelve -1.

En el Código 8.1 mostramos una posible implementación de este algoritmo.
A continuación varias ejecuciones de prueba:

```
>>> busqueda_binaria([1, 3, 5], 0)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 0 der: 0 medio: 0
-1
```

Código 8.1 `busqueda_binaria.py`**:** Función de búsqueda binaria

```
def busqueda_binaria(lista, x):
    """Búsqueda binaria

    Precondición: la lista está ordenada
    Devuelve -1 si x no está en lista;
    Devuelve p tal que lista[p] == x, si x está en lista
    """

    izq = 0
    der = len(lista) - 1

    while izq <= der:
        medio = (izq + der) // 2
        print("[DEBUG]", "izq:", izq, "der:", der, "medio:", medio)

        if lista[medio] == x:
            # Encontramos el elemento; devolvemos su posición
            return medio

        if lista[medio] > x:
            # Seguimos buscando en el segmento de la izquierda
            der = medio - 1
        else:
            # Seguimos buscando en el segmento de la derecha
            izq = medio + 1

    # El valor no fue encontrado
    return -1
```

```
>>> busqueda_binaria([1, 3, 5], 1)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 0 der: 0 medio: 0
0
>>> busqueda_binaria([1, 3, 5], 2)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 0 der: 0 medio: 0
-1
>>> busqueda_binaria([1, 3, 5], 3)
[DEBUG] izq: 0 der: 2 medio: 1
1
>>> busqueda_binaria([1, 3, 5], 5)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 2 der: 2 medio: 2
2
>>> busqueda_binaria([1, 3, 5], 6)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 2 der: 2 medio: 2
-1
>>> busqueda_binaria([], 0)
-1
>>> busqueda_binaria([1], 1)
```

```
[DEBUG] izq: 0 der: 0 medio: 0
0
>>> busqueda_binaria([1], 3)
[DEBUG] izq: 0 der: 0 medio: 0
-1
```

Ejercicio 8.2. En la línea 13 de `busqueda_binaria.py` efectuamos la división usando el operador `//` en lugar de `/`. ¿Qué sucedería si utilizáramos `/`?

¿Cuántas comparaciones hace este programa?

Para responder esto pensemos en el peor caso, es decir, que se descartaron varias veces partes del segmento para finalmente llegar a un segmento vacío y el valor buscado no se encontraba en la lista.

En cada paso el segmento se divide por la mitad y se desecha una de esas mitades, y en cada paso se hace una comparación con el valor buscado. Por lo tanto, la cantidad de comparaciones que hacen con el valor buscado es aproximadamente igual a la cantidad de pasos necesarios para llegar a un segmento de tamaño 1. Veamos el caso más sencillo para razonar, y supongamos que la longitud de la lista es una potencia de 2, es decir `len(lista)` $= 2^k$:

- Luego del primer paso, el segmento a tratar es de tamaño 2^k.
- Luego del segundo paso, el segmento a tratar es de tamaño 2^{k-1}.
- Luego del tercer paso, el segmento a tratar es de tamaño 2^{k-2}.

  ...
- Luego del paso k, el segmento a tratar es de tamaño $2^{k-k} = 1$.

Por lo tanto este programa hace aproximadamente k comparaciones con el valor buscado cuando `len(lista)` $= 2^k$. Pero si despejamos k de la ecuación anterior, podemos ver que este programa realiza aproximadamente $\log_2($`len(lista)`$)$ comparaciones.

Cuando `len(lista)` no es una potencia de 2 el razonamiento es menos prolijo, pero también vale que este programa realiza aproximadamente $\log_2($`len(lista)`$)$ comparaciones. Concluimos entonces que:

> Si podemos suponer que la lista está previamente ordenada, podemos utilizar el algoritmo de búsqueda binaria, que es *muchísimo* más eficiente que la búsqueda lineal.

Veamos un ejemplo para entender cuánto más eficiente es la búsqueda binaria. Supongamos que tenemos una lista de un millón de elementos.

- El algoritmo de búsqueda lineal hará una cantidad de operaciones proporcional a un millón; es decir que en el peor caso hará 1.000.000 de comparaciones, y en un caso promedio, 500.000 comparaciones.
- El algoritmo de búsqueda binaria hará como máximo $\log_2(1\,000\,000)$ comparaciones, o sea ¡no más que 20 comparaciones!.

8.3 Resumen

- La **búsqueda** de un elemento en una secuencia es un algoritmo básico pero importante. El problema que intenta resolver puede plantearse de la siguiente manera: Dada una secuencia de valores y un valor, devolver el índice del valor en la secuencia, si se encuentra, de no encontrarse el valor en la secuencia señalizarlo apropiadamente.
- Una de las formas de resolver el problema es mediante la **búsqueda lineal**, que consiste en ir revisando uno a uno los elementos de la secuencia y comparándolos con el elemento a buscar. Este algoritmo no requiere que la secuencia se encuentre ordenada.
- Cuando la secuencia sobre la que se quiere buscar está ordenada, se puede utilizar el algoritmo de **búsqueda binaria**. Al estar ordenada la secuencia, se puede desacartar en cada paso la mitad de los elementos, quedando entonces con una eficiencia algorítmica relativa al $log($`len(secuencia)`$)$. Este algoritmo sólo tiene sentido utilizarlo sobre una secuencia ordenada.
- El análisis del comportamiento de un algoritmo puede ser muy engañoso si se tiene en cuenta el mejor caso, por eso suele ser mucho más ilustrativo tener en cuenta el **peor caso**. En algunos casos particulares podrá ser útil tener en cuenta, además, el **caso promedio**.

8.4 Ejercicios

Ejercicio 8.4.1. Escribir una función que reciba una lista desordenada y un elemento, que:

a) Busque todos los elementos coincidan con el pasado por parámetro y devuelva la cantidad de coincidencias encontradas.

b) Busque la primera coincidencia del elemento en la lista y devuelva su posición.

c) Utilizando la función anterior, busque todos los elementos que coincidan con el pasado por parámetro y devuelva una lista con las posiciones.

Ejercicio 8.4.2. Escribir una función que reciba una lista de números no ordenada, que:

a) Devuelva el valor máximo.

b) Devuelva una tupla que incluya el valor máximo y su posición.

c) ¿Qué sucede si los elementos son cadenas de caracteres?

Nota: no utilizar `lista.sort()`

Ejercicio 8.4.3. Agenda simplificada
Escribir una función que reciba una cadena a buscar y una lista de tuplas (nombre_completo, telefono), y busque dentro de la lista, todas las entradas que contengan en el nombre completo la cadena recibida (puede ser el nombre, el apellido o sólo una parte de cualquiera de ellos). Debe devolver una lista con todas las tuplas encontradas.

Ejercicio 8.4.4. Sistema de facturación simplificado
Se cuenta con una lista ordenada de productos, en la que uno consiste en una tupla de (identificador, descripción, precio), y una lista de los productos a facturar, en la que cada uno consiste en una tupla de (identificador, cantidad).
Se desea generar una factura que incluya la cantidad, la descripción, el precio unitario y el precio total de cada producto comprado, y al final imprima el total general.
Escribir una función que reciba ambas listas e imprima por pantalla la factura solicitada.

Ejercicio 8.4.5. Escribir una función que reciba una lista ordenada y un elemento. Si el elemento se encuentra en la lista, debe encontrar su posición mediante búsqueda binaria y devolverlo. Si no se encuentra, debe agregarlo a la lista en la posición correcta y devolver esa nueva posición. (No utilizar `lista.sort()`.)

Apéndice 8.A Filtros, transformaciones y acumulaciones

Supongamos que manejamos una librería, y disponemos de una base de datos con el inventario. Cada entrada del inventario está compuesta por el título del libro, el autor, la cantidad disponible y el precio. Por ejemplo:

Título	Autor	Cantidad	Precio
The Art of Computer Programming, Volumes 1-4	Donald Knuth	12	179.62
Concrete Mathematics: A Foundation for Computer Science	Donald Knuth	5	54.77
The Pragmatic Programmer: From Journeyman to Master	Andrew Hunt, David Thomas	3	33.17
Clean Code: A Handbook of Agile Software Craftsmanship	Robert C. Martin	7	38.99
Code Complete: A Practical Handbook of Software Construction	Steve McConnell	0	29.97
Learning Python	Mark Lutz	4	40.95
…	…	…	…

Podemos representar nuestro inventario en Python utilizando una lista de tuplas:

```
inventario = [
    ('The Art of Computer Programming, Volumes 1-4',
     'Donald Knuth', 12, 179.62),
    ('Concrete Mathematics: A Foundation for Computer Science',
     'Donald Knuth', 5, 54.77),
    ('The Pragmatic Programmer: From Journeyman to Master',
     'Andrew Hunt and David Thomas', 3, 33.17),
    ...
]
```

Una vez que disponemos de nuestro inventario en una estructura de datos, podemos sacar todo tipo de reportes: la cantidad total de libros, el valor total del inventario, el precio promedio por libro, etc.

Veamos un ejemplo simple: supongamos que queremos obtener la cantidad total de libros de un autor determinado. Podemos diseñar un algoritmo muy simple:

```
def total_libros_autor(inventario, autor_buscado):
    total = 0
    for titulo, autor, cantidad, precio in inventario:
        if autor == autor_buscado:
            total += cantidad
    return total
```

Otro ejemplo: queremos obtener la cantidad de títulos de los cuales no hay suficiente stock (menos de 5 unidades):

```
def cantidad_poco_stock(inventario):
    total = 0
    for titulo, autor, cantidad, precio in inventario:
        if cantidad < 5:
            total += 1
    return total
```

¿Y si quisiéramos obtener la lista de títulos cuyo precio supera los $100?

```
def titulos_caros(inventario):
    titulos = []
    for titulo, autor, cantidad, precio in inventario:
        if precio > 100:
            titulos.append(titulo)
    return titulos
```

Acabamos de "inventar" tres algoritmos para resolver tres problemas diferentes… pero ¿son realmente diferentes? ¿No tienen nada en común?

Si observamos con detenimiento, los tres algoritmos comparten un mismo esquema:

```
def f(L):
    inicializar acumulador
    por cada elemento en el la lista L:
        si se cumple alguna condición:
            hacer algún cálculo en base al elemento y acumular
    devolver acumulador
```

Y este esquema en el fondo puede pensarse como una composición de tres problemas más simples:

1. A partir de una lista, *filtrar* la lista según una condición determinada y obtener una lista con los elementos que pasan la condición.
2. A partir de una lista, aplicar una *transformación* a cada elemento y obtener una lista con los resultados.
3. A partir de una lista, *acumular* los elementos según un criterio determinado.

Por ejemplo, podemos repensar nuestro primer algoritmo, que nos permitía calcular la cantidad total de libros de un determinado autor, como:

1. A partir del inventario, *filtrar* según el autor: el resultado será una lista que contiene los libros del autor buscado.
2. A partir de la lista obtenida, *transformar* cada una de las tuplas `(titulo, autor, cantidad, precio)` para descartar todo menos la `cantidad`. Es decir, nos quedamos con una lista de números enteros, donde cada número representa una cantidad de libros.
3. A partir de la lista obtenida, *acumular* los elementos sumando uno a uno.

Una ventaja de pensar el algoritmo de esta manera es que Python nos provee una forma muy fácil de implementar filtros y transformaciones: las *listas por comprensión*.

8.A.1 Listas por comprensión

Concentrémonos en el filtro según el autor propuesto en el ejemplo anterior. Una forma de implementarlo es:

```
def filtrar_autor(inventario, autor_buscado):
    filtrado = []
    for libro in inventario:
        if libro[2] == autor_buscado:
            filtrado.append(libro)
    return filtrado
```

En Python podemos obtener el mismo resultado utilizando una *lista por comprensión*:

```
def filtrar_autor(inventario, autor_buscado):
    return [libro for libro in inventario if libro[2] == autor_buscado]
```

Lo que vemos aquí es una sintaxis especial, que nos permite crear una lista filtrando una secuencia según una condición:

```
[<variable> for <variable> in <secuencia> if <condicion>]
```

Para aplicar la transformación propuesta (quedándonos únicamente con las cantidades), podríamos implementarlo de esta manera:

```
def obtener_cantidades(inventario):
    cantidades = []
    for titulo, autor, cantidad, precio in inventario:
        cantidades.append(cantidad)
    return cantidades
```

Pero en este caso también podemos obtener el mismo resultado con una lista por comprensión:

```
def obtener_cantidades(inventario):
    return [cantidad for titulo, autor, cantidad, precio in inventario]
```

En este caso la sintaxis utilizada es un poco diferente:

```
[<expresión> for <variable> in <secuencia>]
```

Opcionalmente podemos combinar el filtro y la transformación en una única lista por comprensión:

```
def cantidades_autor(inventario, autor_buscado):
    return [
        cantidad
        for titulo, autor, cantidad, precio in inventario
        if autor == autor_buscado
    ]
```

Es decir que la sintaxis general de las listas por comprensión es:

```
[<expresión> for <variable> in <secuencia> if <condicion>]
```

Volviendo a nuestro problema inicial: obtener la cantidad total de libros del autor; ya tenemos una forma de filtrar y transformar, y lo único que nos falta es la acumulación. Pero recordemos que ya conocíamos una forma simple de acumular sumando elementos: ¡la función `sum`!

```
def total_libros_autor(inventario, autor_buscado):
    return sum([
        cantidad
        for titulo, autor, cantidad, precio in inventario
        if autor == autor_buscado
    ])
```

Planteemos ahora las soluciones para los otros dos problemas utilizando filtros, transformaciones y acumulaciones. Nuestro segundo problema era obtener la cantidad de títulos de los cuales no hay suficiente stock.

1. Filtramos según la cantidad de stock, quedándonos con los libros para los cuales `cantidad < 5`.
2. No es necesario aplicar una transformación.
3. Solo necesitamos la cantidad de títulos, y eso es simplemente la cantidad de elementos de la lista producida en el paso anterior. Es decir que nuestra función de acumulación es `len`.

```
def cantidad_poco_stock(inventario):
    return len([libro for libro in inventario if libro[3] < 5])
```

Nuestro tercer problema era obtener la lista de títulos cuyo precio supera los $100.

1. Filtramos según el precio, quedándonos con la lista de libros con `precio > 100`.
2. Transformamos cada tupla quedándonos únicamente con el `titulo`.
3. No es necesario aplicar una acumulación.

```
def titulos_caros(inventario):
    return [
        titulo
        for titulo, autor, cantidad, precio in inventario
        if precio > 100
    ]
```

Comparando estas soluciones con las primeras tres soluciones propuestas, vemos que son dos estilos de programación diferentes:

- Las primeras soluciones corresponden a un estilo más *procedural* e *imperativo*. Cuando pensamos en este estilo nos concentramos en dar órdenes para especificar *cómo* queremos que la computadora resuelva el problema, paso por paso.
- Las soluciones planteadas utilizando filtros, transformaciones y acumulaciones corresponden a un estilo más *funcional* y *declarativo*, en el cual dividimos el problema en subproblemas más simples, y nos concentramos en especificar cómo es el flujo de datos.

La discusión acerca de si uno de los dos estilos es "mejor" que el otro queda fuera del alcance de este apunte, pero en general se considera que el uso de listas por comprensión es *idiomático* en Python. Es decir, los programadores Python experimentados van a preferir leer y escribir código que utilice listas por comprensión en lugar de implementar los filtros y transformaciones a mano.

Por último, observamos que las listas por comprensión ofrecen una alternativa sintáctica a las funciones `map` y `filter`, explicadas en el Apéndice 7.A. Dejamos como ejercicio implementar las funciones `total_libros_autor`, `cantidad_poco_stock` y `titulos_caros` en términos de `map` y `filter`.

Unidad 9

Diccionarios

En esta unidad analizaremos otra estructura de datos importante: los diccionarios. Su importancia radica no sólo en las grandes posibilidades que presentan como estructuras para almacenar información, sino también en que, en Python, son utilizados por el propio lenguaje para realizar diversas operaciones y para almacenar información de otras estructuras.

9.1 Qué es un diccionario

Según Wikipedia, "[u]n diccionario es una obra de consulta de palabras y/o términos que se encuentran generalmente ordenados alfabéticamente. De dicha compilación de palabras o términos se proporciona su significado, etimología, ortografía y, en el caso de ciertas lenguas fija su pronunciación y separación silábica."

Al igual que los diccionarios a los que se refiere Wikipedia, y que usamos habitualmente en la vida diaria, los diccionarios de Python son una lista de consulta de términos de los cuales se proporcionan valores asociados. A diferencia de los diccionarios a los que se refiere Wikipedia, los diccionarios de Python no están ordenados.

En Python, un diccionario es una colección no-ordenada de valores que son accedidos a traves de una clave. Es decir, en lugar de acceder a la información mediante el índice numérico, como es el caso de las listas y tuplas, es posible acceder a los valores a través de sus claves, que pueden ser de diversos tipos.

Las claves son únicas dentro de un diccionario, es decir que no puede haber un diccionario que tenga dos veces la misma clave. Si se asigna un valor a una clave ya existente, se reemplaza el valor anterior.

No hay una forma directa de acceder a una clave a través de su valor, y nada impide que un mismo valor se encuentre asignado a distintas claves.

La informacion almacenada en los diccionarios no tiene un orden particular. Ni por clave ni por valor, ni tampoco por el orden en que han sido agregados al diccionario.

Al igual que las listas, los diccionarios son mutables. Esto significa que podemos agregar, quitar y modificar los elementos de un diccionario posteriormente a su creación.

Cualquier valor de tipo inmutable puede ser clave de un diccionario: cadenas, enteros, tuplas (con valores inmutables en sus miembros), etc. No hay restricciones para los valores que el diccionario puede contener, cualquier tipo puede ser el valor: listas, cadenas, tuplas, otros diccionarios, etc.

Sabías que…

En otros lenguajes, a los diccionarios se los llama *arreglos asociativos*, *mapas*, *tablas* o también *tablas de hash*.

9.2 Utilizando diccionarios en Python

De la misma forma que con listas, es posible definir un diccionario directamente con los miembros que va a contener, o bien inicializar el diccionario vacío y luego agregar los valores de a uno o de a muchos.

Para definirlo junto con los miembros que va a contener, se encierra el listado de valores entre llaves, las parejas de clave y valor se separan con comas, y la clave y el valor se separan con ':'.

```
punto = {'x': 2, 'y': 1, 'z': 4}
```

En Python el tipo de dato asociado a los diccionarios se llama `dict`:

```
>>> type(punto)
<class 'dict'>
```

Para declararlo vacío y luego ingresar los valores, se lo declara como un par de llaves sin nada en medio, y luego se asignan valores directamente a los índices.

```
materias = {}
materias["lunes"] = [6103, 7540]
materias["martes"] = [6201]
materias["miércoles"] = [6103, 7540]
materias["jueves"] = []
materias["viernes"] = [6201]
```

Para acceder al valor asociado a una determinada clave, se lo hace de la misma forma que con las listas, pero utilizando la clave elegida en lugar del índice.

```
>>> materias["lunes"]
[6103, 7540]
```

Atención

El acceso por clave falla si se provee una clave que no está en el diccionario:

```
>>> materias["domingo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'domingo'
```

El operador `in` nos permite preguntar si una clave se encuentra o no en el diccionario:

```
>>> "lunes" in materias
True
>>> "domingo" in materias
False
```

Además podemos utilizar la función `get`, que recibe una clave k y un valor por omisión v, y devuelve el valor asociado a la clave k, en caso de existir, o el valor v en caso contrario.

```
>>> materias.get("lunes", [])
[6103, 7540]
>>> materias.get("domingo", [])
[]
```

Existen diversas formas de recorrer un diccionario. Es posible recorrer sus claves y usar esas claves para acceder a los valores.

```
for dia in materias:
    print("El {} tengo que cursar {}".format(dia, materias[dia])
```

Es posible, también, obtener los valores como tuplas donde el primer elemento es la clave y el segundo el valor.

```
for dia, codigos in materias.items():
    print("El {} tengo que cursar {}".format(dia, codigos)
```

Hay muchas otras operaciones que se pueden realizar sobre los diccionarios, que permiten manipular la información según sean nuestras necesidades. Algunos de estos métodos pueden verse en la referencia al final de la unidad.

Sabías que…

El algoritmo que usa Python internamente para buscar un elemento en un diccionario es muy distinto que el que utiliza para buscar en listas.

Para buscar en las listas, se utiliza un algoritmos de comparación que tarda cada vez más a medida que la lista se hace más larga. En cambio, para buscar en diccionarios se utiliza un algoritmo llamado *hash*, que se basa en realizar un cálculo numérico sobre la clave del elemento, y tiene una propiedad muy interesante: sin importar cuántos elementos tenga el diccionario, el tiempo de búsqueda es siempre aproximadamente igual.

Este algoritmo de *hash* es también la razón por la cual las claves de los diccionarios deben ser inmutables, ya que la operación hecha sobre las claves debe dar siempre el mismo resultado, y si se utilizara un valor mutable esto no sería posible.

No es posible obtener porciones de un diccionario usando `[:]`, ya que al no tener un orden determinado para los elementos, no sería posible tomarlos en orden.

9.3 Algunos usos de diccionarios

Los diccionarios son una herramienta muy versátil. Se puede utilizar un diccionario, por ejemplo, para contar cuántas apariciones de cada palabra hay en un texto, o cuántas apariciones de cada letra.

Es posible utilizar un diccionario, también, para tener una agenda donde la clave es el nombre de la persona, y el valor es una lista con los datos correspondientes a esa persona.

También podría utilizarse un diccionario para mantener los datos de los alumnos inscriptos en una materia, siendo la clave el número de padrón, y el valor una lista con todas las notas asociadas a ese alumno.

En general, los diccionarios sirven para crear bases de datos muy simples, en las que la clave es el identificador del elemento, y el valor son todos los datos del elemento a considerar.

Otro posible uso de un diccionario sería para realizar traducciones, donde la clave sería la palabra en el idioma original y el valor la palabra en el idioma al que se quiere traducir. Sin embargo esta aplicación es poco destacable, ya que esta forma de traducir suele dar resultados poco satisfactorios.

9.4 Resumen

- Los diccionarios (también llamados *arreglos asociativos*, *mapas* o *tablas de hash* en otros lenguajes), son una estructura de datos muy poderosa, que permite almacenar un conjunto de pares *clave* → *valor*.
- Las claves deben ser inmutables y únicas.
- Los valores pueden ser de cualquier tipo, y pueden no ser únicos.
- Los diccionarios no están ordenados. Si bien se los puede recorrer, el orden en el que se tomarán los elementos no está determinado.

Referencia Python

`{clave1:valor1, clave2:valor2}`

Se crea un nuevo diccionario con los valores asociados a las claves. Si no se ingresa ninguna pareja de clave y valor, se crea un diccionario vacío.

`diccionario[clave]`

Accede al valor asociado con `clave` en el diccionario. Falla si la clave no está en el diccionario.

`clave in diccionario`

Indica si un diccionario tiene o no una determinada clave.

`diccionario.get(clave, valor_predeterminado)`

Devuelve el valor asociado a la clave. A diferencia del acceso directo utilizando `[clave]`, en el caso en que el valor no se encuentre devuelve el `valor_predeterminado`.

`for clave in diccionario:`

Permite recorrer una a una todas las claves almacenadas en el diccionario.

`diccionario.keys()`

Devuelve una secuencia desordenada, con todas las claves que se hayan ingresado al diccionario

`diccionario.values()`

Devuelve una secuencia desordenada, con todos los valores que se hayan ingresado al diccionario.

`diccionario.items()`

Devuelve una secuencia desordenada con tuplas de dos elementos, en las que el primer elemento es la clave y el segundo el valor.

`diccionario.pop(clave)`

Quita del diccionario la clave y su valor asociado, y devuelve el valor.

9.5 Ejercicios

Ejercicio 9.5.1. Escribir una función que reciba una lista de tuplas, y que devuelva un diccionario en donde las claves sean los primeros elementos de las tuplas, y los valores una lista con los segundos.

Por ejemplo:

```
>>> l = [ ('Hola', 'don Pepito'), ('Hola', 'don Jose'),
          ('Buenos', 'días') ]
>>> print(tuplas_a_diccionario(l))
{ 'Hola': ['don Pepito', 'don Jose'], 'Buenos': ['días'] }
```

Ejercicio 9.5.2. Diccionarios usados para contar.

a) Escribir una función que reciba una cadena y devuelva un diccionario con la cantidad de apariciones de cada palabra en la cadena. Por ejemplo, si recibe "Qué lindo día que hace hoy" debe devolver: `{ 'que': 2, 'lindo': 1, 'día': 1, 'hace': 1, 'hoy': 1}`.

b) Escribir una función que cuente la cantidad de apariciones de cada caracter en una cadena de texto, y los devuelva en un diccionario.

c) Escribir una función que reciba una cantidad de iteraciones de una tirada de 2 dados a realizar y devuelva la cantidad de veces que se observa cada valor de la suma de los dos dados.
Nota: utilizar el módulo `random` para obtener tiradas aleatorias.

Ejercicio 9.5.3. Continuación de la agenda.
Escribir un programa que vaya solicitando al usuario que ingrese nombres.

a) Si el nombre se encuentra en la agenda (*implementada con un diccionario*), debe mostrar el teléfono y, opcionalmente, permitir modificarlo si no es correcto.

b) Si el nombre no se encuentra, debe permitir ingresar el teléfono correspondiente.

El usuario puede utilizar la cadena "*", para salir del programa.

Ejercicio 9.5.4. Escribir una función que reciba un texto y para cada caracter presente en el texto devuelva la cadena más larga en la que se encuentra ese caracter.

Apéndice 9.A Conjuntos

Supongamos que queremos modelar un registro de donantes de órganos. Este registro se comporta como un *conjunto* de elementos, donde cada elemento es una persona, y el conjunto:

1. no puede contener elementos repetidos: una persona puede ser donante o no, pero no puede figurar dos o más veces en el registro.
2. debe permitir averiguar si un elemento pertenece o no al conjunto en *tiempo constante*: no importa si hay 1, 10 o 100000 donantes, queremos tener la capacidad de averiguar si una persona determinada es donante o no rápidamente.

Si implementáramos el registro usando una lista de Python, nos encontraríamos con que no cumplimos con los requisitos:

1. La lista puede contener elementos repetidos. Si bien podemos salvar este detalle preguntando si el elemento se encuentra o no en la lista antes de agregarlo, esto sería muy poco eficiente, ya que:
2. Como vimos en la Sección 8.1, buscar un elemento en la lista consume una cantidad de tiempo proporcional a la cantidad de elementos presentes en la lista, con lo que no es posible cumplir con el requerimiento de tiempo constante.

Una solución posible es usar un diccionario, guardando como clave el número de documento de la persona donante, y como valor… ¡cualquier cosa! No importa qué vayamos asignar como valor, ya que lo único que queremos aprovechar del diccionario es la capacidad de tener claves únicas y poder preguntar en tiempo constante si una clave está o no presente. Por ejemplo, si usamos `True` como valor:

```
>>> donantes = {12345: True, 23456: True}
>>> 23456 in donantes
True
>>> 34567 in donantes
False
>>> donantes.pop(23456)
>>> 23456 in donantes
False
```

Sin embargo, utilizar un diccionario para modelar un conjunto de elementos no es la solución más elegante. Hay una mejor forma de hacerlo, que es utilizando el tipo de datos `set`[1].

Para crear un `set` usamos la sintaxis `{<expresión>, <expresión>, ...}`:

```
>>> donantes = {12345, 23456}
>>> type(donantes)
<class 'set'>
>>> 23456 in donantes
True
>>> 34567 in donantes
False
>>> donantes.remove(23456)
>>> 23456 in donantes
False
```

[1] La palabra “set” significa “conjunto” en inglés.

Un `set` es una estructura de datos mutable (como las listas y los diccionarios), que permite agregar y quitar elementos cumpliendo los requisitos de unicidad y búsqueda en tiempo constante. Además es posible hacer operaciones entre `set`s como unión, intersección y diferencia muy fácilmente:

```
>>> s1 = {1, 2, 3, 4}
>>> s1
{1, 2, 3, 4}
>>> s1.add(1)
>>> s1
{1, 2, 3, 4}
>>> s2 = {3, 4, 5, 6}
>>> s1.union(s2)
{1, 2, 3, 4, 5, 6}
>>> s1.intersection(s2)
{3, 4}
>>> s1.difference(s2)
{1, 2}
```

Notar que la sintaxis para crear un conjunto es muy similar a la de creación de diccionarios. El caso especial es cuando queremos crear un conjunto vacío: la sintaxis `{}` no funcionará, ya que eso crea un diccionario vacío. Podemos crear un conjunto vacío con: `set()`.

```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
```

La referencia completa del tipo de dato `set` puede verse en https://docs.python.org/3/library/stdtypes.html#set.

Unidad 10

Contratos y Mutabilidad

En esta unidad se le dará cierta formalización a algunos temas que habían sido presentados informalmente, como por ejemplo la documentación de las funciones.

Se formalizarán las condiciones que debe cumplir un algoritmo al comenzar, en su transcurso, y al terminar, y algunas técnicas para tener en cuenta estas condiciones.

También se verá una forma de modelar el espacio donde *viven* las variables.

10.1 Pre y Postcondiciones

Cuando hablamos de *contratos* o *programación por contratos*, nos referimos a la necesidad de estipular tanto lo que necesita como lo que devuelve nuestro código.

Las condiciones que deben estar dadas para que el código funcione las llamamos *precondiciones* y las condiciones sobre el estado en que quedan las variables y él o los valores de retorno, las llamamos *postcondiciones*.

En definitiva, este concepto es similar al ya mencionado con respecto a la documentación de funciones, es decir que se debe *documentar cómo deben ser los parámetros recibidos, cómo va a ser lo que se devuelve, y qué sucede con los parámetros en caso de ser modificados*.

Esta estipulación es mayormente para que la utilicen otros programadores, por lo que es particularmente útil cuando se encuentra dentro de la documentación. En ciertos casos, además, puede quererse que el programa revise si las condiciones realmente se cumplen y de no ser así, actúe en consecuencia.

Existen herramientas en algunos lenguajes de programación que facilitan estas acciones, en el caso de Python, es posible utilizar la instrucción `assert`.

10.1.1 Precondiciones

Las precondiciones son las condiciones que deben cumplir los parámetros que una función recibe, para que esta se comporte correctamente.

Por ejemplo, en una función división las precondiciones son que los parámetros son números, y que el divisor sea distinto de 0. Tener una precondición permite asumir desde el código que no es necesario lidiar con los casos en que las precondiciones no se cumplen.

10.1.2 Postcondiciones

Las postcodiciones son las condiciones que cumplirá el valor de retorno, y los parámetros recibidos, en caso de que hayan sido alterados, siempre que se hayan cumplido las precondi-

ciones.

En el ejemplo anterior, la función división con las precondiciones asignadas, puede asegurar que devolverá un número correspondiente al cociente solicitado.

10.1.3 Aseveraciones

Tanto las precondiciones como las postcondiciones son *aseveraciones* (en inglés *assertions*). Es decir, afirmaciones realizadas en un momento particular de la ejecución sobre el estado computacional. Si llegaran a ser falsas significaría que hay algún error en el diseño o utilización del algoritmo.

Para comprobar estas afirmaciones desde el código en algunos casos podemos utilizar la instrucción `assert`. Esta instrucción recibe una condición a verificar (o sea, una expresión booleana) Si la condición es `True`, la instrucción no hace nada; en caso contrario se produce un error.

```
>>> assert True
>>> assert False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Opcionalmente, la instrucción `assert` puede recibir un mensaje de error que mostrará en caso que la condición no se cumpla.

```
>>> n = 0
>>> assert n != 0, "El divisor no puede ser 0"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: El divisor no puede ser 0
```

> **Atención**
>
> Es importante tener en cuenta que `assert` está pensado para ser usado en la etapa de desarrollo. Un programa terminado nunca debería dejar de funcionar por este tipo de errores.

10.1.4 Ejemplos

Usando los ejemplos anteriores, la función `division` nos quedaría de la siguiente forma:

```
def division(dividendo, divisor):
    """Cálculo de la división

    Pre: Recibe dos números, divisor debe ser distinto de 0.
    Post: Devuelve un número real, con el cociente de ambos.
    """
    assert divisor != 0, "El divisor no puede ser 0"
    return dividendo / divisor
```

Otro ejemplo, tal vez más interesante, puede ser una función que implemente una sumatoria ($\sum_{i=inicial}^{final} f(i)$). En este caso hay que analizar cuáles van a ser los parámetros que recibirá la función, y las precondiciones que estos parámetros deberán cumplir.

La función `sumatoria` a escribir necesita de un valor inicial, un valor final, y una función a la cual llamar en cada paso. Es decir que recibe tres parámetros.

```
def sumatoria(inicial, final, f):
```

Tanto `inicial` como `final` deben ser números enteros, y dependiendo de la implementación a realizar o de la especificación previa, puede ser necesario que `final` deba ser mayor o igual a `inicial`.

Con respecto a `f`, se trata de una función que será llamada con un parámetro en cada paso y se requiere poder sumar el resultado, por lo que debe ser una función que reciba un número y devuelva un número.

La declaración de la función queda, entonces, de la siguiente manera.

```
def sumatoria(inicial, final, f):
    """Calcula la sumatoria desde i=inicial hasta final de f(i)

    Pre: inicial y final son números enteros, f es una función que
         recibe un entero y devuelve un número.
    Post: Se devuelve el valor de la sumatoria de aplicar f a cada
          número comprendido entre inicial y final.
    """
```

Ejercicio 10.1.1. Realizar la implementación correspondiente a la función `sumatoria`.

En definitiva, la documentación de pre y postcondiciones dentro de la documentación de las funciones es una forma de especificar claramente el comportamiento del código de forma que quienes lo vayan a utilizar no requieran conocer cómo está implementado para poder aprovecharlo.

Esto es útil incluso en los casos en los que el programador de las funciones es el mismo que el que las va a utilizar, ya que permite separar responsabilidades. Las pre y postcondiciones son, en efecto, un *contrato* entre el código invocante y el invocado.

10.2 Invariantes de ciclo

Los invariantes se refieren a estados o situaciones que no cambian dentro de un contexto o porción de código. Hay invariantes de ciclo, que son los que veremos a continuación, e invariantes de estado, que se verán más adelante.

El invariante de ciclo permite conocer cómo llegar desde las precondiciones hasta las postcondiciones, cuando la implementación se compone de un ciclo. El invariante de ciclo es, entonces, una aseveración que debe ser verdadera al comienzo de cada iteración.

Por ejemplo, si el problema es ir desde el punto A al punto B, las precondiciones dicen que estamos parados en A y las postcondiciones que estamos parados en B, un invariante podría ser "estamos en algún punto entre A y B, en el punto más cercano a B que estuvimos hasta ahora.".

Más específicamente, si analizamos el ciclo para buscar el máximo en una lista desordenada, la precondición es que la lista contiene elementos que son comparables y la postcondición es que se devuelve el elemento máximo de la lista.

```
def maximo(lista):
    "Devuelve el elemento máximo de la lista o None si está vacía."
    if not lista:
        return None
    max_elem = lista[0]
    for elemento in lista:
        if elemento > max_elem:
            max_elem = elemento
```

```
    return max_elem
```

En este caso, el invariante del ciclo es que `max_elem` contiene el valor máximo de la porción de lista analizada.

Los invariantes son de gran importancia al momento de demostrar que un algoritmo funciona, pero aún cuando no hagamos una demostración formal es muy útil tener los invariantes a la vista, ya que de esta forma es más fácil entender cómo funciona un algoritmo y encontrar posibles errores.

Los invariantes, además, son útiles a la hora de determinar las condiciones iniciales de un algoritmo, ya que también deben cumplirse para ese caso. Por ejemplo, consideremos el algoritmo para obtener la potencia `n` de un número.

```
def potencia(b, n):
    "Devuelve la potencia n del número b, con n entero mayor que 0."
    p = 1
    for i in range(n):
        p *= b
    return p
```

En este caso, el invariante del ciclo es que la variable `p` contiene el valor de la potencia correspondiente a esa iteración. Teniendo en cuenta esta condición, es fácil ver que `p` debe comenzar el ciclo con un valor de 1, ya que ese es el valor correspondiente a p^0.

De la misma manera, si la operación que se quiere realizar es sumar todos los elementos de una lista, el invariante será que una variable `suma` contenga la suma de todos los elementos ya recorridos, por lo que es claro que este invariante debe ser 0 cuando aún no se haya recorrido ningún elemento.

```
def suma(lista):
    "Devuelve la suma de todos los elementos de la lista."
    suma = 0
    for elemento in lista:
        suma += elemento
    return suma
```

10.2.1 Comprobación de invariantes desde el código

Cuando la comprobación necesaria para saber si seguimos "en camino" es simple, se la puede tener directamente dentro del código. Evitando seguir avanzando con el algoritmo si se produjo un error crítico.

Por ejemplo, en una búsqueda binaria, el elemento a buscar debe ser mayor que el elemento inicial y menor que el elemento final, de no ser así, no tiene sentido continuar con la búsqueda. Es posible, entonces, agregar una instrucción que compruebe esta condición y de no ser cierta realice alguna acción para indicar el error, por ejemplo, utilizando la instrucción `assert`, vista anteriormente.

10.3 Mutabilidad e Inmutabilidad

Hasta ahora cada vez que estudiamos un tipo de datos indicamos si son mutables o inmutables.

Cuando un valor es de un tipo inmutable, como por ejemplo una cadena, es posible asignar un nuevo valor a esa variable, pero no es posible modificar su contenido.

```
>>> s = "ejemplo"
>>> s = "otro"
>>> s[2] = "c"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Esto se debe a que cuando se realiza una nueva asignación, no se modifica la cadena en sí, sino que la variable s pasa a *referenciar* a otra cadena. En cambio, no es posible asignar un nuevo caracter en una posición, ya que esto implicaría modificar la cadena inmutable.

En el caso de los parámetros mutables, la asignación tiene el mismo comportamiento, es decir que las variables pasan a apuntar a un nuevo valor.

```
>>> lista1 = [10, 20, 30]
>>> lista2 = lista1
>>> lista1 = [3, 5, 7]
>>> lista1
[3, 5, 7]
>>> lista2
[10, 20, 30]
```

Algo importante a tener en cuenta en el caso de las variables de tipo mutable es que si hay dos o más variables que *referencian* a un mismo valor, y este valor se modifica, el cambio se verá reflejado en ambas variables.

```
>>> lista1 = [1, 2, 3]
>>> lista2 = lista1
>>> lista2[1] = 5
>>> lista1
[1, 5, 3]
```

Sabías que…

En otros lenguajes, como C o C++, existe un tipo de variable especial llamado *puntero*, que se comporta como una referencia a un valor, como es el caso de las variables mutables del ejemplo anterior.

En Python no hay punteros como los de C o C++, pero todas las variables son referencias a una porción de memoria, de modo que cuando se asigna una variable a otra, lo que se está asignando es la porción de memoria a la que refieren. Si esa porción de memoria cambia, el cambio se puede ver en todas las variables que apuntan a esa porción.

10.3.1 Parámetros mutables e inmutables

Las funciones reciben parámetros que pueden ser mutables o inmutables.

Si dentro del cuerpo de la función se modifica uno de estos parámetros para que *referencie* a otro valor, este cambio no se verá reflejado fuera de la función. Si, en cambio, se modifica el *contenido* de alguno de los parámetros mutables, este cambio *sí* se verá reflejado fuera de la función.

A continuación un ejemplo en el cual se asigna la variable recibida, a un nuevo valor. Esa asignación sólo tiene efecto dentro de la función.

```
>>> def no_cambia_lista(lista):
```

```
...     lista = [0, 1, 2, 3]
...     print("Dentro de la funcion lista =", lista)
...
>>> lista = [10, 20, 30, 40]
>>> no_cambia_lista(lista)
Dentro de la funcion lista = [0, 1, 2, 3]
>>> lista
[10, 20, 30, 40]
```

A continuación un ejemplo en el cual se modifica la variable recibida. En este caso, los cambios realizados tienen efecto tanto dentro como fuera de la función.

```
>>> def cambia_lista(lista):
...     for i in range(len(lista)):
...         lista[i] = lista[i] ** 3
...
>>> lista = [1, 2, 3, 4]
>>> cambia_lista(lista)
>>> lista
[1, 8, 27, 64]
```

Atención

En general, se espera que una función que recibe parámetros mutables no los modifique, ya que si se los modifica se podría perder información valiosa.

En el caso en que por una decisión de diseño o especificación se modifiquen los parámetros recibidos, esto debe estar claramente documentado, dentro de las postcondiciones.

10.4 Resumen

- Las **precondiciones** son las condiciones que deben cumplir los parámetros recibidos por una función.
- Las **postcondiciones** son las condiciones cumplidads por los resultados que la función devuelve y por los parámetros recibidos, siempre que las precondiciones hayan sido válidas.
- Los **invariantes de ciclo** son las condiciones que deben cumplirse al comienzo de cada iteración de un ciclo.
- En el caso en que estas **aseveraciones** no sean verdaderas, se deberá a un error en el diseño o utilización del código.
- En general una función no debe modificar el contenido de sus parámetros, aún cuando esto sea posible, a menos que sea la funcionalidad explícita de esa función.

Referencia Python

`assert condicion[, mensaje]`

Verifica si la condición es verdadera. En caso contrario, provoca un error con el mensaje recibido por parámetro.

Apéndice 10.A Acertijo MU

El acertijo MU[1] es un buen ejemplo de un problema lógico donde es útil determinar el invariante. El acertijo consiste en buscar si es posible convertir MI a MU, utilizando las siguientes operaciones.

1. Si una cadena termina con una I, se le puede agregar una U (xI -> xIU)
2. Cualquier cadena luego de una M puede ser totalmente duplicada (Mx -> Mxx)
3. Donde haya tres Is consecutivas (III) se las puede reemplazar por una U (xIIIy -> xUy)
4. Dos Us consecutivas, pueden ser eliminadas (xUUy -> xy)

Para resolver este problema, es posible pasar horas aplicando estas reglas a distintas cadenas. Sin embargo, puede ser más fácil encontrar una afirmación que sea invariante para todas las reglas y que muestre si es o no posible llegar a obtener MU.

Al analizar las reglas, la forma de deshacerse de las Is es conseguir tener tres Is consecutivas en la cadena. La única forma de deshacerse de todas las Is es que haya un cantidad de Is consecutivas múltiplo de tres.

Es por esto que es interesante considerar la siguiente afirmación como invariante: el número de Is en la cadena no es múltiplo de tres.

Para que esta afirmación sea invariante al acertijo, para cada una de las reglas se debe cumplir que: si el invariante era verdadero antes de aplicar la regla, seguirá siendo verdadero luego de aplicarla.

Para ver si esto es cierto o no, es necesario considerar la aplicación del invariante para cada una de las reglas.

1. Se agrega una U, la cantidad de Is no varía, por lo cual se mantiene el invariante.
2. Se duplica toda la cadena luego de la M, siendo n la cantidad de Is antes de la duplicación, si n no es múltiplo de 3, $2n$ tampoco lo será.
3. Se reemplazan tres Is por una U. Al igual que antes, siendo n la cantidad de Is antes del reemplazo, si n no es múltiplo de 3, $n - 3$ tampoco lo será.
4. Se eliminan Us, la cantidad de Is no varía, por lo cual se mantiene el invariante.

Todo esto indica claramente que el invariante se mantiene para cada una de las posibles transformaciones. Esto significa que sea cual fuere la regla que se elija, si la cantidad de Is no es un múltiplo de tres antes de aplicarla, no lo será luego de hacerlo.

Teniendo en cuenta que hay una única I en la cadena inicial MI, y que uno no es múltiplo de tres, es imposible llegar a MU con estas reglas, ya que MU tiene cero Is, que sí es múltiplo de tres.

[1] http://en.wikipedia.org/wiki/Invariant_(computer_science)

Unidad 11

Manejo de archivos

Veremos en esta unidad cómo manejar archivos desde nuestros programas.

Existen dos formas básicas de acceder a un archivo, una es utilizarlo como un archivo de texto, que procesaremos línea por línea; la otra es tratarlo como un archivo binario, que procesaremos byte por byte.

En Python, para abrir un archivo usaremos la función `open`, que recibe el nombre del archivo a abrir.

```
archivo = open("archivo.txt")
```

Esta función intentará abrir el archivo con el nombre indicado. Si tiene éxito devolverá un valor de un tipo especial, que nos permitirá manipular el archivo de diversas maneras.

La operación más sencilla a realizar sobre un archivo es leer su contenido. Para procesarlo línea por línea, es posible hacerlo de la siguiente forma:

```
linea = archivo.readline()
while linea != '':
    # procesar linea
    linea = archivo.readline()
```

Esto funciona ya que cada archivo que se encuentre abierto tiene una posición asociada, que indica el último punto que fue leído. Cada vez que se lee una línea, avanza esa posición. Es por ello que `readline()` devuelve cada vez una línea distinta y no siempre la misma.

La siguiente estructura es una forma equivalente a la vista en el ejemplo anterior.

```
for linea in archivo:
    # procesar linea
```

De esta manera, la variable `linea` irá almacenando distintas cadenas correspondientes a cada una de las líneas del archivo.

Es posible, además, obtener todas las líneas del archivo utilizando una sola llamada a función:

```
lineas = archivo.readlines()
```

En este caso, la variable `lineas` tendrá una lista de cadenas con todas las líneas del archivo.

> **⚠ Atención**
>
> Es importante tener en cuenta que cuando se utilizan funciones como `archivo.readlines()`, se está cargando en memoria el archivo completo. Siempre que una instrucción cargue un archivo completo en memoria debe tenerse cuidado de utilizarla sólo con archivos pequeños, ya que de otro modo podría agotarse la memoria de la computadora.

11.1 Cerrar un archivo

Al terminar de trabajar con un archivo, es importante cerrarlo, por diversos motivos: en algunos sistemas los archivos sólo pueden ser abiertos de a un programa por la vez; en otros, lo que se haya escrito no se guardará realmente hasta no cerrar el archivo; o el límite de cantidad de archivos que puede manejar un programa puede ser bajo, etc.

Para cerrar un archivo simplemente se debe llamar a:

```
archivo.close()
```

Además, Python nos provee con una estructura que permite cerrar el archivo automáticamente, sin necesidad de llamar a `close`:

```
with open("archivo.txt") as archivo:
    #
    # procesar el archivo
    #

# Cuando la ejecución sale del bloque 'with',
# el archivo se cierra automáticamente.
```

11.2 Ejemplo de procesamiento de archivos

Por ejemplo, para mostrar todas las líneas de un archivo, precedidas por el número de línea, podemos hacerlo de la siguiente manera:

```
archivo = open("archivo.txt")
i = 1
for linea in archivo:
    linea = linea.rstrip("\n")
    print("{}: {}".format(i, linea))
    i += 1
archivo.close()
```

La llamada a `rstrip` es necesaria ya que cada línea que se lee del archivo contiene un caracter especial llamado *fin de línea* y con la llamada a `rstrip("\n")` se remueve.

Notar que sería equivalente usar el bloque `with` para ahorrarnos la llamada a `close`:

```
with open("archivo.txt") as archivo:
    i = 1
    for linea in archivo:
        linea = linea.rstrip("\n")
        print("{}: {}".format(i, linea))
        i += 1
```

También podemos utilizar la función `enumerate` (explicada en la sección 7.2.3) para no tener que mantener el contador `i` a mano:

```
with open("archivo.txt") as archivo:
    for i, linea in enumerate(archivo):
        linea = linea.rstrip("\n")
        print("{}: {}".format(i, linea))
```

Sabías que…

Los archivos de texto son sencillos de manejar, pero existen por lo menos 3 formas distintas de marcar un fin de línea. En Unix tradicionalmente se usa el caracter '\n' (valor de ASCII 10, definido como nueva línea) para el fin de línea, mientras que en OSX el fin de línea se solía representar como un '\r' (valor ASCII 13, definido como retorno de carro) y en Windows se usan ambos caracteres '\r\n'.

Al leer archivos en modo texto, Python acepta cualquier tipo de fin de línea como si fuese un \n. Al escribir archivos, Python elegirá automáticamente el modo más apropiado. Si queremos modificar este comportamiento podemos especificar el modo utilizando la opción `newline` de la función `open`.

11.3 Modo de apertura de los archivos

La función `open` recibe un parámetro opcional para indicar el modo en que se abrirá el archivo. Los tres modos de apertura que se pueden especificar son:

- Modo de **sólo lectura** (`'r'`). En este caso no es posible realizar modificaciones sobre el archivo, solamente leer su contenido.
- Modo de **sólo escritura** (`'w'`). En este caso el archivo es truncado (vaciado) si existe, y se lo crea si no existe.
- Modo **sólo escritura posicionándose al final del archivo** (`a`). En este caso se crea el archivo, si no existe, pero en caso de que exista se posiciona al final, manteniendo el contenido original.

Por otro lado, en cualquiera de estos modos se puede agregar un `+` para pasar a un modo lectura-escritura. El comportamiento de `r+` y de `w+` no es el mismo, ya que en el primer caso se tiene el archivo completo, y en el segundo caso se trunca el archivo, perdiendo así los datos.

Si un archivo no existe y se lo intenta abrir en modo lectura, se generará un error; en cambio si se lo abre para escritura, Python se encargará de crear el archivo al momento de abrirlo, ya sea con `'w'`, `'a'`, `'w+'` o con `'a+'`.

En caso de que no se especifique el modo, los archivos serán abiertos en modo sólo lectura (`r`).

Atención

Si un archivo existente se abre en modo escritura (`'w'` o `'w+'`), todos los datos anteriores son borrados y reemplazados por lo que se escriba en él.

11.4 Escribir en un archivo

De la misma forma que para la lectura, existen dos formas distintas de escribir a un archivo. Mediante cadenas:

```
archivo.write(cadena)
```

O mediante listas de cadenas:

```
archivo.writelines(lista_de_cadenas)
```

Así como la función `readline` devuelve las líneas con los caracteres de fin de línea (`\n`), será necesario agregar los caracteres de fin de línea a las cadenas que se vayan a escribir en el archivo.

Por ejemplo, el siguiente programa genera a su vez el código de otro programa Python y lo guarda en el archivo `saludo.py`:

```
with open("saludo.py", "w") as saludo:
    saludo.write("# Este programa fue generado por otro programa!\n")
    saludo.write("print('Hola Mundo')\n")
```

Atención

Si un archivo existente se abre en modo lectura-escritura, al escribir en él se sobreescribirán los datos anteriores, a menos que se haya llegado al final del archivo.

Este proceso de sobreescritura se realiza caracter por caracter, sin consideraciones adicionales para los caracteres de fin de línea ni otros caracteres especiales.

11.5 Agregar información a un archivo

Abrir un archivo en modo *agregar al final* puede parece raro, pero es bastante útil.

Uno de sus usos es para escribir un archivo de bitácora (o archivo de *l*og), que nos permita ver los distintos eventos que se fueron sucediendo, y así encontrar la secuencia de pasos (no siempre evidente) que hace nuestro programa.

Esta es una forma muy habitual de buscar problemas o hacer un seguimiento de los sucesos. Para los administradores de sistemas es una herramienta esencial de trabajo.

En el Código 11.1 se muestra un módulo para manejo de logs, que se encarga de la apertura del archivo, del guardado de las líneas una por una y del cerrado final del archivo.

En este módulo se utiliza el módulo de Python `datetime` para obtener la fecha y hora actual que se guardará en los archivos. Es importante notar que en el módulo mostrado no se abre o cierra un archivo en particular, sino que las funciones están programadas de modo tal que puedan ser utilizadas desde otro programa.

Se trata de un módulo genérico que podrá ser utilizado por diversos programas, que requieran la funcionalidad de registrar los posibles errores o eventos que se produzcan durante la ejecución.

Para utilizar este módulo, será necesario primero llamar a `log.abrir()` para abrir el archivo de log, luego llamar a `log.guardar()` por cada mensaje que se quiera registrar, y finalmente llamar a `log.cerrar()` cuando se quiera concluir la registración de mensajes:

```
import log

archivo_log = log.abrir("log.txt")
# ...
# Hacer cosas que pueden dar error
if error:
    log.guardar(archivo_log, "ERROR: " + error)
# ...
# Finalmente cerrar el archivo
log.cerrar(archivo_log)
```

Código 11.1 `log.py`**:** Módulo para manipulación de archivos de log

```
# El módulo datetime se utiliza para obtener la fecha y hora actual.
import datetime

def abrir(nombre_log):
    """Abre el archivo de log indicado. Devuelve el archivo abierto.
    Pre: el nombre corresponde a un nombre de archivo válido.
    Post: el archivo ha sido abierto posicionándose al final."""
    archivo_log = open(nombre_log, "a")
    guardar_log(archivo_log, "Iniciando registro de errores")
    return archivo_log

def guardar(archivo_log, mensaje):
    """Guarda el mensaje en el archivo de log, con la hora actual.
    Pre: el archivo de log ha sido abierto correctamente.
    Post: el mensaje ha sido escrito al final del archivo."""
    # Obtiene la hora actual en formato de texto
    hora_actual = str(datetime.datetime.now())
    # Guarda la hora actual y el mensaje de error en el archivo
    archivo_log.write("[{}] {}\n".format(hora_actual, mensaje))

def cerrar(archivo_log):
    """ Cierra el archivo de log.
    Pre: el archivo de log ha sido abierto correctamente.
    Post: el archivo de log se ha cerrado. """
    guardar_log(archivo_log, "Fin del registro de errores")
    archivo_log.close()
```

Este código, que incluye el módulo `log` mostrado anteriormente, muestra una forma básica de utilizar un archivo de log. Al iniciarse el programa se abre el archivo de log, de forma que queda registrada la fecha y hora de inicio. Posteriormente se realizan tareas varias que podrían provocar errores, y de haber algún error se lo guarda en el archivo de log. Finalmente, al terminar el programa, se cierra el archivo de log, quedando registrada la fecha y hora de finalización.

El archivo de log generado tendrá la forma:

```
2016-04-10 15:20:32.229556 Iniciando registro de errores
2016-04-10 15:20:50.721415 ERROR: no se pudo acceder al recurso
2016-04-10 15:21:58.625432 ERROR: formato de entrada inválido
2016-04-10 15:22:10.109376 Fin del registro de errores
```

11.6 Manipular un archivo en forma binaria

No todos los archivos son archivos de texto, y por lo tanto no todos los archivos pueden ser procesados por líneas. Existen archivos en los que cada byte tiene un significado particular, y es necesario manipularlos conociendo el formato en que están los datos para poder procesar esa información.

Para abrir un archivo y manejarlo de forma binaria es necesario agregarle una `'b'` al parámetro de modo:

```
archivo_binario = open('imagen.jpg', 'rb')
```

Sabías que…

La `b` en el modo de apertura viene de *binario*, por el sistema de numeración binaria, ya que en el procesador de la computadora la información es manejada únicamente mediante ceros o unos (bits) que conforman números binarios.

Si bien no es necesaria en todos los sistemas (en general el mismo sistema detecta que es un archivo binario sin que se lo pidamos), es una buena costumbre usarla, por más que sirva principalmente como documentación.

Para procesar el archivo de a bytes en lugar de líneas, se utiliza la función `contenido = archivo.read(n)` para leer `n` bytes y `archivo.write(contenido)`, para escribir `contenido` en la posición actual del archivo.

Al manejar un archivo binario, es necesario poder conocer la posición actual en el archivo y poder modificarla. Para obtener la posición actual se utiliza `archivo.tell()`, que indica la cantidad de bytes desde el comienzo del archivo.

Para modificar la posición actual se utiliza `archivo.seek(posicion)`, que permite desplazarse hacia el byte ubicado en la posición indicada.

```
>>> f = open('imagen.jpg', 'rb')
>>> f.tell()
0
>>> datos = f.read(3)
>>> datos
b'\xff\xd8\xff'
>>> type(datos)
<class 'bytes'>
>>> f.tell()
3
>>> f.seek(0)
0
>>> datos = f.read()
>>> len(datos)
3150
```

Atención

Al trabajar con archivos binarios, la función `read` no devuelve cadenas de caracteres (`str`), sino que devuelve una *secuencia de bytes* (`bytes`). Análogamente, la función `write` recibe una secuencia de bytes.

11.7 Persistencia de datos

Se llama **persistencia** a la capacidad de guardar la información de un programa para poder volver a utilizarla en otro momento. Es lo que los usuarios conocen como *Guardar el archivo* y después *Abrir el archivo*. Pero para un programador puede significar más cosas y suele involucrar un proceso de *serialización* de los datos a un archivo o a una base de datos o a algún otro medio similar, y el proceso inverso de recuperar los datos a partir de la información *serializada*.

Por ejemplo, supongamos que en el desarrollo de un juego se quiere guardar en un archivo

la información referente a los ganadores, el puntaje máximo obtenido y el tiempo de juego en el que obtuvieron ese puntaje.

En el juego, esa información podría estar almacenada en una lista de tuplas:

```
[(nombre1, puntaje1, tiempo1), (nombre2, puntaje2, tiempo2), ...]
```

Esta información se puede guardar en un archivo de muchas formas distintas. En este caso, para facilitar la lectura del archivo de puntajes para los humanos, se decide guardarlos en un archivo de texto, donde cada tupla ocupará una línea y los valores de las tuplas estarán separados por comas.

En el Código 11.2 se muestra un módulo capaz de guardar y recuperar los puntajes en el formato especificado.

Código 11.2 `puntajes.py`**:** Módulo para guardar y recuperar puntajes en un archivo

```
def guardar_puntajes(nombre_archivo, puntajes):
    """Guarda la lista de puntajes en el archivo.
    Pre: nombre_archivo corresponde a un archivo válido,
         puntajes corresponde a una lista de tuplas de 3 elementos.
    Post: se guardaron los valores en el archivo,
          separados por comas.
    """

    with open(nombre_archivo, "w") as f:
        for nombre, puntaje, tiempo in puntajes:
            f.write("{},{},{}\n".format(nombre, puntaje, tiempo))

def recuperar_puntajes(nombre_archivo):
    """Recupera los puntajes a partir del archivo provisto.
    Devuelve una lista con los valores de los puntajes.
    Pre: el archivo contiene los puntajes en el formato esperado,
         separados por comas
    Post: la lista devuelta contiene los puntajes en el formato:
          [(nombre1,puntaje1,tiempo1),(nombre2,puntaje2,tiempo2)].
    """

    puntajes = []
    with open(nombre_archivo, "r") as f:
        for linea in f:
            nombre, puntaje, tiempo = linea.rstrip("\n").split(",")
            puntajes.append((nombre, int(puntaje), tiempo))
    return puntajes
```

Dadas las especificaciones del problema al guardar los valores en el archivo, es necesario convertir el puntaje (que es un valor numérico) en una cadena, y al abrir el archivo es necesario convertirlo nuevamente en un valor numérico.

> Es importante notar que tanto la función que almacena los datos como la que los recupera requieren que la información se encuentre de una forma determinada y de no ser así, fallarán. Es por eso que estas condiciones se indican en la documentación de las funciones como sus precondiciones. En próximas unidades veremos cómo evitar que falle una función si alguna de sus condiciones no se cumple.

Es bastate sencillo probar el módulo programado y ver que lo que se guarda es igual que lo que se recupera:

```
>>> import puntajes
>>> valores = [("Pepe", 108, "4:16"), ("Juana", 2315, "8:42")]
>>> puntajes.guardar_puntajes("puntajes.txt", valores)
>>> recuperado = puntajes.recuperar_puntajes("puntajes.txt")
>>> print(recuperado)
[('Pepe', 108, '4:16'), ('Juana', 2315, '8:42')]
```

Guardar el estado de un programa se puede hacer tanto en un archivo de texto, como en un archivo binario. En muchas situaciones es preferible guardar la información en un archivo de texto, ya que de esta manera es posible modificarlo fácilmente desde cualquier editor de textos.

En general, los archivos de texto consumen más espacio, pero son más faciles de entender y fáciles de usar desde cualquier programa.

Por otro lado, en un archivo binario bien definido se puede evitar el desperdicio de espacio, o también hacer que sea más eficiente acceder a los datos.

En definitiva, la decisión de qué formato usar queda a discreción del programador. Es importante recordar que el sentido común es el valor más preciado en un programador.

11.7.1 Persistencia en archivos CSV

Un formato que suele usarse para transferir datos entre programas es **csv** (del inglés *comma separated values*: valores separados por comas). Es un formato bastante sencillo, tanto para leerlo como para procesarlo desde el código, parecido al formato visto en el ejemplo anteriormente.

```
Nombre,Apellido,Telefono,Cumpleaños
"John","Smith","555-0101","1973-11-24"
"Jane","Smith","555-0101","1975-06-12"
```

En el ejemplo se puede ver una pequeña base de datos. En la primera línea del archivo tenemos los nombres de los campos, un dato opcional desde el punto de vista del procesamiento de la información, pero que facilita entender el archivo.

En las siguientes lineas se ingresan los datos de la base de datos, cada campo separado por comas. Los campos que son cadenas se suelen escribir entre comillas dobles. Si alguna cadena contiene alguna comilla doble se la reemplaza por `\"` y una contrabarra se escribe como `\\`.

En Python es bastante sencillo procesar de este tipo de archivos, tanto para la lectura como para la escritura, mediante el módulo `csv`.

La funciones del ejemplo anterior podrían programarse mediante el módulo `csv`. En el Código 11.3 se muestra una posible implementación que utiliza este módulo. Si se prueba este código, se obtiene un resultado idéntico al obtenido anteriormente.

El código en este caso es muy similar, ya que en el ejemplo original se hacían muy pocas consideraciones al respecto de los valores: se asumía que el primero y el tercero eran cadenas mientras que el segundo necesitaba ser convertido a cadena.

> Es importante notar, entonces, que al utilizar el módulo `csv` en lugar de hacer el procesamiento en forma manual, se obtiene un comportamiento más robusto, ya que el módulo `csv` tiene en cuenta muchos más casos que nuestro código original no. Por ejemplo, el código anterior no tenía en cuenta que el nombre pudiera contener una coma.

En el apéndice de esta unidad puede verse una aplicación completa de una agenda, que almacena los datos del programa en archivos csv.

Código 11.3 `puntajes_csv.py`**:** Módulo para guardar y recuperar puntajes en un archivo que usa csv

```
import csv

def guardar_puntajes(nombre_archivo, puntajes):
    """Guarda la lista de puntajes en el archivo.
    Pre: nombre_archivo corresponde a un archivo válido,
         puntajes corresponde a una lista de secuencias de elementos.
    Post: se guardaron los valores en el archivo,
          separados por comas.
    """

    with open(nombre_archivo, "w") as f:
        archivo_csv = csv.writer(f)
        archivo_csv.writerows(puntajes)

def recuperar_puntajes(nombre_archivo):
    """Recupera los puntajes a partir del archivo provisto.
    Devuelve una lista con los valores de los puntajes.
    Pre: el archivo contiene los puntajes en el formato esperado,
         separados por comas
    Post: la lista devuelta contiene los puntajes en el formato:
          [(nombre1,puntaje1,tiempo1),(nombre2,puntaje2,tiempo2)].
    """

    puntajes = []
    with open(nombre_archivo, "r") as f:
        archivo_csv = csv.reader(f)
        for nombre, puntaje, tiempo in archivo_csv:
            puntajes.append((nombre, int(puntaje), tiempo))
    return puntajes
```

11.7.2 Persistencia en archivos binarios

En el caso de que decidiéramos grabar los datos en un archivo binario, Python incluye una herramienta llamada *pickle* que permite hacerlo de forma muy sencilla. Hay que tener en cuenta, sin embargo, que no es nada simple acceder a un archivo en este formato desde un programa que no esté escrito en Python.

En el Código 11.4 se muestra el mismo ejemplo de almacenamiento de puntajes, utilizando el módulo `pickle`.

El funcionamiento de este programa será idéntico a los anteriores. Pero el archivo generado será muy distinto a los archivos generados anteriormente. En lugar de ser un archivo de fácil lectura, tendrá un contenido binario ilegible para un humano:

```
..]q.(X....Pepeq
.KlX....4:16q..q
.X....Juanaq.M..
X....8:42q..q.e.
```

En el apéndice de esta unidad puede verse una aplicación completa de una agenda, que utiliza *pickle* para almacenar datos en archivos.

Código 11.4 `puntajes_pickle.py`**:** Módulo para guardar y recuperar puntajes en un archivo que usa *pickle*

```
import pickle

def guardar_puntajes(nombre_archivo, puntajes):
    """Guarda la lista de puntajes en el archivo.
    Pre: nombre_archivo corresponde a un archivo válido,
         puntajes corresponde a los valores a guardar
    Post: se guardaron los valores en el archivo en formato pickle.
    """

    with open(nombre_archivo, "wb") as f:
        pickle.dump(puntajes, f)

def recuperar_puntajes(nombre_archivo):
    """Recupera los puntajes a partir del archivo provisto.
    Devuelve una lista con los valores de los puntajes.
    Pre: el archivo contiene los puntajes en formato pickle
    Post: la lista devuelta contiene los puntajes en el
          mismo formato que se los almacenó.
    """

    with open(nombre_archivo, "r") as f:
        return pickle.load(f)
```

11.8 Directorios

Hasta aquí se ha mostrado el acceso a los archivos utilizando sólo el nombre del archivo. Esto nos permite acceder a los archivos en el directorio actual donde corre el programa.

Un problema relacionado con la utilización de directorios es que los separadores de directorios en distintos sistemas son distintos: `/` en Unix y OSX, y `\` en Windows. La manera de acceder a directorios independientemente del sistema en el que estamos desde Python es usando el modulo `os`.

Por ejemplo, si queremos acceder al archivo `archivo.csv` que se encuentra en el directorio `data`:

```
ruta = os.path.join("data", "archivo.csv")
with open(ruta) as f:
    ...
```

11.9 Resumen

- Para utilizar un archivo desde un programa, es necesario abrirlo, y cuando ya no se lo necesite, se lo debe cerrar.
- Las instrucciones más básicas para manejar un archivo son leer y escribir.
- Cada archivo abierto tiene relacionada una posición que se puede consultar o cambiar.
- Los archivos de texto se procesan generalmente línea por línea y sirven para intercambiar información entre diversos programas o entre programas y humanos.

- Los archivos binarios se procesan generalmente byte por byte. Suelen ser más eficientes al ser interpretados por una computadora, pero son ilegibles para humanos.
- Es posible acceder de forma secuencial a los datos, o se puede ir accediendo a posiciones en distintas partes del archivo, dependiendo de cómo esté almacenada la información y qué se quiera hacer con ella.
- Es posible leer todo el contenido de un archivo y almacentarlo en una única variable, pero si el archivo es muy grande puede consumir memoria innecesariamente.

Referencia Python

`archivo = open(nombre[, modo])`

Abre un archivo. `nombre` es el nombre completo del archivo, `modo` especifica si se va usar para lectura (`'r'`), escritura truncando el archivo (`'w'`), o escritura agregando al final del archivo (`'a'`), agregándole un `'+'` al modo el archivo se abre en lectura-escritura, agregándole una `'b'` el archivo se maneja como archivo binario, agregándole `'U'` los fin de línea se manejan a mano.

`archivo.close()`

Cierra el archivo.

`with open(nombre) as archivo:`

Abre el archivo para procesar dentro del bloque `with`. El archivo se cerrará automáticamente al salir del bloque.

`linea = archivo.readline()`

Lee una línea de texto del archivo
Si la cadena devuelta es vacía, es que se ha llegado al final del archivo.

`for linea in archivo:`

Itera sobre las lineas del archivo.

`lineas = archivo.readlines()`

Devuelve una lista con todas las líneas del archivo.

`datos = archivo.read([n])`

Si se trata de un archivo de texto, devuelve la cadena de `n` caracteres situada en la posición actual del archivo.
Si se trata de un archivo binario, devuelve una secuencia de `n` bytes.
Si la secuencia devuelta es vacía, es que se ha llegado al final del archivo.
De omitirse el parámetro `n`, devuelve todo el contenido del archivo.

`archivo.write(contenido)`

Escribe `contenido` en la posición actual de `archivo`.

`posicion = archivo.tell()`

Devuelve un número que indica la posición actual en `archivo`, equivalente a la cantidad de bytes desde el comienzo del archivo.

`archivo.seek(posicion)`

Modifica la posición actual en `archivo`, trasladándose hasta el byte `posicion`.

`os.path.exists(ruta)`

Indica si la ruta existe o no. No nos dice si es un directorio, un archivo u otro tipo de archivo especial del sistema.

`os.path.isfile(ruta)`

Indica si la ruta existe y es un archivo.

`os.path.isdir(ruta)`

Indica si la ruta existe y es un directorio.

`os.path.join(ruta, ruta1[, ... rutaN])`

Une las rutas con el caracter de separación de directorios que le corresponda al sistema en uso.

11.10 Ejercicios

Ejercicio 11.10.1. Escribir un programa, llamado **head** que reciba un archivo y un número `N` e imprima las primeras `N` líneas del archivo.

Ejercicio 11.10.2. Escribir un programa, llamado **cp.py**, que copie todo el contenido de un archivo (sea de texto o binario) a otro, de modo que quede exactamente igual.
Nota: utilizar `archivo.read(bytes)` para leer como máximo una cantidad de bytes.

Ejercicio 11.10.3. Escribir un programa, llamado **cut.py**, que dado un archivo de texto, un delimitador, y una lista de campos, imprima solamente esos campos, separados por ese delimitador.

Ejercicio 11.10.4. Escribir un programa, llamado **wc.py** que reciba un archivo, lo procese e imprima por pantalla cuántas líneas, cuantas palabras y cuántos caracteres contiene el archivo.

Ejercicio 11.10.5. Escribir un programa, llamado **grep.py** que reciba una expresión y un archivo e imprima las líneas del archivo que contienen la expresión recibida.

Ejercicio 11.10.6. Escribir un programa, llamado **rot13.py** que reciba un archivo de texto de origen y uno de destino, de modo que para cada línea del archivo origen, se guarde una línea *cifrada* en el archivo destino. El algoritmo de cifrado a utilizar será muy sencillo: a cada caracter comprendido entre la a y la z, se le suma 13 y luego se aplica el módulo 26, para obtener un nuevo caracter.

Ejercicio 11.10.7. Persistencia de un diccionario

a) Escribir una función `cargar_datos` que reciba un nombre de archivo, cuyo contenido tiene el formato `clave, valor` y devuelva un diccionario con el primer campo como clave y el segundo como diccionario.

b) Escribir una función `guardar_datos` que reciba un diccionario y un nombre de archivo, y guarde el contenido del diccionario en el archivo, con el formato `clave, valor`.

Apéndice 11.A Agenda con archivos CSV

A continuación se muestra un programa de agenda que utiliza archivos CSV. Luego se muestran los cambios necesarios para que la agenda que utilice archivos en formato *pickle*, en lugar de CSV.

`agenda-csv.py` Agenda con los datos en formato CSV

```
import csv, os

ARCHIVO = "agenda.csv"
CAMPOS = ("Nombre", "Apellido", "Telefono", "Cumpleaños")

def cargar_agenda(archivo):
    """Carga todos los datos del archivo en una lista y la devuelve."""
    agenda = []
    if not os.path.exists(archivo):
        return agenda
    with open(archivo) as f:
        datos_csv = csv.reader(f)
        encabezado = next(datos_csv)
        for item in datos_csv:
            agenda.append(item)
    return agenda

def guardar_agenda(agenda, archivo):
    """Guarda la agenda en el archivo."""
    with open(archivo, "w") as f:
        datos_csv = csv.writer(f)
        datos_csv.writerow(CAMPOS)
        datos_csv.writerows(agenda)

def leer_busqueda():
    """Solicita al usuario nombre y apellido y los devuelve."""
    nombre = input("Nombre: ")
    apellido = input("Apellido: ")
    return nombre, apellido

def buscar(nombre, apellido, agenda):
    """Busca el primer item que coincida con nombre y con apellido."""
    for item in agenda:
        if nombre in item[0] and apellido in item[1]:
            return item
    return None

def menu_alta(nombre, apellido, agenda):
    """Pregunta si se desea ingresar un nombre y apellido y
       de ser así, pide los datos al usuario."""
    print("No se encuentra {} {} en la agenda.".format(nombre, apellido))
    confirmacion = input("¿Desea ingresarlo? (s/n): ")
    if confirmacion.lower() != "s":
        return
    telefono = input("Telefono: ")
    cumple = input("Cumpleaños: ")
```

```
47      agenda.append([nombre, apellido, telefono, cumple])
48
49  def mostrar_item(item):
50      """Muestra por pantalla un item en particular."""
51      nombre, apellido, telefono, cumple = item
52      print()
53      print("{} {}".format(nombre, apellido))
54      print("Telefono: {}".format(telefono))
55      print("Cumpleaños: {}".format(cumple))
56      print()
57
58  def menu_item():
59      """Muestra por pantalla las opciones disponibles para un item
60         existente."""
61      o = input("b: borrar, m: modificar, ENTER para continuar (b/m): ")
62      return o.lower()
63
64  def modificar(viejo, nuevo, agenda):
65      """Reemplaza el item viejo con el nuevo, en la lista datos."""
66      indice = agenda.index(viejo)
67      agenda[indice] = nuevo
68
69  def menu_modificacion(item, agenda):
70      """Solicita al usuario los datos para modificar una entrada."""
71      nombre = input("Nuevo nombre: ")
72      apellido = input("Nuevo apellido: ")
73      telefono = input("Nuevo teléfono: ")
74      cumple = input("Nuevo cumpleaños: ")
75      modificar(item, [nombre, apellido, telefono, cumple], agenda)
76
77  def baja(item, agenda):
78      """Elimina un item de la lista."""
79      agenda.remove(item)
80
81  def confirmar_salida():
82      """Solicita confirmación para salir"""
83      confirmacion = input("¿Desea salir? (s/n): ")
84      return confirmacion.lower() == "s"
85
86  def agenda():
87      """Función principal de la agenda.
88      Carga los datos del archivo, permite hacer búsquedas, modificar
89      borrar, y al salir guarda. """
90      agenda = cargar_agenda(ARCHIVO)
91      while True:
92          nombre, apellido = leer_busqueda()
93          if nombre + apellido == "":
94              if confirmar_salida():
95                  break
96          item = buscar(nombre, apellido, agenda)
97          if not item:
98              menu_alta(nombre, apellido, agenda)
99              continue
100         mostrar_item(item)
```

```
        opcion = menu_item()
        if opcion == "m":
            menu_modificacion(item, agenda)
        elif opcion == "b":
            baja(item, agenda)
    guardar_agenda(agenda, ARCHIVO)

agenda()
```

`agenda-pickle.py` Modificaciones a la agenda para guardar los datos en formato *pickle*

```
import pickle, os

ARCHIVO = "agenda.dat"

def cargar_agenda(archivo):
    """Carga todos los datos del archivo en una lista y la devuelve."""
    if not os.path.exists(archivo):
        return []
    with open(archivo) as f:
        return pickle.load(f)

def guardar_agenda(datos, archivo):
    """Guarda la agenda en el archivo."""
    with open(archivo, "wb") as f:
        pickle.dump(archivo, datos)
```

Unidad 12

Manejo de errores y excepciones

12.1 Errores

En un programa podemos encontrarnos con distintos tipos de errores, pero a grandes rasgos podemos decir que todos los errores pertenecen a una de las siguientes categorías.

- Errores de sintaxis: estos errores son seguramente los más simples de resolver, pues son detectados por el intérprete (o por el compilador, según el tipo de lenguaje que estemos utilizando) al procesar el código fuente y generalmente son consecuencia de equivocaciones al escribir el programa. En el caso de Python estos errores son indicados con un mensaje *SyntaxError*. Por ejemplo, si trabajando con Python intentamos definir una función y en lugar de `def` escribimos `dev`.
- Errores semánticos: se dan cuando un programa, a pesar de no generar mensajes de error, no produce el resultado esperado. Esto puede deberse, por ejemplo, a un algoritmo incorrecto o a la omisión de una sentencia.
- Errores de ejecución: estos errores aparecen durante la ejecución del programa y su origen puede ser diverso. En ocasiones pueden producirse por un uso incorrecto del programa por parte del usuario, por ejemplo si el usuario ingresa una cadena cuando se espera un número. En otras ocasiones pueden deberse a errores de programación, por ejemplo si una función intenta acceder a la quinta posición de una lista de 3 elementos o realizar una división por cero. Una causa común de errores de ejecución, que generalmente excede al programador y al usuario, son los recursos externos al programa, por ejemplo si el programa intenta leer un archivo y el mismo se encuentra dañado.

Tanto a los errores de sintaxis como a los semánticos se los puede detectar y corregir durante la construcción del programa ayudados por el intérprete y la ejecución de pruebas. Pero no ocurre esto con los errores de ejecución, ya que no siempre es posible saber cuándo ocurrirán y puede resultar muy complejo (o incluso casi imposible) reproducirlos. Es por ello que el resto de la unidad nos centraremos en cómo preparar nuestros programas para lidiar con este tipo de errores.

12.2 Excepciones

Los errores de ejecución son llamados comúnmente *excepciones* y por eso de ahora en más utilizaremos ese nombre. Durante la ejecución de un programa, cualquier línea de código puede

generar una excepción. Se dice también que la línea *levanta* o *lanza* una excepción:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

En este caso la línea `1 / 0` lanzó una excepción de tipo `ZeroDivisionError`. A continuación se listan lgunos tipos de excepciones comunes:

Tipo	Significado
`Exception`	Excepción genérica. Todas las excepciones son de tipo `Exception`
`AssertionError`	Una instrucción `assert` falló
`IndexError`	Se intentó acceder a una secuencia con un índice fuera de rango
`KeyError`	Se intentó acceder a un diccionario con una clave inexistente
`TypeError`	Se aplicó una operación a un valor de tipo inapropiado
`ValueError`	Se aplicó una operación con un parámetro de tipo apropiado pero no así su valor.
`ZeroDivisionError`	Se intentó dividir un número por 0
`IOError`	Error de entrada / salida (por ejemplo al intentar acceder a un archivo)

Podemos lanzar una excepción de un tipo arbitrario utilizando la instrucción `raise`:

```
>>> raise Exception()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception
```

También podemos agregarle un mensaje a la excepción, para darle más información acerca de cuál fue la causa del error:

```
>>> raise Exception("Algo salió mal :(")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Algo salió mal :(
```

Si dentro de una función se lanza una excepción y la función no la *maneja*, la excepción se propaga hacia la función que la invocó; si esta otra tampoco la maneja, la excepción continúa propagándose hasta llegar a la función inicial del programa, y si ésta tampoco la maneja se interrumpe la ejecución del programa.

12.2.1 Manejo de excepciones

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa.

En el caso de Python, el manejo de excepciones se hace mediante los bloques que utilizan las sentencias `try`, `except` y `finally`.

Dentro del bloque `try` se ubica todo el código que pueda llegar a levantar una excepción.

A continuación se ubica el bloque `except`, que se encarga de *capturar* la excepción y nos da la oportunidad de procesarla mostrando por ejemplo un mensaje adecuado al usuario.

Veamos qué sucede si se quiere realizar una división por cero:

```
>>> dividendo = 5
>>> divisor = 0
>>> dividendo / divisor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

En este caso, se levantó la excepción `ZeroDivisionError` cuando se quiso hacer la división. Para evitar que se levante la excepción y se detenga la ejecución del programa, se utiliza el bloque `try-except`.

```
>>> try:
...     cociente = dividendo / divisor
... except:
...     print("No se permite la división por cero")
...
No se permite la división por cero
```

Dado que dentro de un mismo bloque `try` pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques `except`, cada uno para capturar un tipo distinto de excepción.

Esto se hace especificando a continuación de la sentencia `except` el nombre de la excepción que se pretende capturar. Un mismo bloque `except` puede atrapar varios tipos de excepciones, lo cual se hace especificando los nombres de la excepciones separados por comas a continuación de la palabra `except`. Es importante destacar que si bien luego de un bloque `try` puede haber varios bloques `except`, se ejecutará, a lo sumo, uno de ellos.

```
try:
    # aquí ponemos el código que puede lanzar excepciones
except IOError:
    # entrará aquí en caso que se haya producido
    # una excepción IOError
except ZeroDivisionError:
    # entrará aquí en caso que se haya producido
    # una excepción ZeroDivisionError
except:
    # entrará aquí en caso que se haya producido
    # una excepción que no corresponda a ninguno
    # de los tipos especificados en los casos previos
```

Como se muestra en el ejemplo precedente también es posible utilizar una sentencia `except` sin especificar el tipo de excepción a capturar, en cuyo caso se captura cualquier excepción, sin importar su tipo. Cabe destacar también, que en caso de utilizar una sentencia `except` sin especificar el tipo, la misma debe ser siempre la última de las sentencias `except`. Por ejemplo, el siguiente fragmento de código es incorrecto.

```
try:
    # aquí ponemos el código que puede lanzar excepciones
except:
    # ERROR de sintaxis, esta sentencia no puede estar aquí,
    # sino que debería estar luego del except IOError.
except IOError:
    # Manejo de la excepción de entrada/salida
```

Finalmente, puede ubicarse un bloque `finally` donde se escriben las sentencias de finalización, que son típicamente acciones de limpieza. La particularidad del bloque `finally` es que se ejecuta siempre, haya surgido una excepción o no. Si hay un bloque `except`, no es necesario que esté presente el `finally`, y es posible tener un bloque `try` sólo con `finally`, sin `except`.

Veamos ahora cómo es que actúa Python al encontrarse con estos bloques. Python comienza a ejecutar las instrucciones que se encuentran dentro de un bloque `try` normalmente. Si durante la ejecución de esas instrucciones se levanta una excepción, Python interrumpe la ejecución en el punto exacto en que surgió la excepción y pasa a la ejecución del bloque `except` correspondiente.

Para ello, Python verifica uno a uno los bloques `except` y si encuentra alguno cuyo tipo haga referencia al tipo de excepción levantada, comienza a ejecutarlo. Sino encuentra ningún bloque del tipo correspondiente pero hay un bloque `except` sin tipo, lo ejecuta. Al terminar de ejecutar el bloque correspondiente, se pasa a la ejecución del bloque `finally`, si se encuentra definido.

Si, por otra parte, no hay problemas durante la ejecución del bloque `try`, se completa la ejecución del bloque, y luego se pasa directamente a la ejecución del bloque `finally` (si es que está definido).

12.2.2 Manejo de excepciones con archivos

Bajemos todo esto a un ejemplo concreto. Supongamos que nuestro programa tiene que procesar cierta información ingresada por el usuario y guardarla en un archivo. Dado que el acceso a archivos puede levantar excepciones, si no queremos que nuestro programa se interrumpa deberíamos colocar el código de manipulación de archivos dentro de un bloque `try`. Luego deberíamos colocar un bloque `except` que atrape una excepción del tipo `IOError`, que es el tipo de excepciones que lanzan la funciones de manipulación de archivos. Adicionalmente podríamos agregar un bloque `except` sin tipo por si surge alguna otra excepción. Finalmente deberíamos agregar un bloque `finally` para cerrar el archivo, haya surgido o no una excepción.

```
archivo = None
try:
    archivo = open("miarchivo.txt")
    # procesar el archivo
except IOError:
    print("Error de entrada/salida.")
    # realizar procesamiento adicional
except:
    # procesar la excepción
finally:
    # si el archivo está abierto debemos cerrarlo
    if archivo and not archivo.closed:
        archivo.close()
```

Notar que en la primera línea inicializamos la variable `archivo` con el valor `None`, y luego en el bloque `finally` preguntamos si la variable contiene un archivo con `if archivo`, y además si el archivo está abierto con `not archivo.closed`. ¿Por qué es necesario? ¿No se podrá hacer el manejo de error una manera más simple?:

```
try:
    archivo = open("miarchivo.txt")
    # procesar el archivo
...
finally:
    archivo.close() ← ERROR: 'archivo' puede no estar definido
```

Si la llamada a `open` falla, la variable `archivo` nunca se inicializa, por lo tanto no existe cuando se llama a `archivo.close()`. El manejo de error en este caso no es correcto.

Funcionamiento del bloque `with`

Recordemos que podemos abrir un archivo y cerrarlo automáticamente con el bloque `with`:

```
with open("miarchivo.txt") as archivo:
    # procesar el archivo
```

El bloque `with` nos garantiza que el archivo se cierra, independientemente de si se levanta una excepción o no. De hecho, el bloque es equivalente a la siguiente estructura `try-finally`:

```
archivo = open("miarchivo.txt")
try:
    # procesar el archivo
finally:
    archivo.close()
```

Notar que hay ningún bloque `except`. Esto significa que el bloque `with` no atrapa ninguna excepción. Si queremos garantizar que se cierra el archivo y además atrapar posibles excepciones, podemos hacer algo como:

```
try:
    with open("miarchivo.txt") as archivo:
        # procesar el archivo
except IOError:
    # manejar la excepción
```

12.2.3 Procesamiento y propagación de excepciones

Ya sabemos cómo atrapar excepciones, pero ¿qué se supone que tenemos que hacer una vez que atrapamos una excepción?. En primer lugar podríamos ejecutar alguna lógica particular del caso como: cerrar un archivo, realizar una procesamiento alternativo al del bloque `try`, etc. Pero más allá de esto tenemos algunas opciones genéricas que consisten en: dejar constancia de la ocurrencia de la excepción, propagar la excepción o, incluso, hacer ambas cosas.

Para dejar constancia de la ocurrencia de la excepción, se puede escribir en un archivo de *log* o simplemente mostrar un mensaje en pantalla. Generalmente cuando se deja constancia de la ocurrencia de una excepción se suele brindar alguna información del contexto en que ocurrió la excepción, por ejemplo: tipo de excepción ocurrida, momento en que ocurrió la excepción y cuáles fueron las llamadas previas a la excepción. El objetivo de esta información es facilitar el diagnóstico en caso de que alguien deba corregir el programa para evitar que la excepción siga apareciendo.

Es posible, por otra parte, que luego de realizar algún procesamiento particular del caso se quiera que la excepción se propague hacia la función que había invocado a la función actual. Podemos hacer esto utilizando la instrucción `raise`: si se invoca esta instrucción dentro de un bloque `except`, sin pasarle parámetros, Python levantará la excepción atrapada por ese bloque:

```
try:
    with open("miarchivo.txt") as archivo:
        # procesar el archivo
except IOError:
    print("Error al acceder al archivo")
    raise # Propagar la excepción (como si no la hubiéramos atrapado)
```

12.2.4 Acceso a información de contexto

Para obtener más información acerca de la excepción atrapada, se puede utilizar la misma sentencia `except`, pasándole un identificador para que almacene una referencia a la excepción.

```
>>> try:
...     x = 1 / 0
... except ZeroDivisionError as e:
...     print('Excepción atrapada: ', e)
...
Excepción atrapada: int division or modulo by zero
```

Sabías que…

En otros lenguajes, por ejemplo Java, si una función puede lanzar una excepción en alguna situación, la o las excepciones que lance deben formar parte de la declaración de la función y quien invoque dicha función está obligado a hacerlo dentro de un bloque `try` que la atrape.

En Python, al no tener esta obligación por parte del lenguaje debemos tener cuidado de atrapar las excepciones probables, ya que de no ser así los programas se terminarán inesperadamente.

12.3 Validaciones

Las validaciones son técnicas que permiten asegurar que los valores con los que se vaya a operar estén dentro de determinado dominio.

Estas técnicas son particularmente importantes al momento de utilizar entradas del usuario o de un archivo (o entradas externas en general) en nuestro código, y también se las utiliza para comprobar precondiciones. Al uso intensivo de estas técnicas se lo suele llamar *programación defensiva*.

Si bien quien invoca una función debe preocuparse de cumplir con las precondiciones de ésta, si las validaciones están hechas correctamente pueden devolver información valiosa para que el invocante pueda actuar en consecuencia.

Hay distintas formas de comprobar el dominio de un dato. Se puede comprobar el contenido; que una variable sea de un tipo en particular; o que el dato tenga determinada característica, como por ejemplo que sea una secuencia.

También se debe tener en cuenta qué hará nuestro código cuando una validación falle, ya que queremos darle información al invocante que le sirva para procesar el error. El error producido tiene que ser fácilmente reconocible. En algunos casos, como por ejemplo cuando se quiere devolver una posición, devolver `-1` nos puede asegurar que el invocante lo vaya a reconocer. En otros casos, levantar una excepción es una solución más elegante.

En cualquier caso, lo importante es que el resultado generado por nuestro código cuando funciona correctamente y el resultado generado cuando falla debe ser claramente distinto. Por ejemplo, si una función debe devolver un elemento de una secuencia, no es una buena idea que devuelva `None` en el caso de que la secuencia esté vacía, ya que `None` es un elemento válido dentro de una secuencia.

12.3.1 Comprobaciones por contenido

Cuando queremos validar que los datos provistos a una porción de código contengan la información apropiada, ya sea porque esa información la ingresó un usuario, fue leída de un archivo, o porque por cualquier motivo es posible que sea incorrecta, es deseable comprobar que el contenido de las variables a utilizar estén dentro de los valores con los que se puede operar.

Estas comprobaciones no siempre son posibles, ya que en ciertas situaciones puede ser muy costoso corroborar las precondiciones de una función. Es por ello que este tipo de comprobaciones se realizan sólo cuando sea posible.

Por ejemplo, la función factorial está definida para los números naturales incluyendo el 0. Es posible utilizar `assert` (que es otra forma de levantar una excepción) para comprobar las precondiciones de factorial.

```
def factorial(n):
    """Calcula el factorial de n.
    Pre: n debe ser un entero, mayor igual a 0
    Post: se devuelve el valor del factorial pedido
    """
    assert n >= 0, "n debe ser mayor igual a 0"
    fact = 1
    for i in range(2, n + 1):
        fact *= i
    return fact
```

12.3.2 Entrada del usuario

En el caso particular de una porción de código que trate con entrada del usuario, no se debe asumir que el usuario vaya a ingresar los datos correctamente, ya que los seres humanos tienden a cometer errores al ingresar información.

Por ejemplo, si se desea que un usuario ingrese un número:

```
def pedir_entero():
    """Solicita un valor entero y lo devuelve.
    Si el valor ingresado no es entero, lanza una excepción.
    """
    return int(input("Ingrese un número entero: "))
```

Esta función devuelve un valor entero, o lanza una excepción si la conversión no fue posible (la excepción la lanza la función `int`, y al no atraparla, se propaga automáticamente).

Sin embargo, esto no es satisfactorio: si el usuario no ingresa la información correctamente, ¡el programa se interrumpe!. Podemos hacerlo más *amigable* atrapando la excepción y haciendo que se vuelva a pedir al usuario que ingrese la información:

```
def pedir_entero():
    """Solicita un valor entero y lo devuelve.
    Mientras el valor ingresado no sea entero, vuelve a solicitarlo."""
    while True:
        valor = input("Ingrese un número entero: ")
        try:
            return int(valor)
        except ValueError:
            print("'{}' no es un número entero.".format(valor))
```

Podría ser deseable, además, poner un límite a la cantidad máxima de intentos que el usuario tiene para ingresar la información correctamente y, superada esa cantidad máxima de intentos, levantar una excepción para que sea manejada por el código invocante.

```
def pedir_entero():
    """Solicita un valor entero y lo devuelve.
    Si el valor ingresado no es entero, da 5 intentos para ingresarlo
    correctamente, y de no ser así, lanza una excepción."""
    intentos = 0
    while intentos < 5:
        valor = input("Ingrese un número entero: ")
        try:
            return int(valor)
        except ValueError:
            print("'{}' no es un número entero.".format(valor))
            intentos += 1
    raise ValueError("Valor incorrecto ingresado en 5 intentos")
```

Por otro lado, cuando la entrada ingresada sea una cadena, no es esperable que el usuario la vaya a ingresar en mayúsculas o minúsculas; ambos casos deben ser considerados.

```
def lee_opcion():
    """Solicita una opción de menú y la devuelve."""
    while True:
        opcion = input("Ingrese A (Altas) - B (Bajas) - M (Modificaciones): ")
        if opcion.upper() in ("A", "B", "M"):
            return opcion
```

12.3.3 Comprobaciones por tipo

En esta clase de comprobaciones nos interesa el tipo del dato que vamos a tratar de validar. Python nos indica el tipo de una variable usando la función `type`. Por ejemplo, para comprobar que una variable contenga un tipo entero podemos hacer:

```
if type(i) is not int:
    raise TypeError("i debe ser del tipo int")
```

Sin embargo, ya hemos visto que tanto las listas como las tuplas y las cadenas son secuencias, y muchas de las funciones utilizadas puede utilizar cualquiera de estas secuencias. De la misma manera, una función puede utilizar un valor numérico, y que opere correctamente ya sea entero, flotante, o complejo.

Es posible comprobar el tipo de nuestra variable contra una secuencia de tipos posibles.

```
if type(i) not in (int, float, complex):
    raise TypeError("i debe ser numérico")
```

Si bien esto es bastante más flexible que el ejemplo anterior, también puede ser restrictivo ya que -como se verá más adelante- cada programador puede definir sus propios tipos utilizando como base los que ya están definidos. Con este código se están descartando todos los tipos que se basen en `int`, `float` o `complex`.

Para poder incluir estos tipos en la comprobación a realizar, Python nos provee de la función `isinstance(variable, tipos)`.

```
if not isinstance(i, (int, float, complex)):
    raise TypeError("i debe ser numérico")
```

Con esto comprobamos si una variable es de determinado tipo o subtipo de éste. Esta opción es bastante flexible, pero existen aún más opciones.

> **Atención**
>
> Hacer comprobaciones sobre los tipos de las variables suele resultar demasiado restrictivo, ya que es muy posible que una porción de código que opere con un tipo en particular funcione correctamente con otros tipos de variables que se comporten de forma similar.
>
> Es por eso que hay que tener mucho cuidado al limitar el uso de una variable por su tipo, y en muchos casos es preferible limitarlas por sus propiedades.

Para la mayoría de los tipos básicos de Python existe una función que se llama de la misma manera que el tipo que devuelve un elemento de ese tipo, por ejemplo, `int()` devuelve 0, `dict()` devuelve `{}` y así. Además, estas funciones suelen poder recibir un elemento de otro tipo para tratar de convertirlo, por ejemplo, `int(3.0)` devuelve 3, `list("Hola")` devuelve `['H', 'o', 'l', 'a']`.

Usando está conversión conseguimos dos cosas: podemos convertir un tipo recibido al que realmente necesitamos, a la vez que tenemos una copia de este, dejando el original intacto, que es importante cuando estamos tratando con tipos mutables.

Por ejemplo, si se quiere contar con una función de división entera que pueda recibir diversos parámetros, podría hacerse de la siguiente manera.

```
def division_entera(x, y):
    """Calcula la división entera después de convertir los parámetros a
    enteros."""
    return int(x) // int(y)
```

De esta manera, la función `division_entera` puede ser llamada incluso con cadenas que contengan expresiones enteras. Que este comportamiento sea deseable o no, depende siempre de cada caso.

```
>>> division_entera("5", 4.3)
1
>>> division_entera(5, None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in division_entera
TypeError: int() argument must be a string, a bytes-like object or a number,
not 'NoneType'
```

12.3.4 Comprobaciones por características

Otra posible comprobación, dejando de lado los tipos, consiste en verificar si una variable tiene determinada característica o no. Python promueve este tipo de programación, ya que el mismo intérprete utiliza este tipo de comprobaciones. Por ejemplo, para imprimir una variable, Python convierte esa variable a una cadena; no hay en el intérprete una verificación para cada tipo, sino que busca una función especial, llamada `__str__`, en la variable a imprimir, y si existe, la utiliza para convertir la variable a una cadena.

Para comprobar si una variable tiene o no una función Python provee la función `hasattr(variable, atributo)`, donde `atributo` puede ser el nombre de la función o de la variable que se quiera verificar. Se verá más sobre atributos en la Unidad 14.

Sabías que…

Python promueve la idea de *duck typing*:

> *Si algo parece un pato, camina como un pato y grazna como un pato, entonces, se lo puede considerar un pato.*

Esto se refiere a no diferenciar las variables por los tipos a los que pertenecen, sino por las funciones que tienen.

Por ejemplo, existe la función `__add__` para realizar operaciones de suma entre elementos. Si se quiere corroborar si un elemento es sumable, se lo haría de la siguiente forma.

```
def hacer_algo_con(x, y):
    if not hasattr(x, "__add__"):
        raise TypeError("x no es sumable")
    if not hasattr(y, "__add__"):
        raise TypeError("y no es sumable")
    z = x + y
    ...
```

Sin embargo, que el atributo exista no quiere decir que vaya a funcionar correctamente en todos los casos. Por ejemplo, tanto las cadenas como los números definen su propia "suma", pero no es posible sumar cadenas y números, de modo que en este caso sería necesario tener en cuenta una posible excepción.

Por otro lado, en la mayoría de los casos se puede aplicar la frase: *es más fácil pedir perdón que permiso*, atribuída a la programadora Grace Hopper. Es decir, en este caso es más sencillo hacer la suma directamente; y Python se encargará de lanzar la excepción apropiada en caso de que algo salga mal:

```
def hacer_algo_con(x, y):
    z = x + y
    ...
```

12.4 Resumen

- Los errores que se pueden presentar en un programa son: de sintaxis (detectados por el intérprete), de semántica (el programa no funciona correctamente), o de ejecución (*excepciones*).
- Cuando el código a ejecutar pueda producir una excepción, es posible encerrarlo en los bloques correspondientes para *atrapar* la excepción y actuar en consecuencia.
- Si una función no atrapa una excepción, ésta se propaga a la función invocante; si ésta no la atrapa, la excepción se pasa a la invocante, hasta que se llega a una porción de código que atrapa la excepción, o bien se interrumpe la ejecución del programa.
- Cuando se atrapa una excepción es importante actuar en consecuencia, ya sea mostrando un mensaje de error, guardándolo en un archivo, o modificando el resultado final de la función.

- Antes de actuar sobre un dato en una porción de código, es deseable corroborar que se lo pueda utilizar. Para ello se puede validar su contenido, su tipo o sus atributos.
- Cuando no es posible utilizar un dato dentro de una porción de código, es importante informar el problema al código invocante, ya sea mediante una excepción o mediante un valor de retorno especial.

Referencia Python

`try: ... except:`

```
try:
    # código
except [tipo_de_excepción [as variable]]:
    # manejo de excepción
```

Si el código dentro del bloque `try` levanta una excepción, se ejecuta el código dentro del bloque `except` correspondiente.

El bloque puede tener tantos `except` como sea necesario; el último puede no tener un tipo de excepción asociado.

`try: ... finally:`

```
try:
    # código
finally:
    # código de limpieza
```

El código que se encuentra en el bloque `finally` se ejecuta al finalizar el código que se encuentra en el bloque `try`, sin importar si se levantó o no una excepción.

`try: ... except: ... finally:`

```
try:
    # código
except [tipo_de_excepción [as variable]]:
    # manejo de excepción
finally:
    # código de limpieza
```

Es una combinación de los otros dos casos. Si el código del bloque `try` levanta una excepción, se ejecutará el manejador correspondiente y, sin importar lo que haya sucedido, se ejecutará el bloque `finally` al concluir los otros bloques.

`raise [excepción([mensaje])]`

Levanta una excepción, para interrumpir el código de la función invocante.

Puede usarse sin parámetros, para levantar la última excepción atrapada.

Apéndice 12.A Ejemplo: programa con manejo de excepciones

A continuación se muestra un programa que calcula el módulo de un vector de n dimensiones, utilizando excepciones para hacer que la implementación sea más robusta.

```
def modulo(vector):
    """Calcula el módulo de un vector.
       Pre: el vector es una secuencia de números.
       Post: Devuelve el módulo del vector."""
    if not vector:
        raise ValueError("El vector debe tener al menos dimensión 1")
    suma = 0
    for x in vector:
        if not isinstance(x, (int, float, complex)):
            raise TypeError("El vector debe contener valores numéricos")
        suma += x * x
    return suma ** 0.5

def pedir_vector():
    vector = []
    while True:
        x = input("Ingrese un valor del vector, * para terminar: ")
        if x == "*":
            break
        try:
            vector.append(float(x))
        except ValueError:
            print("El valor no es numérico.")
    return tuple(vector)

def main():
    vector = pedir_vector()
    try:
        m = modulo(vector)
        print("El modulo de {} es {}".format(vector, m))
    except (TypeError, ValueError) as e:
        print("Error:", e)
```

Unidad 13

Procesamiento de archivos

En la unidad 11 se explicó como abrir, leer y escribir datos en los archivos. En general se quiere poder procesar la información que contienen estos archivos, para hacer algo útil con ella.

Dentro de las operaciones a realizar más sencillas se encuentran los denominados *filtros*, programas que procesan la entrada línea por línea, pudiendo seleccionar qué líneas formarán parte de la salida y pudiendo aplicar una operación determinada a cada una de estas líneas antes de pasarla a la salida.

En esta unidad se muestran algunas formas más complejas de procesar la información leída, que resultan útiles para procesar grandes cantidades de datos, evitando cargar la totalidad de los mismos en memoria.

13.1 Corte de control

La idea básica de este algoritmo es poder analizar información, generalmente provista mediante *registros*, agrupándolos según diversos criterios. Como precondición se incluye que la información debe estar ordenada según los mismos criterios por los que se la quiera agrupar. De modo que si varios registros tienen el mismo valor en uno de sus *campos*, se encuentren juntos, formando un grupo.

Se lo utiliza principalmente para realizar reportes que requieren subtotales, cantidades o promedios parciales u otros valores similares.

El algoritmo consiste en ir recorriendo la información, de modo que cada vez que se produzca un cambio en alguno de los campos correspondiente a uno de los criterios, se ejecutan los pasos correspondientes a la finalización de un criterio y el comienzo del siguiente.

Ejemplo

Supongamos que en un archivo **csv** tenemos los datos de las ventas de una empresa a sus clientes y se necesita obtener las ventas por cliente, mes por mes, con un total por año, otro por cliente y uno de las ventas totales. El formato está especificado de la siguiente forma:

```
cliente,año,mes,día,venta
```

Para poder hacer el reporte como se solicita, el archivo debe estar ordenado en primer lugar por `cliente`, luego por `año`, y luego por `mes`.

Teniendo el archivo ordenado de esta manera, es posible recorrerlo e ir realizando los subtotales correspondientes, a medida que se los va obteniendo. Se muestra la implementación en el Código 13.1.

Código 13.1 `ventas.py`**: Ejemplo de corte de control.** Recorre un archivo de ventas e imprime totales y subtotales

```
import csv

def ventas_clientes_mes(archivo_ventas):
    """Calcula totales por mes, año y cliente a partir de la
    información almacenada en el archivo CSV, que debe tener
    el formato: cliente,año,mes,día,venta"""

    with open(archivo_ventas) as f:
        ventas_csv = csv.reader(f)

        item = next(ventas_csv, None)
        total = 0

        while item:
            # Inicialización para el bucle de cliente
            cliente = item[0]
            total_cliente = 0
            print("Cliente: {}".format(cliente))

            while item and item[0] == cliente:
                # Inicialización para el bucle de año
                año = item[1]
                total_año = 0
                print("\tAño: {}".format(año))

                while item and item[0] == cliente and item[1] == año:
                    mes, monto = item[2], float(item[4])
                    print("\t\tVentas del mes {}: {:.2f}".format(mes, monto))
                    total_año += monto

                    # Siguiente registro
                    item = next(ventas_csv, None)

                # Final del bucle de año
                print("\tTotal para el año {}: {:.2f}".format(año, total_año))
                total_cliente += total_año

            # Final del bucle de cliente
            print("Total para el cliente {}: {:.2f}\n".format(cliente,
    ↪ total_cliente))
            total += total_cliente

        # Final del bucle principal
        print("Total general: {:.2f}".format(total))
```

Se puede ver que para resolver el problema es necesario contar con tres bucles anidados, que van incrementando la cantidad de condiciones a verificar.

> Las soluciones de corte de control son siempre de esta forma: una serie de bucles anidados, que incluyen las condiciones del bucle padre y agregan su propia condición, y el movimiento hacia el siguiente registro se realiza en el bucle con mayor nivel de anidación.

Nota. En `ventas.py` utilizamos una función que no habíamos mencionado hasta ahora:

```
item = next(ventas_csv, None)
```

La función `next` lee el siguiente registro del archivo CSV y lo devuelve. Cuando no quedan más registros para leer, devuelve el valor que le pasamos en el segundo parámetro (`None`).

13.2 Apareo

Así como el corte de control nos sirve para generar un reporte, el apareo nos sirve para asociar/relacionar datos que se encuentran en distintos archivos.

La idea básica es: a partir de dos archivos (uno principal y otro relacionado) que tienen alguna información que los enlace, generar un tercero (o una salida por pantalla), como una mezcla de los dos.

Para hacer esto es conveniente que ambos archivos estén ordenados por el valor que los relaciona.

Ejemplo

Por ejemplo, si se tiene un archivo con un listado de alumnos (padrón, apellido, nombre, carrera), y otro archivo que contiene las notas de esos alumnos (padrón, materia, nota), y se quieren listar todas las notas que corresponden a cada uno de los alumnos, se lo puede hacer como se muestra en el Código 13.2.

En este ejemplo usamos apareo de datos para combinar y mostrar información. De forma similar se puede utilizar para agregar información nueva, borrar información o modificar datos de la tabla principal. Gran parte de las bases de datos relacionales basan su funcionamiento en estas funcionalidades.

13.3 Resumen

- Existen diversas formas de procesar archivos con grandes cantidades de información. Se puede simplemente filtrar la entrada para obtener una salida, o se pueden realizar operaciones más complejas como el **corte de control** o el **apareo**
- El corte de control es una técnica de procesamiento de datos ordenados por diversos criterios, que permite agruparlos para obtener subtotales.
- El apareo es una técnica de procesamiento que involucra dos o más archivos con datos ordenados, y permite generar una salida combinada a partir de los mismos.

Código 13.2 `notas.py`**: Ejemplo de apareo.** Recorre un archivo de alumnos y otro de notas e imprime las notas que corresponden a cada alumno

```
import csv

def imprimir_notas_alumnos(alumnos, notas):
    """Procesa los archivos de alumnos y notas, y por cada
    alumno imprime todas las notas que le corresponden."""
    with open(notas) as f_notas, open(alumnos) as f_alumnos:
        notas_csv = csv.reader(f_notas)
        alumnos_csv = csv.reader(f_alumnos)

        # Saltea los encabezados
        next(notas_csv)
        next(alumnos_csv)

        # Empieza a leer
        alumno = next(alumnos_csv, None)
        nota = next(notas_csv, None)
        while alumno:
            padron, apellido, nombre, carrera = alumno
            print("{}, {} ({})".format(apellido, nombre, padron))
            if not nota or nota[0] != padron:
                print("\tNo se registran notas")
            while nota and nota[0] == padron:
                print("\t{}: {}".format(nota[1], nota[2]))
                nota = next(notas_csv, None)
            alumno = next(alumnos_csv, None)
```

Unidad 14

Objetos

Los *objetos* son una manera de organizar datos y de relacionar esos datos con el código apropiado para manejarlo. Son los protagonistas de un paradigma de programación llamado *Programación Orientada a Objetos*.

Nosotros ya usamos objetos en Python sin mencionarlo explícitamente. Es más, todos los tipos de datos que Python nos provee son, en realidad, objetos.

De forma que, cuando utilizamos `cadena.upper()`, le estamos diciendo a Python que llame a la función `upper` del tipo `str` para `cadena` que es lo mismo que decir que llame al *método* `upper` del objeto `cadena`.

A su vez, a las variables que un objeto contiene se las llama *atributos*.

Sabías que…

La Programación Orientada a Objetos introduce bastante terminología, y una gran parte es simplemente darle un nuevo nombre a cosas que ya estuvimos usando. Esto si bien parece raro es algo bastante común en el aprendizaje humano.

Para poder pensar abstractamente, los humanos necesitamos asignarle distintos nombres a cada cosa o proceso. De la misma manera, para poder hacer un cambio en una forma de ver algo ya establecido (realizar un *cambio de paradigma*), suele ser necesario cambiar la forma de nombrar a los elementos que se comparten con el paradigma anterior, ya que sino es muy difícil realizar el salto al nuevo paradigma.

14.1 Tipos

En los temas que vimos hasta ahora nos hemos encontrado con numerosos tipos provistos por Python, los *números*, las *cadenas de caracteres*, las *listas*, las *tuplas*, los *diccionarios*, los *archivos*, etc. Cada uno de estos tipos tiene sus características, tiene operaciones propias de cada uno y nos provee de una gran cantidad de funcionalidades que podemos utilizar para nuestros programas.

Como ya se dijo en unidades anteriores, para saber de qué tipo es un valor, utilizamos la función `type`. Para saber qué métodos y atributos tiene ese valor utilizamos la función `dir`:

```
>>> f = open("archivo.txt")
>>> type(f)
<class '_io.TextIOWrapper'>
```

```
>>> dir(f)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__',
'__init__', '__iter__', '__le__', '__lt__', '__ne__', '__new__',
'__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_checkClosed',
'_checkReadable', '_checkSeekable', '_checkWritable', '_finalizing',
'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno',
'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines',
'read', 'readable', 'readline', 'readlines', 'seek', 'seekable',
'tell', 'truncate', 'writable', 'write', 'writelines']
```

En este caso, la función `dir` nos muestra los métodos que tiene un objeto del tipo `_io.TextIOWrapper` (el tipo que Python le asigna internamente a un archivo). Podemos ver en el listado los métodos que ya hemos visto al operar con archivos, junto con otros métodos con nombres *raros* como `__str__`, o `__doc__`. Estos métodos son especiales en Python; más adelante veremos para qué sirven y cómo se usan.

En el listado que nos da `dir` están los atributos y métodos mezclados. Si necesitamos saber cuáles son atributos y cuáles son métodos, podemos hacerlo nuevamente mediante el uso de `type`.

```
>>> type(f.name)
<type 'str'>
>>> f.name
'archivo.txt'
>>> type (f.tell)
<type 'builtin_function_or_method'>
>>> f.tell()
0L
```

Es decir que `name` es un atributo del objeto (el nombre del archivo), mientras que `tell` es un método, que para utilizarlo debemos llamarlo con paréntesis.

En Python los métodos se invocan con la *notación punto*: `archivo.tell()`, `cadena.split(":")`.

Analicemos la segunda expresión. El significado de ésta es: la variable `cadena` llama al método `split` (del cual es dueña por tratarse de una variable de tipo `str`) con el argumento `":"`.

Sería equivalente a llamar a la función `split` pasándole como primer parámetro la variable `cadena`, y como segundo parámetro el delimitador `":"`. Pero la diferencia de notación resalta que el método `split` es un método **de** cadenas, y que no se lo puede utilizar con variables de otros tipos.

Esta notación provocó un cambio de paradigma en la programación, y es uno de los ejes de la *Programación Orientada a Objetos*.

14.2 Qué es un objeto

En Python, todos los tipos son objetos. Pero no en todos los lenguajes de programación es así. En general, podemos decir que un objeto es una forma ordenada de agrupar datos (los *atributos*) y operaciones a utilizar sobre esos datos (los *métodos*).

Es importante notar que cuando decimos *objetos* podemos estar haciendo referencia a dos cosas parecidas, pero distintas.

Por un lado, la definición del tipo, donde se indican cuáles son los atributos y métodos que van a tener todas las variables que sean de ese tipo. Esta definición se llama específicamente, la **clase** del objeto.

A partir de una clase es posible crear distintos valores que son de ese tipo. A cada uno de los valores generados a partir de una clase se los llama **instancia** de esa clase.

> Se dice que los objetos tienen **estado** y **comportamiento**, ya que los valores que tengan los atributos de una instancia determinan el estado actual de esa instancia, y los métodos definidos en una clase determinan cómo se va a comportar ese objeto.

14.3 Definiendo nuevos tipos

Sin bien Python nos provee con un gran número de tipos ya definidos, en muchas situaciones utilizar solamente los tipos provistos por el lenguaje resultará insuficiente. En estas situaciones queremos poder crear nuestros propios tipos, que almacenen la información relevante para el problema a resolver y contengan las funciones para operar con esa información.

Por ejemplo, si se quiere representar un punto en el plano, es posible hacerlo mediante una tupla de dos elementos, pero esta implementación es limitada, ya que si se quiere poder operar con distintos puntos (sumarlos, restarlos o calcular la distancia entre ellos) se deberán tener funciones *sueltas* para realizar las diversas operaciones.

Podemos hacer algo mejor definiendo un nuevo tipo `Punto`, que almacene la información relacionada con el punto, y contenga las operaciones que nos interese realizar sobre él.

14.3.1 Nuestra primera clase: Punto

Queremos definir nuestra clase que represente un punto en el plano. Lo primero que debemos notar es que existen varias formas de representar un punto en el plano, por ejemplo, coordenadas polares o coordenadas cartesianas. Además, existen varias operaciones que se pueden realizar sobre un punto del plano, e implementarlas todas podría llevar mucho tiempo.

Si representamos nuestro punto utilizando coordenadas cartesianas, vamos a necesitar dos atributos numéricos, `x` e `y`, para almacenar las coordenadas. En la Figura 14.1, se muestra un diagrama simple con el diseño de la clase `Punto`.

Figura 14.1: Diseño de la *clase* `Punto`, y dos posibles *instancias*

Veamos cómo crear nuestra clase `Punto` en Python:

```
class Punto: ❶
    """Representación de un punto en el plano en
       coordenadas cartesianas (x, y)"""

    def __init__(self, x, y): ❷
        "Constructor de Punto. x e y deben ser numéricos"
        self.x = x ❸
        self.y = y
```

❶ En la primera línea de código indicamos que vamos a crear una nueva clase, llamada `Punto`. Por convención, en los nombres de las clases definidas por el programador se escribe

cada palabra del nombre con la primera letra en mayúsculas (`Punto`, `Rectangulo`, `ListaEnlazada`, `Hotel`, etc.).

❷ Además definimos uno de los métodos especiales, `__init__`, el **constructor** de la clase. Este método se llama automáticamente cada vez que se crea una nueva instancia de la clase.

Todos los métodos de cualquier clase reciben como primer parámetro a la instancia sobre la que está trabajando. Por convención, a ese primer parámetro se lo suele llamar `self` (que podríamos traducir como *yo mismo*), pero puede llamarse de cualquier forma.

En nuestro caso el constructor `__init__` recibe la instancia `self` y dos parámetros más, `x` e `y`. De esta forma indicamos que para crear una instancia de `Punto` será necesario proveer el valor de ambas coordenadas.

❸ En el cuerpo del constructor se definen los atributos, utilizando la notación punto con la instancia `self`. En este caso los atributos se llamarán igual que los parámetros del constructor, `x` e `y`, y se inicializan con el valor de los parámetros.

Para utilizar esta clase que acabamos de definir, lo haremos de la siguiente forma:

```
>>> p = Punto(5, 7)
>>> type(p)
<class '__main__.Punto'>
>>> p.x
5
>>> p.y
7
```

Al realizar la llamada `Punto(5, 7)`:

1. Se crea una nueva instancia de la clase `Punto`.
2. Se ejecuta el constructor `__init__`, con `self` → la instancia nueva, `x` → 5, `y` → 7.
3. El constructor asigna los atributos `self.x` → 5, `self.y` → 7.
4. Cuando finaliza la ejecución del constructor, se asigna `p` → la instancia de `Punto` recién creada.

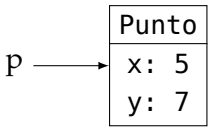


Figura 14.2: Una instancia de `Punto`, referenciada con la variable `p`

14.3.2 Agregando validaciones al constructor

Hemos creado una clase `Punto` que permite guardar valores `x` e `y`. Sin embargo, por más que en la documentación se indique que los valores deben ser numéricos, el código mostrado hasta ahora no impide que a `x` e `y` se les asigne un valor cualquiera, no numérico.

```
>>> q = Punto("A", True)
>>> q.x
'A'
>>> q.y
True
```

Si queremos impedir que esto suceda, debemos agregar validaciones al constructor, como las vistas en unidades anteriores.

Verificaremos que los valores pasados para `x` e `y` sean numéricos, utilizando la función `validar_numero`:

```
def validar_numero(valor):
    "Si el valor es numérico, lo devuelve. En caso contrario lanza TypeError."
    if not isinstance(valor, (int, float, complex)):
        raise TypeError("{:r} no es un valor numérico".format(valor))
    return valor
```

El nuevo constructor quedará así:

```
    def __init__(self, x, y):
        """Constructor de Punto. x e y deben ser numéricos,
           de no ser así, se levanta una excepción TypeError"""
        self.x = validar_numero(x)
        self.y = validar_numero(y)
```

Este constructor impide que se creen instancias con valores inválidos para `x` e `y`.

```
>>> Punto("A", True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "punto.py", line 12, in __init__
    self.x = validar_numero(x)
  File "punto.py", line 5, in validar_numero
    raise TypeError("{!r} no es un valor numérico".format(valor))
TypeError: 'A' no es un valor numérico
```

14.3.3 Agregando operaciones

Hasta ahora hemos creado una clase `Punto` que permite construirla con un par de valores, que deben ser sí o sí numéricos, pero no podemos operar con esos valores. Para aprovechar la ventaja de los objetos, tenemos que definir operaciones adicionales que vayamos a querer realizar sobre esos puntos.

Queremos, por ejemplo, poder calcular la distancia entre dos puntos. Para ello definimos un nuevo método `distancia` que recibe el punto de la instancia actual y el punto para el cual se quiere calcular la distancia.

```
    def distancia(self, otro):
        """Devuelve la distancia entre ambos puntos."""
        dx = self.x - otro.x
        dy = self.y - otro.y
        return (dx * dx + dy * dy) ** 0.5
```

Una vez agregado este método a la clase, será posible obtener la distancia entre dos puntos, de la siguiente manera:

```
>>> p = Punto(5, 7)
>>> q = Punto(2, 3)
>>> p.distancia(q)
5.0
```

Podemos ver, sin embargo, que la operación para calcular la distancia incluye la operación de restar dos puntos y la de obtener la norma de un vector. Sería deseable incluir también estas dos operaciones dentro de la clase `Punto`.

Agregaremos, entonces, el método para restar dos puntos:

```
    def restar(self, otro):
        """Devuelve el Punto que resulta de la resta
           entre dos puntos."""
        return Punto(self.x - otro.x, self.y - otro.y)
```

La resta entre dos puntos es un nuevo punto. Es por ello que este método devuelve una nueva instancia de `Punto`, en lugar de modificar las instancias `self` u `otro`.

A continuación definimos el método para calcular la norma del vector que se forma uniendo un punto con el origen.

```
    def norma(self):
        """Devuelve la norma del vector que va desde el origen
           hasta el punto. """
        return (self.x * self.x + self.y * self.y) ** 0.5
```

En base a estos dos métodos podemos ahora volver a escribir el método `distancia` para que aproveche el código de ambos:

```
    def distancia(self, otro):
        """Devuelve la distancia entre ambos puntos."""
        return self.restar(otro).norma()
```

En definitiva, hemos definido tres operaciones en la clase `Punto`, que nos sirve para calcular restas, normas de vectores al origen, y distancias entre puntos.

```
>>> p = Punto(5, 7)
>>> q = Punto(2, 3)
>>> r = p.restar(q)
>>> (r.x, r.y)
(3, 4)
>>> r.norma()
5.0
>>> q.distancia(r)
1.41421356237
```

Atención

Cuando definimos los métodos que va a tener una determinada clase es importante tener en cuenta que el listado de métodos debe ser lo más conciso posible.

Es decir, si una clase tiene algunos métodos básicos que pueden combinarse para obtener distintos resultados, no queremos implementar toda posible combinación de llamadas a los métodos básicos, sino sólo los básicos y aquellas combinaciones que sean muy frecuentes, o en las que tenerlas como un método aparte implique una ventaja significativa en cuanto al tiempo de ejecución de la operación.

Este concepto se llama **ortogonalidad** de los métodos, basado en la idea de que cada método debe realizar una operación independiente de los otros. Entre las motivaciones que puede haber para agregar métodos que no sean ortogonales, se encuentran la *simplicidad de uso* y la *eficiencia*.

14.4 Métodos especiales

Así como el constructor, `__init__`, existen diversos métodos especiales que, si están definidos en nuestra clase, Python los llamará por nosotros cuando se utilice una instancia en situaciones particulares.

14.4.1 Conversión a cadena de texto

Veamos qué pasa cuando intentamos convertir la instancia de `Punto` a una cadena de texto:

```
>>> p = Punto(5, 7)
>>> str(p)
'<__main__.Punto object at 0x7fa6da7694a8>'
```

Esto no resulta muy satisfactorio. Lo mismo sucede cuando intentamos imprimir el objeto con `print` (internamente la función `print` aplica la función `str` a todos los objetos que recibe antes de imprimirlos):

```
>>> print(p)
<__main__.Punto object at 0x7fa6da7694a8>
```

Para poder modificar el comportamiento del objeto al convertir a cadena, Python indica que hay que agregarle a la clase un método especial, llamado `__str__` que debe recibir un solo parámetro (`self`), y debe devolver la cadena de texto deseada. Ese método se invocará cuando se llame a la función `str`. El método `__str__` tiene un solo parámetro, `self`.

En nuestro caso decidimos mostrar el punto como un par ordenado, por lo que escribimos el siguiente método dentro de la clase `Punto`:

```
    def __str__(self):
        """Devuelve la representación del Punto como
           cadena de texto."""
        return "({}, {})".format(self.x, self.y)
```

Una vez definido este método, nuestro punto se mostrará como un par ordenado cuando se necesite una representación de cadenas.

```
>>> p = Punto(-6, 18)
>>> str(p)
'(-6, 18)'
>>> print(p)
(-6, 18)
```

Sabías que…

Muchas de las funciones provistas por Python, que ya hemos utilizado en unidades anteriores, como `str`, `len` o `help`, invocan internamente a los métodos especiales de los objetos.

Es decir que la función `str` internamente invoca al método `__str__` del objeto que recibe como parámetro. De la misma manera `len` invoca internamente al método `__len__`, si es que está definido.

Cuando mediante `dir` vemos que un objeto tiene alguno de estos métodos especiales, utilizamos la función de Python correspondiente a ese método especial.

La conversión a cadena con `__str__` funciona con `str` y con `print`, pero aun hay algunos casos en los que Python imprime la representación por omisión:

```
>>> p = Punto(-6, 18)
>>> str(p)
'(-6, 18)'
>>> p
<__main__.Punto object at 0x7fa6da7694a8>
>>> str([p])
'[<__main__.Punto object at 0x7fa6da7694a8>]'
```

```
>>> [p]
[<__main__.Punto object at 0x7fa6da7694a8>]
```

En estos casos, en lugar de llamar a `str` Python llama a la función `repr`. Mientras que La función `str` se usa para obtener una representación "informal" o "legible" del objeto, pensada tal vez para mostrar a un usuario, el objetivo de `repr` es obtener una representación "formal" y "desambiguada", pensada para mostrar a un programador:

```
>>> str("hola")
'hola'
>>> repr("hola")
"'hola'"
```

Internamente, la función `repr` invoca al método `__repr__` del objeto, y podemos implementarlo para modificar el comportamiento por omisión:

```
    def __repr__(self):
        """Devuelve la representación formal del Punto como
           cadena de texto."""
        return "Punto({}, {})".format(self.x, self.y)
```

```
>>> p = Punto(-6, 18)
>>> str(p)
'(-6, 18)'
>>> repr(p)
'Punto(-6, 18)'
>>> p
Punto(-6, 18)
```

14.4.2 Métodos para operar matemáticamente

Ya hemos visto un método que permitía restar dos puntos. Si bien esta implementación es perfectamente válida, no es posible usar esa función para realizar una resta con el operador `-`.

```
>>> p = Punto(3,4)
>>> q = Punto(2,5)
>>> p - q
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'Punto' and 'Punto'
```

Si queremos que este operador (o el equivalente para la suma) funcione, será necesario implementar algunos métodos especiales.

```
    def __add__(self, otro):
        """Devuelve la suma de ambos puntos."""
        return Punto(self.x + otro.x, self.y + otro.y)

    def __sub__(self, otro):
        """Devuelve la resta de ambos puntos."""
        return Punto(self.x - otro.x, self.y - otro.y)
```

El método `__add__` es el que se utiliza para el operador +; el primer parámetro es el primer operando de la suma, y el segundo parámetro el segundo operando. Debe devolver una nueva instancia, nunca modificar la clase actual. De la misma forma, el método `__sub__` es el utilizado por el operador `-`.

Ahora es posible operar con los puntos directamente mediante los operadores, en lugar de llamar a métodos:

```
>>> p = Punto(3, 4)
>>> q = Punto(2, 5)
>>> p - q
Punto(1, -1)
>>> p + q
Punto(5, 9)
```

De la misma forma, si se quiere poder utilizar cualquier otro operador matemático, será necesario definir el método apropiado.

Sabías que…

La posibilidad de definir cuál será el comportamiento de los operadores básicos (como `+`, `-`, `*`, `/`), se llama **sobrecarga de operadores**.

No todos los lenguajes lo permiten, y si bien es cómodo y permite que el código sea más elegante, no es algo esencial a la Programación Orientada a Objetos.

Entre los lenguajes más conocidos que no soportan sobrecarga de operadores están C, Java, Pascal, Objective C. Entre los lenguajes más conocidos que sí soportan sobrecarga de operadores están Python, C++, C#, Ruby, PHP.

14.5 Composición de objetos

Supongamos que queremos diseñar un objeto para representar un rectángulo en el plano. ¿Qué atributos tendría?

Podemos hacer que nuestro `Rectangulo` tenga cuatro números: `x1, y1, x2, y2`, donde `x1` e `y1` son las coordenadas de la esquina superior izquierda y `x2` e `y2` las coordenadas de la esquina inferior derecha.

Otra opción es aprovechar que ya tenemos una clase para representar un punto en el plano, y *componer* nuestro `Rectangulo` con la clase `Punto`. En lugar de tener cuatro números, podemos tener dos `Puntos` que podemos llamar `noroeste` y `sudeste`.

Y estas no son las únicas dos opciones posibles. Podemos por ejemplo representar nuestro `Rectangulo` con un `Punto` para la esquina superior izquierda, y luego dos números `ancho` y `alto`.

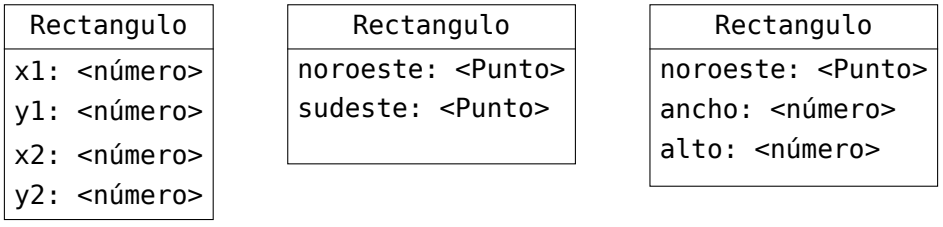


Figura 14.3: Tres posibles diseños para la clase `Rectangulo`

Si elegimos por ejemplo la representación utilizando dos Puntos, podemos implementarlo de la siguiente manera:

```
class Rectangulo:
```

```
    "Representa un rectángulo en el plano"

    def __init__(self, noroeste, sudeste):
        """Crea un Rectangulo a partir de los Puntos correspondientes a las
        esquinas superior izquierda e inferior derecha"""
        self.noroeste = noroeste
        self.sudeste = sudeste
```

Y a partir de ahora podemos crear rectángulos de la siguiente manera:

```
>>> p = Punto(3, 4)
>>> q = Punto(6, 5)
>>> r = Rectangulo(p, q)
```

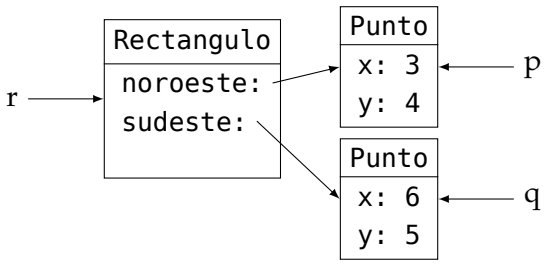


Figura 14.4: Una instancia de `Rectangulo` compuesta de dos instancias de `Punto`

14.6 Mutabilidad

En Python todos los objetos creados por el programador son *mutables*; es decir que luego de ser creada una instancia podemos modificar el valor de sus atributos (modificando así el *estado* del objeto):

```
>>> p = Punto(5, 7)
>>> p
Punto(5, 7)
>>> p.x = 3
>>> p
Punto(3, 7)
```

Si bien Python permite modificar atributos como en el ejemplo anterior, en muchos casos es más conveniente *encapsular* el comportamiento adentro de un método. Por ejemplo, si queremos permitir que un `Punto` pueda ser desplazado en el plano, podemos agregar el método `desplazar`:

```
    def desplazar(self, dx, dy):
        """Desplaza el punto según dx y dy."""
        self.x += dx
        self.y += dy
```

De esta manera, podemos llamar a `desplazar` para mutar los atributos del punto:

```
>>> p = Punto(5, 7)
>>> p.desplazar(-2, 0)
>>> p
Punto(3, 7)
```

Como ya vimos en la Sección 10.3, los valores mutables introducen complejidad adicional en un programa, ya que el comportamiento de cualquier línea de código pasa a depender además del *estado* de cada objeto mutable.

Tomemos por ejemplo el caso de nuestra clase `Rectangulo`:

```
>>> p = Punto(3, 4)
>>> q = Punto(6, 5)
>>> r = Rectangulo(p, q)
>>> r.noroeste
Punto(3, 4)
>>> p.desplazar(1, 1)
>>> r.noroeste
Punto(4, 5)
```

Al llamar a `p.desplazar(1, 1)` estamos modificando el estado del punto `p`, pero al mismo tiempo, inadvertidamente estamos modificando el rectángulo `r`, ya que su atributo `noroeste` es una referencia a *la misma instancia* de `Punto` que acabamos de modificar (ver Figura 14.4).

La conclusión es que al diseñar un objeto tenemos que evaluar si queremos que sea mutable o inmutable. Como regla general, los valores inmutables suelen favorecer la mantenibilidad del código, pero puede haber casos en los que la mutabilidad se hace necesaria para mejorar la eficiencia.

En otros lenguajes orientados a objetos hay formas de declarar la mutabilidad o inmutabilidad de una clase; En Python, si queremos que nuestro objeto sea inmutable tenemos que hacerlo *por convención*: decir, simplemente no agregamos ningún método que modifique el estado.

14.7 Creando clases más complejas

Nos contratan para diseñar una clase para evaluar la relación calidad-precio de diversos hoteles. Nos dicen que los atributos que se cargarán de los hoteles son: nombre, ubicación, puntaje obtenido por votación, y precio, y que además de guardar hoteles y mostrarlos, debemos poder compararlos en términos de sus valores de relación calidad-precio, de modo tal que $x < y$ signifique que el hotel x es peor en cuanto a la relación calidad-precio que el hotel y, y que dos hoteles son iguales si tienen la misma relación calidad-precio. La relación calidad-precio de un hotel la definen nuestros clientes como $10 \cdot \mathit{puntaje}^2/\mathit{precio}$.

Además, y como resultado de todo esto, tendremos que ser capaces de ordenar de menor a mayor una lista de hoteles, usando el orden que nos acaban de definir.

Averiguamos un poco más respecto de los atributos de los hoteles:

- El nombre y la ubicación deben ser cadenas no vacías.
- El puntaje debe ser un número (sin restricciones sobre su valor)
- El precio debe ser un número distinto de cero.

Empezamos diseñar a la clase:

- El método `__init__`:
  - Creará objetos de la clase `Hotel` con los atributos que se indicaron (`nombre`, `ubicacion`, `puntaje`, `precio`).

  - Necesitamos validar que `puntaje` y `precio` sean números positivos. Para esto utilizaremos una función `validar_numero_positivo` similar a la función `validar_numero` que usamos para la clase `Punto`.
  - Necesitamos validar que nombre y ubicación sean cadenas no vacías (para lo cual tenemos que construir una función `validar_cadena_no_vacia`).
  - Cuando los datos no satisfagan los requisitos se levantará una excepción `TypeError`.

- Contará con un método `__str__` para mostrar a los hoteles mediante una cadena del estilo: `"Hotel City de Mercedes - Puntaje: 3.25 - Precio: 78 pesos"`.

- Respecto a la relación de orden entre hoteles, la clase deberá poder contar con los métodos necesarios para realizar esas comparaciones y para ordenar una lista de hoteles.

Podemos realizar casi todas las tareas con los temas vistos para la creación de la clase `Punto`. Para el último ítem deberemos introducir nuevos métodos especiales.

Ejercicio 14.1. Escribir las funciones `validar_numero_positivo` y `validar_cadena_no_vacia`, que deben lanzar `TypeError` si la validación del valor falla, y en caso contrario simplemente devolver el valor. Incluir todas las funciones de validación en un módulo `validaciones.py`.

El fragmento inicial de la clase programada en Python queda así:

```
class Hotel:
    """Representa un hotel: sus atributos son:
       nombre, ubicacion, puntaje y precio."""

    def __init__(self, nombre, ubicacion, puntaje, precio):
        """Crea un Hotel.
           nombre y ubicacion deben ser cadenas no vacías,
           puntaje y precio son números."""
        self.nombre = validar_cadena_no_vacia(nombre)
        self.ubicacion = validar_cadena_no_vacia(ubicacion)
        self.puntaje = validar_numero_positivo(puntaje)
        self.precio = validar_numero_positivo(precio)

    def __str__(self):
        """Conversión a cadena de texto."""
        return "{} de {} - Puntaje: {} - Precio: {} pesos".format(
            self.nombre,
            self.ubicacion,
            self.puntaje,
            self.precio,
        )

    def ratio(self):
        """Calcula la relación calidad-precio de un hotel"""
        return ((self.puntaje ** 2) * 10) / self.precio
```

Con este código tenemos ya la posibilidad de construir un hotel, asegurando que los atributos de los tipos correspondientes, mostrarlo según nos lo han solicitado y calcular su relación calidad-precio:

```
>>> h = Hotel("Hotel City", "Mercedes", 3.2, 20)
>>> print(h)
Hotel City de Mercedes - Puntaje: 3.2 - Precio: 20 pesos.
```

```
>>> h.ratio()
5.12
```

14.7.1 Métodos para comparar objetos

Para resolver las comparaciones entre hoteles, será necesario definir algunos métodos especiales que permiten comparar objetos.

En particular, cuando se quiere que los objetos puedan ser ordenados, es suficiente con definir el método `__lt__` (*less than*), que corresponde al operador matemático de comparación <. El método `__lt__` recibe dos parámetros, `self` y `otro` y debe devolver `True` si `self` es comparativamente "menor" a `otro`.

En el caso de nuestra clase `Hotel`, podemos decir que un hotel es "menor" a otro si su relación calidad-precio es menor:

```
    def __lt__(self, otro):
        """Compara dos hoteles según sus ratios."""
        return self.ratio() < otro.ratio()
```

Una vez que está definida esta función podremos realizar comparaciones entre los hoteles:

```
>>> h = Hotel("Hotel City", "Mercedes", 3.25, 78)
>>> i = Hotel("Hotel Mascardi", "Bariloche", 6, 150)
>>> i < h
False
>>> i > h
True
```

Al implementar el método `__lt__`, las instancias de la clase se pueden comparar con los operadores < y > (El operador > se fija primero si existe `__gt__`, y si no existe invoca a `__lt__`).

Pero aún quedan sin definir las operaciones ==, !=, <= y >=. Si queremos tener la posibilidad de preguntar si dos instancias son "iguales" o "distintas" con == y !=, tenemos que definir el método `__eq__`. Y si queremos tener la posibilidad de comparar con <= o >= tenemos que implementar el método `__le__`.

14.7.2 Ordenar de menor a mayor listas de hoteles

En la sección 7.3 vimos que se puede ordenar una lista usando el método `sort`:

```
>>> l1 = [10, -5, 8, 12, 0]
>>> l1.sort()
>>> l1
[-5, 0, 8, 10, 12]
```

De la misma forma, una vez que hemos definido el método `__lt__`, podemos ordenar listas de hoteles, ya que internamente el método `sort` comparará los hoteles mediante el método de comparación que hemos definido:

```
>>> h1 = Hotel("Hotel 1* normal", "MDQ", 1, 10)
>>> h2 = Hotel("Hotel 2* normal", "MDQ", 2, 40)
>>> h3 = Hotel("Hotel 3* carisimo", "MDQ", 3, 130)
>>> h4 = Hotel("Hotel vale la pena" ,"MDQ", 4, 130)
>>> lista = [h1, h2, h3, h4]
>>> lista.sort()
>>> for hotel in lista:
...     print(hotel)
```

```
...
Hotel 3* carisimo de MDQ - Puntaje: 3 - Precio: 130 pesos.
Hotel 1* normal de MDQ - Puntaje: 1 - Precio: 10 pesos.
Hotel 2* normal de MDQ - Puntaje: 2 - Precio: 40 pesos.
Hotel vale la pena de MDQ - Puntaje: 4 - Precio: 130 pesos.
```

Podemos verificar cuál fue el criterio de ordenamiento invocando al método `ratio` en cada caso:

```
>>> h1.ratio()
1.0
>>> h2.ratio()
1.0
>>> h3.ratio()
0.69230769230769229
>>> h4.ratio()
1.2307692307692308
```

Y vemos que efectivamente:

- "Hotel 3* carisimo", con la menor relación calidad-precio aparece primero.
- "Hotel 1* normal" y "Hotel 2* normal" con la misma relación calidad-precio (igual a 1.0 en ambos casos) aparecen en segundo y tercer lugar en la lista.
- "Hotel vale la pena" con la mayor relación calidad-precio aparece en cuarto lugar en la lista.

Hemos por lo tanto ordenado la lista de acuerdo al criterio solicitado.

14.7.3 Otras formas de comparación

Si además de querer listar los hoteles por su relación calidad-precio también se quiere poder listarlos según su puntaje, o según su precio, no se lo puede hacer mediante el método `__lt__` (a menos que redefinamos el método, pero eso sería poco práctico).

Para situaciones como esta, `sort` puede recibir, opcionalmente, otro parámetro `key` que es la función que calcula la *clave de comparación* de cada elemento.

La clave de comparación es simplemente el valor numérico que queremos asignar a cada elemento para comparar y ordenar. Cuando ordenamos los hoteles en el ejemplo anterior, podríamos decir que la clave de comparación era el ratio, y por lo tanto podríamos haber ordenado de la siguiente manera, sin necesidad de implementar `__lt__`:

```
lista.sort(key=Hotel.ratio)
```

Así, para ordenar según el precio:

```
>>> def precio(hotel):
...     return hotel.precio
>>> lista.sort(key=precio)
>>> for hotel in lista:
...     print(hotel)
...
Hotel 1* normal de MDQ - Puntaje: 1 - Precio: 10 pesos.
Hotel 2* normal de MDQ - Puntaje: 2 - Precio: 40 pesos.
Hotel 3* carisimo de MDQ - Puntaje: 3 - Precio: 130 pesos.
Hotel vale la pena de MDQ - Puntaje: 4 - Precio: 130 pesos.
```

14.7.4 Comparación sólo por igualdad o desigualdad

Existen clases, como la clase `Punto` vista anteriormente, que no se pueden ordenar, ya que no se puede decir si dos puntos son menores o mayores, con lo cual no se puede implementar un método `__lt__`.

Pero en estas clases, en general, será posible comparar si dos objetos son o no iguales, es decir si tienen o no el mismo valor, aún si se trata de objetos distintos.

```
>>> p = Punto(3, 4)
>>> q = Punto(3, 4)
>>> p == q
False
```

En este caso, por más que los puntos tengan el mismo valor, al no estar definido ningún método de comparación Python no sabe cómo comparar los valores, y lo que compara son las instancias. `p` y `q` son instancias distintas, por más que tengan los mismos valores.

Para obtener el comportamiento esperado en estos casos, se redefine el método `__eq__`. Por ejemplo, el método `__eq__` para la clase `Punto` sería:

```
    def __eq__(self, otro):
        """Devuelve si dos puntos son iguales."""
        return self.x == otro.x and self.y == otro.y
```

Una vez agregados estos métodos ya se puede comparar los puntos por su igualdad o desigualdad:

```
>>> p = Punto(3, 4)
>>> q = Punto(3, 4)
>>> p == q
True
>>> p != q
False
>>> r = Punto(2, 3)
>>> p == r
False
>>> p != r
True
```

14.8 Resumen

- Los **objetos** son formas ordenadas de agrupar datos (*atributos*, *estado*) y operaciones sobre estos datos (*métodos*, *comportamiento*).
- Cada objeto es de una **clase** o tipo, que define cuáles serán sus atributos y métodos. Y cuando se crea un valor a partir de una clase en particular, decimos que se crea una **instancia** de esa clase.
- Para nombrar una clase definida por el programador, se suele usar una letra mayúscula al comienzo de cada palabra.
- El **constructor** de una clase es el método que se ejecuta cuando se crea una nueva instancia de esa clase.
- Es posible definir una gran variedad de métodos dentro de una clase, incluyendo métodos especiales que pueden utilizados para mostrar, sumar, comparar u ordenar los objetos.

- En Python, los objetos creados por el programador son **mutables**. Podemos diseñar objetos inmutables por convención: simplemente no agregamos ningún método que modifique el estado del objeto.

Referencia Python

`class NombreClase:`

Indica que se comienza a definir una clase con el nombre indicado.

`def __init__(self, ...):`

Define el *constructor* de la clase. En general, dentro del constructor se establecen los valores iniciales de todos los atributos.

`variable = NombreClase(...)`

Crea una nueva instancia de la clase. Los parámetros que se ingresen serán pasados al constructor, luego del parámetro especial `self`.

`variable.atributo`

Permite obtener o modificar el valor de un atributo de la instancia.

`def metodo(self, ...)`

El primer parámetro de cada método de una clase es una referencia a la instancia sobre la que va a operar el método. Se lo llama por convención `self`, pero puede tener cualquier nombre.

`variable.metodo(...)`

Invoca al método `metodo` de la clase de la cual `variable` es una instancia. El primer parámetro que se le pasa a `metodo` será `self` → `variable`.

`def __str__(self):`

Método especial que debe devolver una cadena de caracteres, con la representación "informal" de la instancia. Se invoca al hacer `str(variable)` o `print(variable)`.

`def __repr__(self):`

Método especial que debe devolver una cadena de caracteres, con la representación "formal" de la instancia. Se invoca al hacer `repr(variable)`.

`def __add__(self, otro):`, `def __sub__(self, otro):`

Métodos especiales para sobrecargar los operadores + y - respectivamente. Reciben las dos instancias sobre las que se debe operar, debe devolver una nueva instancia con el resultado.

`def __lt__(self, otro):`

Método especial para permitir la comparación de objetos mediante los operadores < y >. Recibe las dos instancias a comparar. Debe devolver `True` si `self` es comparativamente "menor" a `otro`.

`def __le__(self, otro):`

Método especial para permitir la comparación de objetos mediante los operadores <= y >=. Devuelve `True` si `self` es comparativamente "menor o igual" a `otro`.

`def __eq__(self, otro):`

Método especial para permitir la comparación de objetos mediante los operadores == y `!=`. Devuelve `True` si `self` y `otro` son comparativamente "iguales".

14.9 Ejercicios

Ejercicio 14.9.1. Mejorar la clase `Rectangulo`, agregando métodos para calcular las dimensiones alto y ancho, y las coordenadas del punto central.

Ejercicio 14.9.2. Modificar el método `__lt__` de `Hotel` para poder ordenar de menor a mayor las listas de hoteles según el criterio: primero por ubicación, en orden alfabético y dentro de cada ubicación por la relación calidad-precio.

Ejercicio 14.9.3.

a) Implementar la clase `Intervalo(desde, hasta)` que representa un intervalo entre dos instantes de tiempo (números enteros expresados en segundos), con la condición `desde < hasta`.

b) Implementar el método `duracion` que devuelve la duración en segundos del intervalo.

c) Implementar el método `interseccion` que recibe otro intervalo y devuelve un nuevo intervalo resultante de la intersección entre ambos, o lanzar una excepción si la intersección es nula.

d) Implementar el método `union` que recibe otro intervalo. Si los intervalos no son adyacentes ni intersectan, debe lanzar una excepción. En caso contrario devuelve un nuevo intervalo resultante de la unión entre ambos.

Ejercicio 14.9.4.

a) Crear una clase `Fraccion`, que cuente con dos atributos: `dividendo` y `divisor`, que se asignan en el constructor, y se imprimen como `X/Y` en el método `__str__`.

b) Implementar el método `__add__` que recibe otra fracción y devuelve una nueva fracción con la suma de ambas.

c) Implementar el método `__mul__` que recibe otra fracción y devuelve una nueva fracción con el producto de ambas.

d) Crear un método `simplificar` que modifica la fracción actual de forma que los valores del `dividendo` y `divisor` sean los menores posibles.

Ejercicio 14.9.5.

a) Crear una clase `Vector`, que en su constructor reciba una lista de elementos que serán sus coordenadas. En el método `__str__` se imprime su contenido con el formato `[x,y,z]`

b) Implementar el método `__add__` que reciba otro vector, verifique si tienen la misma cantidad de elementos y devuelva un nuevo vector con la suma de ambos. Si no tienen la misma cantidad de elementos debe levantar una excepción.

c) Implementar el método `__mul__` que reciba un número y devuelva un nuevo vector, con los elementos multiplicados por ese número.

Ejercicio 14.9.6. Escribir una clase `Caja` para representar cuánto dinero hay en una caja de un negocio, desglosado por tipo de billete (por ejemplo, en el quiosco de la esquina hay 6 billetes de 500 pesos, 7 de 100 pesos y 4 monedas de 2 pesos). Las denominaciones permitidas son 1, 2, 5, 10, 20, 50, 100, 200, 500 y 1000 pesos. Debe comportarse según el siguiente ejemplo:

```
>>> c = Caja({500: 6, 300: 7, 2: 4})
ValueError: Denominación "300" no permitida
>>> c = Caja({500: 6, 100: 7, 2: 4})
```

```
>>> str(c)
'Caja {500: 6, 100: 7, 2: 4} total: 3708 pesos'
>>> c.agregar({250: 2})
ValueError: Denominación "250" no permitida
>>> c.agregar({50: 2, 2: 1})
>>> str(c)
'Caja {500: 6, 100: 7, 50: 2, 2: 5} total: 3810 pesos'
>>> c.quitar({50: 3, 100: 1})
ValueError: No hay suficientes billetes de denominación "50"
>>> c.quitar({50: 2, 100: 1})
200
>>> str(c)
'Caja {500: 6, 100: 6, 2: 5} total: 3610 pesos'
```

Ejercicio 14.9.7. Crear las clases `Materia` y `Carrera`, que se comporten según el siguiente ejemplo:

```
>>> analisis2 = Materia("61.03", "Análisis 2", 8)
>>> fisica2 = Materia("62.01", "Física 2", 8)
>>> algo1 = Materia("75.40", "Algoritmos 1", 6)
>>> c = Carrera([analisis2, fisica2, algo1])
>>> str(c)
Créditos: 0 -- Promedio: N/A -- Materias aprobadas:
>>> c.aprobar("95.14", 7)
ValueError: La materia 75.14 no es parte del plan de estudios
>>> c.aprobar("75.40", 10)
>>> c.aprobar("62.01", 7)
>>> str(c)
Créditos: 14 -- Promedio: 8.5 -- Materias aprobadas:
75.40 Algoritmos 1 (10)
62.01 Física 2 (7)
```

Unidad 15

Polimorfismo, Herencia y Delegación

En esta unidad veremos algunos temas que son centrales a la programación orientada a objetos: polimorfismo, herencia y delegación.

15.1 Polimorfismo

El concepto de *polimorfismo* (del griego *muchas formas*) implica que si en una porción de código se invoca un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto. Esto se debe a que distintos objetos pueden tener un método con un mismo nombre, pero que realice distintas operaciones.

En las unidades anteriores, varias veces utilizamos las posibilidades provistas por el polimorfismo, sin haberle puesto este nombre. Algunos ejemplos:

- Es posible recorrer cualquier tipo de secuencia (ya sea una lista, una tupla, un diccionario, un archivo o cualquier otro tipo de secuencia) utilizando la misma estructura de código: `for elemento in secuencia`.
- Se puede obtener la cantidad de elementos de cualquier secuencia utilizando la misma función: `len(secuencia)`.
- Varias funciones y operaciones pueden trabajar con los distintos tipos numéricos sin hacer distinción sobre de qué tipo de número se trataba (entero, real, largo o complejo); por ejemplo `abs(numero)`.
- La función `str` permite convertir cualquier objeto a una representación en cadena de texto, sin importar el tipo del objeto.

Todos los casos mencionados son ejemplos de polimorfismo: la misma operación produce resultados diferentes (o el mismo resultado, pero que se calcula de manera diferente) según el tipo de objeto al que se aplica la operación.

15.1.1 Interfaz

Llamamos **interfaz** a un conjunto de funciones, métodos o atributos con nombres específicos. Una interfaz es un *contrato* entre el programador que realiza una clase y el que la utiliza, y puede consistir en uno solo o varios métodos o atributos.

Por ejemplo, para que un objeto se pueda sumar con otro utilizando el operador +, debe cumplir con la interfaz *sumable*, que en Python implica incluir el método `__add__` visto en la unidad anterior.

Una manera de implementar polimorfismo es utilizar distintos tipos de datos a través de una interfaz común. Por ejemplo, cuando sumamos dos objetos `a` y `b` con la expresión:

```
a + b
```

no es necesario que sepamos de qué tipo son los objetos; si alguno de los dos cumple con la interfaz *sumable*, podemos decir que la operación será exitosa y producirá algún resultado.

15.1.2 Redefinición de métodos

Llamamos **redefinición** a la acción de definir un método con el mismo nombre en distintas clases, de forma tal que provea una interfaz.

Un bloque de código será *polimórfico* cuando dentro de ese código se realicen llamadas a métodos que puedan estar redefinidos en distintas clases.

Tomemos por ejemplo el caso ya mencionado en el que se recorre una secuencia (lista, tupla, archivo, etc) mediante una misma estructura de código. Esto es posible gracias a la redefinición del método especial `__iter__`, que devuelve un *iterador*. Un bloque que utiliza una secuencia en forma genérica es, entonces, un bloque polimórfico.

Sabías que…

En Python, al no ser necesario especificar explícitamente el tipo de los parámetros que recibe una función, todas las funciones son naturalmente polimórficas.

En otros lenguajes puede darse que sólo algunas funciones específicas sean polimórficas (como en C++, por ejemplo), o que sea extremadamente difícil obtener un comportamiento polimórfico (como es el caso de C).

En la vida real, cuando analizamos las funciones de respiración, reproducción o alimentación de los seres vivos vemos que siempre se repite el mismo patrón: si bien la acción en todos los casos es la misma, puede suceder que haya diferencias en la *implementación* en cada tipo, ya que la respiración de una mojarrita que la de un malvón no funcionan de la misma manera, o tampoco así la reproducción de una ameba y la de un elefante.

Análogamente, al implementar nuestras clases podemos proveer distintas implementaciones para un método con el mismo nombre, para que puedan comportarse polimórficamente, como ser las redefiniciones de los métodos `__str__` o `__lt__`.

En particular en Python, la *sobrecarga de operadores* (mencionada en la sección 14.4.2), es un proceso que se realiza mediante la redefinición de algunos métodos especiales. En otros lenguajes se utilizan técnicas distintas para obtener el mismo resultado.

Ejemplos de polimorfismo

En la unidad anterior se vio la clase `Punto` que representa a un punto en el plano. Es posible definir también una clase `Punto3D`, que represente un punto en el espacio. Esta nueva clase contendrá los mismos métodos que se vieron para `Punto`, pero para tres coordenadas.

Si a ambas clases le agregamos un método para dividir por un escalar (`__div__(self, escalar)`), podríamos tener la siguiente función polimórfica:

Sabías que…

El término *sobrecarga* viene de un posible uso de polimorfismo que está presente en algunos lenguajes orientados a objetos: la posibilidad de tener, dentro de una misma clase, dos métodos que se llamen igual pero reciban parámetros de distintos tipos. Es decir, que el método al que hay que llamar se decide por el tipo del parámetro, no por el tipo del objeto que lo contiene.

En Python no tenemos sobrecarga de métodos, ya que al no definir los tipos de los parámetros en el encabezado, no sería posible distinguir a qué método hay que llamar. Sin embargo, se puede decir que sí tenemos sobrecarga de operadores, ya que al encontrar un operador, Python llamará a distintos métodos según el tipo de las variables que se quiera sumar, restar, multiplicar, etc.

```
def obtener_versor(punto):
    return punto / punto.norma()
```

Esta función devolverá un versor de dos dimensiones o de tres dimensiones, según a qué clase pertenezca la variable `punto`.

Atención

Que una función sea polimórfica no significa que pueda operar sobre parámetros de *cualquier* tipo de dato, sino que suele imponer restricciones. En el ejemplo anterior, las restricciones son que el objeto `punto` tenga el método `norma` y la posibilidad de dividirlo por un escalar.

Veamos otro ejemplo: una función que cuenta la frecuencia de aparación de elementos dentro de una secuencia cualquiera.

```
def frecuencias(secuencia):
    """Calcula las frecuencias de aparición de los elementos de
       la secuencia recibida.
       Devuelve un diccionario con elementos: {valor: frecuencia}
    """
    frec = {}
    for elemento in secuencia:
        frec[elemento] = frec.get(elemento, 0) + 1
    return frec
```

Vemos que el parámetro `secuencia` puede ser de cualquier tipo que se encuentre dentro de la "familia" de las secuencias. En cambio, si llamamos a la función con un entero se levanta una excepción.

```
>>> frecuencias(["peras", "manzanas", "peras", "manzanas", "uvas"])
{'uvas': 1, 'peras': 2, 'manzanas': 2}
>>> frecuencias((1, 3, 4, 2, 3, 1))
{1: 2, 2: 1, 3: 2, 4: 1}
>>> frecuencias("Una frase")
{'a': 2, ' ': 1, 'e': 1, 'f': 1, 'n': 1, 's': 1, 'r': 1, 'U': 1}
>>> frecuencias(range(3, 10, 2))
{9: 1, 3: 1, 5: 1, 7: 1}
>>> frecuencias(4)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    frecuencias(4)
```

```
  File "frecuencias.py", line 12, in frecuencias
    for elemento in secuencia:
TypeError: 'int' object is not iterable
```

15.2 Herencia

Otro mecanismo de la programación orientada a objetos que permite implementar comportamientos polimórficos es la *herencia*, que permite crear clases nuevas a partir de clases preexistentes. Mediante la herencia, una clase puede *extender* a otra y de esta manera *heredar* sus atributos y comportamientos, pudiendo modificarlos para implementar un comportamiento diferente.

La clase vieja se llama *clase base* y la que se construye a partir de ella es una *clase derivada*.

Por ejemplo, a partir de una clase `Persona` con los atributos `identificacion`, `nombre` y `apellido`, podemos construir la clase `AlumnoFIUBA` que extiende a `Persona` y agrega como atributo el `padron`.

Definamos primero la clase base, `Persona`:

```
class Persona:
    def __init__(self, identificacion, nombre, apellido):
        self.identificacion = identificacion
        self.nombre = nombre
        self.apellido = apellido

    def __str__(self):
        return "{}: {}, {}".format(
            self.identificacion, self.apellido, self.nombre
        )
```

A continuación definimos `AlumnoFIUBA`, indicando entre paréntesis que la clase base es `Persona`:

```
class AlumnoFIUBA(Persona):
    def __init__(self, identificacion, nombre, apellido, padron):
        super().__init__(identificacion, nombre, apellido)
        self.padron = padron
```

En la primera línea del constructor invocamos al constructor de `Persona`, pasándole los parámetros necesarios (si no hiciéramos esto, las instancias de `AlumnoFIUBA` no tendrían los atributos `identificacion`, `nombre` o `apellido`). En la segunda línea asignamos el atributo `padron`.

Probamos la nueva clase:

```
>>> a = AlumnoFIUBA("DNI 35123456", "Damien", "Thorn", "98765")
>>> str(a)
'DNI 35123456: Thorn, Damien'
```

Vemos que se heredó el método `__str__` de la clase base. Si queremos, podemos redefinirlo:

```
    def __str__(self):
        return "Padrón {}: {}, {}".format(
            self.padron, self.apellido, self.nombre
        )
```

```
>>> a = AlumnoFIUBA("DNI 35123456", "Damien", "Thorn", "98765")
>>> str(a)
'Padrón 98765: Thorn, Damien'
```

De una clase base se pueden construir muchas clases derivadas: así como hemos derivado alumnos, podríamos derivar docentes, empleados, clientes, proveedores, o lo que fuera necesario según la aplicación que estemos desarrollando. Incluso podríamos aplicar más de un nivel de herencia: `B` extiende `A`, `C` extiende `B`, …

Sabías que…

En el diseño de jerarquias de herencia no siempre es del todo fácil decidir cuándo una clase debe extender a otra. Una regla práctica para decidir si una clase `S` puede ser definida como heredera de otra `T` es que debe cumplirse que "S es un T". Por ejemplo, *Perro* es un *Animal*, pero *Vehículo* no es un *Motor*.

Esta regla se desprende del principio de sustitución de Liskov (formulado por Barbara Liskov y Jeannette Wing).

Barbara Liskov es una mujer importante en la historia de la informática, no sólo por este principio, sino que fue la primera mujer en recibir un doctorado en las ciencias de la computación, creadora de varios lenguajes y actualmente es profesora e investigadora del MIT.

En el caso de Python, también es posible construir una clase derivada a partir de varias clases base. Por ejemplo, un ayudante de segunda en la UBA es un alumno que también trabaja de docente. Esta posbilidad se llama *herencia múltiple*.

Ejemplo de herencia

Un ejemplo clásico de herencia es el de las figuras geométricas: todas las figuras tienen un nombre y un color, pero cada figura diferente puede tener atributos diferentes como radio, base, altura, etc., y una fórmula diferente para calcular el área.

- La clase base:

```
class Figura:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color

    def area(self):
        raise NotImplementedError("Este método debe ser redefinido.")

    def __str__(self):
        return "<{} - color: {} - área: {:.2f}>".format(
            self.nombre, self.color, self.area()
        )
```

- Los círculos:

```
from math import pi

class Circulo(Figura):
    def __init__(self, radio, color):
        super().__init__("Círculo", color)
        self.radio = radio

    def area(self):
```

```
        return pi * self.radio * self.radio
```

- Y los triángulos:

```
class Triangulo(Figura):
    def __init__(self, base, altura, color):
        super().__init__("Triángulo", color)
        self.base = base
        self.altura = altura

    def area(self):
        return self.base * self.altura / 2
```

Y ahora las pruebas:

```
>>> c = Circulo(4, "verde")
>>> str(c)
'<Círculo - color: verde - área: 50.27>'
>>> t = Triangulo(3, 5, "azul")
>>> str(c)
'<Triángulo - color: azul - área: 7.5>'
```

15.3 Delegación

Llamamos delegación a la situación en la que una clase contiene (como atributos) una o más instancias de otra clase, a las que *delegará* parte de sus funcionalidades. Esta relación entre clases suele ser la más indicada cuando es necesaria una asociación entre las clases pero el principio de Liskov no se cumple. También puede verse como la relación entre clases "S contiene a T". Por ejemplo, Vehículo **contiene** un Motor, pero Alumno no contiene a Persona, sino que **es** una Persona.

Por ejemplo, la clase `Hotel` vista en la unidad anterior, podría contener una instancia de `Disponibilidad`, que almacene la disponibilidad de las habitaciones del hotel para distintas fechas. La clase `Hotel` debería tener, entonces, los métodos `consultar_disponibilidad`, `reservar` y `cancelar`, que todos delegarían en la clase `Disponibilidad` su funcionamiento principal.

Delegación y Referencias

Queremos construir una clase `Rectangulo`, que se describe mediante los siguientes atributos:

- **Longitud de su base**: un número.
- **Longitud de su altura**: un número.
- **El punto del plano de su esquina inferior izquierda**: un punto del plano.

Incluiremos métodos para inicializar y mostrar, para calcular el área y para trasladar el rectángulo en el plano.

La implementación básica puede verse en el Código 15.1. Se puede ver que el rectángulo realiza internamente la operación para calcular el área, pero para la operación `trasladar`, delega la suma de los puntos al operador `__add__` de la clase `Punto` (recordamos que cuando se hace `self.origen + Punto(dx,dy)`, Python llama al método `__add__` de la clase `Punto`, que recibe los dos puntos y devuelve un nuevo punto con la suma de ambos).

Para construir y utilizar el rectángulo, lo haremos de la siguiente forma:

Código 15.1 `Rectangulo.py`**:** Clase para modelar un Rectángulo

```
from Punto import Punto

class Rectangulo:
    """Representa un rectángulo en el plano.

       Atributos:
        origen (Punto) posición de la esquina inferior izquierda.
        base (número): longitud de su base,
        altura (número): longitud de su altura,
    """
    def __init__(self, origen, base, altura):
        self.origen = origen
        self.base = base
        self.altura = altura

    def trasladar(self, dx, dy):
        self.origen = self.origen + Punto(dx,dy)

    def area(self):
        return self.base * self.altura

    def __str__(self):
        return "Origen: {}, Base: {}, Altura: {} ".format(
            str(self.origen), self.base, self.altura
        )
```

```
>>> from Punto import Punto
>>> from Rectangulo import Rectangulo
>>> r = Rectangulo(Punto(1, 2), 2, 3)
>>> str(r)
'Origen: (1, 2), Base: 2, Altura: 3'
>>> r.area()
6
```

Lo que acabamos de crear es un objeto de acuerdo al siguiente diagrama que se muestra en la Figura 15.1.

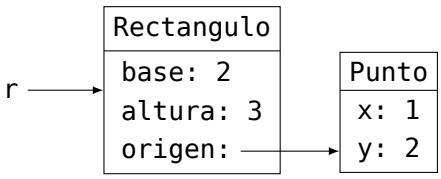


Figura 15.1: Estado de las variables, al momento de crear el rectángulo

El punto que describe la posición de la esquina inferior izquierda del rectángulo es un objeto `Punto`. El atributo `origen` contiene una *referencia* a dicho objeto.

Utilizando el método `trasladar`, podemos modificar el valor del punto contenido dentro del rectángulo.

```
>>> r.trasladar(2, 4)
```

```
>>> str(r)
'Origen: (3, 6), Base: 2, Altura: 3'
```

También es posible directamente reemplazar el punto contenido, por un nuevo punto.

```
>>> q = Punto(7, 2)
>>> r.origen = q
>>> str(r)
'Origen: (7, 2), Base: 2, Altura: 3'
```

Con lo cual el diagrama pasa a ser el de la Figura 15.2.

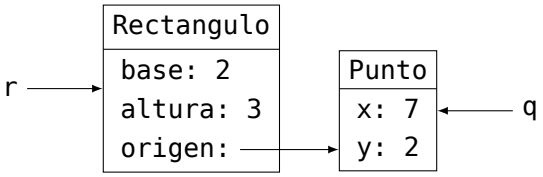


Figura 15.2: Estado de las variables, luego de reemplazar el origen

El `Punto(1, 2)` y `Punto(3,6)` que habían sido creados previamente, están ahora fuera de uso, por lo que quedan a disposición de un mecanismo de *recolección de basura* que se encarga de recuperar automáticamente las secciones de memoria que quedan fuera de uso durante la ejecución de un programa.

15.4 Resumen

- Se llama **polimorfismo** a la posibilidad de obtener distintos comportamientos mediante la invocación a métodos de un mismo nombre, pero de clases distintas.
- Se llama **herencia** a la relación entre clases en la cual una es una clase base y otra es una clase derivada, que *hereda* los métodos y atributos de la clase base.
- Se llama **delegación** a la relación entre clases en la cual una instancia de una clase contiene como atributo una instancia de otra clase, y dentro de sus métodos realiza invocaciones a los métodos de la clase contenida.
- Se denomina **referencia** a las variables que permiten acceder a un determinado objeto, ya sea un atributo dentro de un objeto, o una variable en una porción de código cualquiera.

15.5 Ejercicios

Ejercicio 15.5.1. Papel, Birome, Marcador

a) Escribir una clase `Papel` que contenga un texto, un método `escribir`, que reciba una cadena para agregar al texto, y el método `__str__` que imprima el contenido del texto.

b) Escribir una clase `Birome` que contenga una cantidad de tinta, y un método `escribir`, que reciba un texto y un papel sobre el cual escribir. Cada letra escrita debe reducir la cantidad de tinta contenida. Cuando la tinta se acabe, debe lanzar una excepción.

c) Escribir una clase `Marcador` que herede de `Birome`, y agregue el método `recargar`, que reciba la cantidad de tinta a agregar.

Ejercicio 15.5.2. Juego de Rol

a) Escribir una clase `Personaje` que contenga los atributos `vida`, `posicion` y `velocidad`, y los métodos `recibir_ataque`, que reduzca la vida según una cantidad recibida y lance una excepción si la vida pasa a ser menor o igual que cero, y `mover` que reciba una dirección y se mueva en esa dirección la cantidad indicada por velocidad.

b) Escribir una clase `Soldado` que herede de `Personaje`, y agregue el atributo `ataque` y el método `atacar`, que reciba otro personaje, al que le debe hacer el daño indicado por el atributo ataque.

c) Escribir una clase `Campesino` que herede de `Personaje`, y agregue el atributo `cosecha` y el método `cosechar`, que devuelva la cantidad cosechada.

Unidad 16

Listas enlazadas

En esta unidad nos dedicaremos a construir nuestras propias listas, que consistirán de cadenas de objetos enlazadas mediante referencias, como las vistas en la unidad anterior.

Si bien Python ya cuenta con sus propias listas, las listas enlazadas que implementaremos en esta unidad nos resultarán también útiles.

16.1 Una clase sencilla de *vagones*

En primer lugar, definiremos una clase muy simple, `Nodo`, que se comportará como un vagón: tendrá sólo dos atributos: `dato`, que servirá para almacenar cualquier información, y `prox`, que servirá para poner una referencia al siguiente vagón.

Además, como siempre, implementaremos el constructor y el método `__str__` para poder obtener una representación en cadena de texto.

```
class Nodo:
    def __init__(self, dato=None, prox=None):
        self.dato = dato
        self.prox = prox

    def __str__(self):
        return str(self.dato)
```

Sabías que…

Al implementar una función o un método, Python nos permite definir valores por omisión para sus parámetros, con la notación `parametro=valor`. Por ejemplo, si definimos la función:

```
def saludar(nombre="?"):
    return "¡Hola, {}!".format(nombre)
```

al invocarla podemos omitir el parámetro `nombre`, en cuyo caso se le asignará el valor `"?"`:

```
>>> saludar("Alan")
'¡Hola, Alan!'
>>> saludar()
'¡Hola, ?!'
```

Ejecutamos este código:

```
>>> n3 = Nodo("Bananas")
>>> n2 = Nodo("Peras", n3)
>>> n1 = Nodo("Manzanas", n2)
>>> str(n1)
'Manzanas'
>>> str(n2)
'Peras'
>>> str(n3)
'Bananas'
```

Con esto hemos generado la estructura de la Figura 16.1.

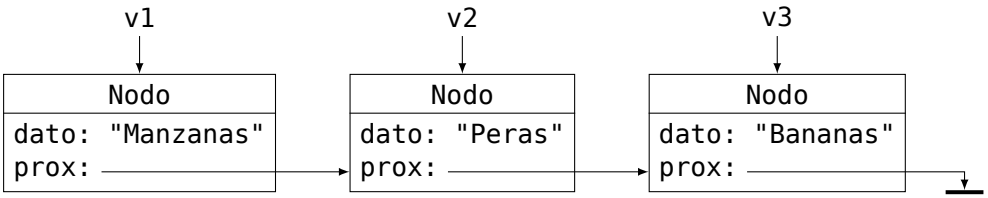


Figura 16.1: Nodos enlazados

El atributo `prox` de `n3` tiene una referencia nula, lo que indica que `n3` es el último vagón de nuestra estructura.

Hemos creado una lista en forma manual. Si nos interesa recorrerla, podemos hacer lo siguiente:

```
def ver_lista(nodo):
    """Recorre todos los nodos a través de sus enlaces,
       mostrando sus contenidos."""

    while nodo is not None:
        print(nodo)
        nodo = nodo.prox
```

```
>>> ver_lista(n1)
Manzanas
Peras
Bananas
```

Es interesante notar que la estructura del recorrido de la lista es el siguiente:

- Se le pasa a la función sólo la referencia al primer nodo.
- El resto del recorrido se consigue siguiendo la cadena de referencias dentro de los nodos.

Si se desea *desenganchar* un vagón del medio de la lista, alcanza con cambiar la referencia `prox`:

```
>>> n1.prox = n3
>>> ver_lista(n1)
Manzanas
Bananas
>>> n1.prox = None
>>> ver_lista(n1)
Manzanas
```

De esta manera también se pueden generar estructuras impensables: ¿qué sucede si escribimos `n1.prox = n1`? La representación es finita y sin embargo en este caso `ver_lista(n1)` no termina nunca. Hemos creado una *lista infinita*, también llamada *lista circular*.

16.1.1 Caminos

En una lista implementada con nodos, si seguimos las flechas dadas por las referencias, obtenemos un *camino* en la lista.

Los caminos cerrados se denominan *ciclos*. Son ciclos, por ejemplo, la autorreferencia de `n1` a `n1`, como así también una flecha de `n1` a `n2` seguida de una flecha de `n2` a `n1`.

> **Atención**
>
> Las listas circulares no tienen nada de malo en sí mismas, mientras su representación sea finita. El problema, en cambio, es que debemos tener mucho cuidado al escribir programas para recorrerlas, ya que el recorrido debe ser acotado (por ejemplo no habría problema en ejecutar un programa que liste los 20 primeros nodos de una lista circular).
>
> Cuando una función recibe una lista y el recorrido no está acotado, se debe aclarar en su precondición que la ejecución de la misma terminará sólo si la lista no contiene ciclos. Ése es el caso de la función `ver_lista(n1)`.

16.1.2 Referenciando el principio de la lista

Una cuestión no contemplada hasta el momento es la de mantener una referencia a la lista completa. Por ahora para nosotros la lista es la colección de nodos que se enlazan a partir de `n1`. Sin embargo puede suceder que queramos quitar a `n1` y continuar con el resto de la lista como la colección de nodos a tratar.

Una solución muy simple es asociar una referencia al principio de la lista, que llamaremos `lista`, y que mantendremos independientemente de cuál sea el nodo que está al principio de la lista:

```
>>> n3 = Nodo("Bananas")
>>> n2 = Nodo("Peras", n3)
>>> n1 = Nodo("Manzanas", n2)
>>> lista = n1
>>> ver_lista(lista)
Manzanas
Peras
Bananas
```

Ahora sí estamos en condiciones de eliminar el primer elemento de la lista sin perder la identidad de la misma:

```
>>> lista = lista.prox
>>> ver_lista(lista)
Peras
Bananas
```

16.2 Tipos abstractos de datos

Los tipos nuevos que habíamos definido en unidades anteriores fueron tipos de datos concretos: un punto se definía como un par ordenado de números, un hotel se definía por dos

cadenas de caracteres (nombre y unicación) y dos números (calidad y precio), etc.

Vamos a ver ahora una nueva manera de definir datos: por las operaciones que tienen y por lo que tienen que hacer esas operaciones (cuál es el resultado esperado de esas operaciones).

Esa manera de definir datos se conoce como *tipos abstractos de datos* o *TADs*.

Lo novedoso de este enfoque respecto del anterior es que en general se puede encontrar más de una representación mediante tipos concretos para representar el mismo TAD, y que se puede elegir la representación más conveniente en cada caso, según el contexto de uso.

Los programas que los usan hacen referencia a las operaciones que tienen, no a la representación, y por lo tanto ese programa sigue funcionando si se cambia la representación.

Dentro del ciclo de vida de un TAD hay dos fases: la programación del TAD y la construcción de los programas que lo usan.

Durante la fase de programación del TAD, habrá que elegir una representación, y luego programar cada uno de los métodos sobre esa representación.

Durante la fase de construcción de los programas, no será relevante para el programador que utiliza el TAD cómo está implementado, sino únicamente los métodos que posee.

> Utilizando el concepto de *interfaz* visto en la unidad anterior, podemos decir que a quien utilice el TAD sólo le interesará la interfaz que éste ofrezca.

16.3 La clase `ListaEnlazada`

Basándonos en los nodos implementados anteriormente, pero buscando desligar al programador que desea usar la lista de la responsabilidad de manipular las referencias, definiremos ahora la clase `ListaEnlazada`, de modo tal que no haya que operar mediante las referencias internas de los nodos, sino que se lo pueda hacer a través de operaciones de lista.

Más allá de la implementación en particular, se podrá notar que implementaremos los mismos métodos de las listas de Python, de modo que más allá del funcionamiento interno, ambas serán **listas**.

Definimos a continuación las operaciones que inicialmente deberá cumplir la clase `ListaEnlazada`.

- `__str__`, para obtener una representación en cadena de texto.
- `__len__`, para calcular la longitud de la lista.
- `append(x)`, para agregar un elemento al final de la lista.
- `insert(i, x)`, para agregar el elemento `x` en la posición `i` (levanta una excepción si la posición `i` es inválida).
- `remove(x)`, para eliminar la primera aparición de `x` en la lista (levanta una excepción si `x` no está).
- `pop([i])`, para eliminar el elemento que está en la posición `i` y devolver su valor. Si no se especifica el valor de `i`, `pop()` elimina y devuelve el elemento que está en el último lugar de la lista (levanta una excepción si se hace referencia a una posición no válida de la lista).
- `index(x)`, devuelve la posición de la primera aparición de `x` en la lista (levanta una excepción si `x` no está).

Más adelante podrá agregarse a la lista otros métodos que también están implementados por las listas de Python.

Valen ahora algunas consideraciones más antes de empezar a implementar la clase:

- Por lo dicho anteriormente, es claro que la lista deberá tener como atributo la referencia al primer nodo que la compone.
- Una implementación trivial del método `__len__` podría recorrer todos los nodos de la lista y contar la cantidad de elementos, pero si la lista tiene muchos elementos esto podría ser poco eficiente.

  Para mejorar la eficiencia alcanza con agregar un atributo numérico que contenga la cantidad de nodos. Así, este atributo que llamaremos `len` se inicializará en 0 cuando se cree la lista vacía, se incrementará en 1 cada vez que se agregue un elemento y se decrementará en 1 cada vez que se elimine un elemento.

  > **Atención**
  >
  > Al agregar el atributo `len` estamos *duplicando información*, ya que habrá dos formas de obtener la longitud de la lista:
  >
  > 1. contar los nodos
  > 2. obtener el valor de `len`
  >
  > Como consecuencia, vamos a tener que prestar especial atención para que el atributo `len` siempre contenga un valor consistente; es decir que su valor sea siempre igual a la cantidad de nodos que contiene la lista.
  >
  > En la sección 16.4 se explica más formalmente este concepto.

- Por otro lado, como vamos a incluir todas las operaciones de listas que sean necesarias para operar con ellas, no es necesario que la clase `Nodo` esté disponible para que otros programadores puedan modificar (y romper) las listas a voluntad usando operaciones de nodos. Para eso incluiremos la clase `Nodo` de manera *privada* (es decir oculta), de modo que la podamos usar nosotros como dueños (fabricantes) de la clase, pero no cualquier programador que utilice la lista.

Python tiene una convención para hacer que atributos, métodos o clases dentro de una clase dada no puedan ser usados por los usuarios, y sólo tengan acceso a ellos quienes programan la clase: su nombre tiene que empezar con un guión bajo y terminar sin guión bajo. Así que para hacer que los nodos sean privados, cambiaremos el nombre de la clase a `_Nodo`.

> Se trata sólo de una convención, aun con el nombre `_Nodo` la clase está disponible, pero respetaremos esa convención de aquí en adelnte.

16.3.1 Construcción de la lista

Empezamos escribiendo la clase con su constructor.

```
class ListaEnlazada:
    """Modela una lista enlazada."""
```

```
def __init__(self):
    """Crea una lista enlazada vacía."""
    # referencia al primer nodo (None si la lista está vacía)
    self.prim = None
    # cantidad de elementos de la lista
    self.len = 0
```

Nuestra estructura ahora será como la representada por la Figura 16.2.

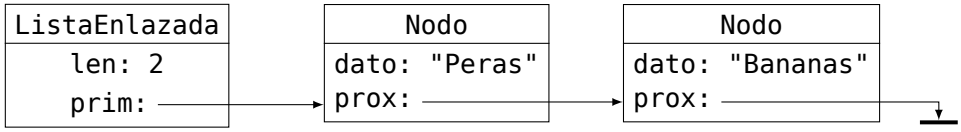


Figura 16.2: Una lista enlazada

Ejercicio 16.1. Escribir los métodos `__str__` y `__len__` para la lista.

Sabías que…

Una característica importante de la implementación de listas enlazadas es que eliminar el primer elemento es una operación de *tiempo constante*, es decir que no depende de la longitud de la lista. En las listas de Python, por contraste, esta operación requiere un *tiempo proporcional a la longitud de la lista*.

Sin embargo no todo es tan positivo: el acceso a la posición p se realiza en *tiempo proporcional a* p, mientras que en las listas de Python esta operación se realiza en *tiempo constante*.

Conociendo las ventajas y desventajas podremos elegir el tipo de lista que necesitemos según los requerimientos de cada problema.

16.3.2 Eliminar un elemento de una posición

Analizaremos a continuación `pop([i])`, que elimina el elemento que está en la posición `i` y devuelve su valor. Si no se especifica el valor de `i`, `pop()` elimina y devuelve el elemento que está en el último lugar de la lista. Por otro lado, levanta una excepción si se hace referencia a una posición no válida de la lista.

Dado que se trata de una función con cierta complejidad, separaremos el código en las diversas consideraciones a tener en cuenta.

- Si no se indica la posición, `i` toma la última posición de la lista:

```
if i is None:
    i = self.len - 1
```

- Si la posición es inválida (`i` menor que 0 o mayor o igual a la longitud de la lista), se considera error y se levanta la excepción `IndexError`.

  Esto se resuelve con este fragmento de código:

```
if i < 0 or i >= self.len:
    raise IndexError("Índice fuera de rango")
```

- Cuando la posición es 0 se trata de un caso particular, ya que en ese caso hay que cambiar la referencia de `self.prim` para que apunte al nodo siguiente. Es decir, pasar de `self.prim` → `nodo0` → `nodo1` a `self.prim` → `nodo1`).

```
if i == 0:
    dato = self.prim.dato
    self.prim = self.prim.prox
```

  (Guardamos el `dato` del nodo descartado para poder devolverlo al finalizar la función).

- Vemos ahora el caso general:

  Mediante un ciclo, se deben ubicar los nodos n_{i-1} y n_i que están en las posiciones $i-1$ e i de la lista, respectivamente, de modo de poder ubicar no sólo el nodo que se descartará, sino también estar en condiciones de saltear el nodo descartado en los enlaces de la lista. La lista debe pasar de contener el camino $n_{i-1} \to n_i \to n_{i+1}$ a contener el camino $n_{i-1} \to n_{i+1}$.

  Nos basaremos un esquema muy simple y útil que se denomina *máquina de parejas*:

  Si nuestra secuencia tiene la forma *ABCDE*, se itera sobre ella de modo de tener las parejas *AB*, *BC*, *CD*, *DE*. En la pareja *XY*, llamaremos a *X* el *elemento anterior* y a *Y* el *elemento actual*. En general estos ciclos terminan o bien cuando no hay más parejas que formar, o bien cuando el elemento actual cumple con una determinada condición.

  En nuestro problema, tenemos la siguiente situación:

  – Las parejas son parejas de nodos.
  – Para avanzar en la secuencia se usa la referencia al próximo nodo de la lista.
  – La condición de terminación es siempre que la posición del nodo en la lista sea igual al valor buscado. En este caso particular no debemos preocuparnos por la terminación de la lista porque la validez del índice buscado ya fue verificada más arriba.

  Esta es la porción de código correspondiente a la búsqueda. Llamamos `n_ant` y `n_act` a los elementos anterior y actual de la pareja de nodos:

```
n_ant = self.prim
n_act = n_ant.prox
for pos in range(1, i):
    n_ant = n_act
    n_act = n_ant.prox
```

  Al finalizar el ciclo, `n_ant` será una referencia al nodo $i-1$ y `n_act` una referencia al nodo i.

  Una vez obtenidas las referencias, se obtiene el dato y se cambia el camino para descartar el nodo `n_act`:

```
dato = n_act.dato
n_ant.prox = n_act.prox
```

- Finalmente, en todos los casos de éxito se debe devolver el dato que contenía el nodo descartado y decrementar la longitud en 1:

```
self.len -= 1
return dato
```

Finalmente, en el Código 16.1 se incluye el código completo del método `pop`.

Código 16.1 pop: Método pop de la lista enlazada

```
def pop(self, i=None):
    """Elimina el nodo de la posición i, y devuelve el dato contenido.
       Si i está fuera de rango, se levanta la excepción IndexError.
       Si no se recibe la posición, devuelve el último elemento."""

    if i is None:
        i = self.len - 1

    if i < 0 or i >= self.len:
        raise IndexError("Índice fuera de rango")

    if i == 0:
        # Caso particular: saltear la cabecera de la lista
        dato = self.prim.dato
        self.prim = self.prim.prox
    else:
        # Buscar los nodos en las posiciones (i-1) e (i)
        n_ant = self.prim
        n_act = n_ant.prox
        for pos in range(1, i):
            n_ant = n_act
            n_act = n_ant.prox

        # Guardar el dato y descartar el nodo
        dato = n_act.dato
        n_ant.prox = n_act.prox

    self.len -= 1
    return dato
```

16.3.3 Eliminar un elemento por su valor

Análogamente se resuelve `remove(self, x)`, que debe eliminar la primera aparición de x en la lista, o bien levantar una excepción si x no se encuentra en la lista.

Nuevamente, dado que se trata de un método de cierta complejidad, lo resolveremos por partes, teniendo en cuenta los casos particulares y el caso general.

- Si la lista está vacía levantamos una excepción. Tenemos que tratarlo como un caso particular, ya que en todos los siguientes casos necesitamos que haya al menos un nodo.

```
if self.prim is None
    raise ValueError("La lista está vacía")
```

- El caso en el que x está en el primer nodo también es particular, ya que modificar la referencia `self.prim`:

```
if self.prim.dato == x:
    self.prim = self.prim.prox
```

- El caso general también implica un recorrido con máquina de parejas, sólo que esta vez la condición de terminación es: o bien la lista se terminó o bien encontramos un nodo con el valor x buscado.

```
n_ant = self.prim
n_act = n_ant.prox
while n_act is not None and n_act.dato != x:
    n_ant = n_act
    n_act = n_ant.prox
```

En este caso, al terminarse el ciclo será necesario corroborar si se terminó porque llegó al final de la lista, y de ser así levantar una excepción; o si se terminó porque encontró el dato, y de ser así eliminarlo.

```
if n_act is None:
    raise ValueError("El valor no está en la lista.")
n_ant.prox = n_act.prox
```

- Finalmente, en todos los casos de éxito debemos decrementar en 1 el valor de `self.len`.

En el Código 16.2 se incluye el código completo del método `remove`.

Código 16.2 `remove`**:** Método remove de la lista enlazada

```
def remove(self, x):
    """Borra la primera aparición del valor x en la lista.
       Si x no está en la lista, levanta ValueError"""

    if self.len == 0:
        raise ValueError("Lista vacía")

    if self.prim.dato == x:
        # Caso particular: saltear la cabecera de la lista
        self.prim = self.prim.prox
    else:
        # Buscar el nodo anterior al que contiene a x (n_ant)
        n_ant = self.prim
        n_act = n_ant.prox
        while n_act is not None and n_act.dato != x:
            n_ant = n_act
            n_act = n_ant.prox

        if n_act == None:
            raise ValueError("El valor no está en la lista.")

        # Descartar el nodo
        n_ant.prox = n_act.prox

    self.len -= 1
```

16.3.4 Insertar nodos

Debemos programar ahora `insert(i, x)`, que debe agregar el elemento `x` en la posición `i` (y levantar una excepción si la posición `i` es inválida).

Veamos qué debemos tener en cuenta para programar esta función.

- Si se intenta insertar en una posición menor que cero o mayor que la longitud de la lista debe levantarse una excepción.

```
if i < 0 or i > self.len:
    raise IndexError("Posición inválida")
```

- Para los demás casos hay que crear un nodo, que será el que se insertará en la posición que corresponda. Construimos un nodo `nuevo` cuyo `dato` será `x`.

```
nuevo = _Nodo(x)
```

- Si se quiere insertar en la posición 0, hay que cambiar la referencia de `self.prim`.

```
if i == 0:
    nuevo.prox = self.prim
    self.prim = nuevo
```

- Para los demás casos, nuevamente será necesaria la máquina de parejas. Obtenemos el nodo anterior a la posición en la que queremos insertar.

```
n_ant = self.prim
for pos in range(1, i):
    n_ant = n_ant.prox

nuevo.prox = n_ant.prox
n_ant.prox = nuevo
```

- En todos los casos de éxito se debe incrementar en 1 la longitud de la lista.

En el Código 16.3 se incluye el código resultante del método `insert`.

Ejercicio 16.2. Completar la clase `ListaEnlazada` con los métodos que faltan: `append` e `index`.

Ejercicio 16.3. En los bucles de *máquina de parejas* mostrados anteriormente, no siempre es necesario tener la referencia al nodo actual, puede alcanzar con la referencia al nodo anterior. Donde sea posible, eliminar la referencia al nodo actual. Una vez hecho esto, analizar el código resultante, ¿Es más elegante?

Ejercicio 16.4. Mantenimiento: Con esta representación conseguimos que la inserción en la posición 0 se realice en tiempo constante, sin embargo ahora `append` es lineal en la longitud de la lista. Si queremos mejorar esto, debemos agregar un atributo más a los objetos de la clase: la referencia al último nodo, y modificar `append` para que se pueda ejecutar en tiempo constante. Por supuesto que además hay que modificar todos los métodos de la clase para que se mantenga la propiedad de que ese atributo siempre es una referencia al útimo nodo.

16.4 Invariantes de objetos

Los invariantes son condiciones que deben ser siempre ciertas. En la sección 10.2 mencionamos los invariantes de ciclos, que son condiciones que deben permanecer ciertas durante la ejecución de un ciclo. Existen también los invariantes de objetos, que son condiciones que deben ser ciertas a lo largo de toda la existencia de un objeto.

La clase `ListaEnlazada` presentada en la sección anterior, cuenta con dos invariantes que siempre debemos mantener. Por un lado, el atributo `len` debe contener siempre la cantidad de

Código 16.3 `insert`**:** Método insert de la lista enlazada

```
def insert(self, i, x):
    """Inserta el elemento x en la posición i.
       Si la posición es inválida, levanta IndexError"""

    if i < 0 or i > self.len:
        raise IndexError("Posición inválida")

    nuevo = _Nodo(x)

    if i == 0:
        # Caso particular: insertar al principio
        nuevo.prox = self.prim
        self.prim = nuevo
    else:
        # Buscar el nodo anterior a la posición deseada
        n_ant = self.prim
        for pos in range(1, i):
            n_ant = n_ant.prox

        # Intercalar el nuevo nodo
        nuevo.prox = n_ant.prox
        n_ant.prox = nuevo

    self.len += 1
```

nodos de la lista. Es decir, siempre que se modifique la lista, agregando o quitando un nodo, se debe actualizar `len` como corresponda.

Por otro lado, el atributo `prim` referencia siempre al primer nodo de la lista. Si se agrega o elimina este primer nodo, es necesario actualizar esta referencia.

Cuando se desarrolla una estructura de datos como la lista enlazada, es importante destacar cuáles serán sus invariantes, ya que en cada método habrá que tener especial cuidado de que los invariantes permanezcan siempre ciertos.

Así, si se modifica la lista para que la inserción al final pueda hacerse en tiempo constante (como se pide en el ejercicio 16.4), se está agregando a la lista un nuevo invariante (un atributo de la lista que apunte siempre al último elemento) y no es sólo el método `append` el que hay que modificar, sino todos los métodos que puedan de una u otra forma cambiar la referencia al último elemento de la lista.

16.5 Otras listas enlazadas

Las listas presentadas hasta aquí son las *listas simplemente enlazadas*, que son sencillas y útiles cuando se quiere poder insertar o eliminar nodos de una lista en tiempo constante.

Existen otros tipos de listas enlazadas, cada uno con sus ventajas y desventajas.

Listas doblemente enlazadas

Las listas doblemente enlazadas son aquellas en que los nodos cuentan no sólo con una referencia al siguiente, sino también con una referencia al anterior. Esto permite que la lista

pueda ser recorrida en ambas direcciones.

En una lista doblemente enlazada es posible, por ejemplo, eliminar un nodo sin necesidad de saber cuál es el anterior.

Entre las desventajas podemos mencionar que al tener que mantener dos referencias el código se vuelve más complejo, y también que ocupa más espacio en memoria.

Listas circulares

Las listas circulares, que ya fueron mencionadas al comienzo de esta unidad, son aquellas en las que el último nodo contiene una referencia al primero. Pueden ser tanto simplemente como doblemente enlazadas.

Se las utiliza para modelar situaciones en las cuales los elementos no tienen un primero o un último, sino que forman una cadena infinita, que se recorre una y otra vez.

Sabías que…

El código del kernel Linux, que está programado en C, incluye una implementación de lista enlazada circular utilizada en la mayoría de los subsistemas.

Por ejemplo, la lista de tareas que se están ejecutando es una lista circular. El *scheduler* (planificador) del kernel permite que cada tarea utilice el procesador durante una porción de tiempo y luego pasa a la siguiente, aplicando así una "ronda de turnos" sin que haya una primera o una última tarea.

16.6 Iteradores

En la unidad anterior se hizo referencia a que todas las secuencias pueden ser recorridas mediante una misma estructura (`for variable in secuencia`), ya que todas implementan el método especial `__iter__`. Este método debe devolver un *iterador* capaz de recorrer la secuencia como corresponda.

Un iterador es un objeto que permite recorrer uno a uno los elementos almacenados en una estructura de datos, y operar con ellos.

Un ejemplo de iteradores que estuvimos usando desde la primera unidad es la función `range`:

```
>>> r = range(3)
>>> type(r)
<class 'range'>
```

Lo primero que observamos es que la función `range` no devuelve una lista de números, sino que devuelve una instancia de la clase `range`. Para obtener un iterador a partir de este objeto podemos aplicar la función `iter` (que a su vez llamará al método `__iter__`):

```
>>> i = iter(r)
>>> type(i)
<class 'range_iterator'>
```

En Python los iteradores implementan un método `__next__` que debe devolver los elementos, de a uno por vez, comenzando por el primero. Y al llegar al final de la estructura, debe levantar una excepción de tipo `StopIteration`.

La función `next` permite invocar manualmente al método `__next__`:

```
>>> next(i)
0
>>> next(i)
1
>>> next(i)
2
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Cuando hacemos `for x in range(3)`, el bucle `for` automáticamente llama a las funciones `iter` y `next`, asignando a x los valores que devuelve `next` en cada iteración.

Es decir que las siguientes estructuras son equivalentes:

```
for elemento in secuencia:
    # hacer algo con elemento
```

```
iterador = iter(secuencia)
while True:
    try:
        elemento = next(iterador)
    except StopIteration:
        break
    # hacer algo con elemento
```

Iterador para la lista enlazada

Si queremos implementar un iterador para la lista enlazada, la mejor solución implica crear una nueva clase, `_IteradorListaEnlazada`, que implemente el método `__next__` de la forma apropiada.

> **Atención**
>
> Utilizamos la notación de clase privada, utilizada también para la clase `_Nodo`, ya que si bien se devolverá el iterador cuando sea necesario, un programador externo no debería construir el iterador sin pasar a través de la lista enlazada.

Para inicializar la clase, lo único que se necesita es una referencia al primer elemento de la lista.

```
class _IteradorListaEnlazada:
    def __init__(self, prim):
        self.actual = prim
```

A partir de allí, el iterador irá avanzando a través de los elementos de la lista mediante el método `__next__`. Para verificar que no se haya llegado al final de la lista, se corroborará que la referencia `self.actual` sea distinta de `None`.

```
if self.actual is None:
    raise StopIteration()
```

Una vez que se pasó la verificación, la primera llamada a `__next__` debe devolver el primer elemento, pero también debe avanzar, para que la siguiente llamada devuelva el siguiente elemento. Por ello, se utiliza la estructura *guardar, avanzar, devolver*.

```
dato = self.actual.dato
self.actual = self.actual.prox
return dato
```

En el Código 16.4 se puede ver el código completo del iterador.

Código 16.4 `_IteradorListaEnlazada`**:** Un iterador para la lista enlazada

```
1  class _IteradorListaEnlazada:
2      def __init__(self, prim):
3          self.actual = prim
4
5      def __next__(self):
6          if self.actual is None:
7              raise StopIteration()
8
9          dato = self.actual.dato
10         self.actual = self.actual.prox
11         return dato
```

Finalmente, es necesario modificar la clase `ListaEnlazada` para que devuelva una instancia del iterador cuando se llama al método `__iter__`.

```
    def __iter__(self):
        """Devuelve un iterador de la lista."""
        return _IteradorListaEnlazada(self.prim)
```

Con todo esto será posible recorrer nuestra lista con la estructura a la que estamos acostumbrados.

```
>>> l = ListaEnlazada()
>>> l.append(7)
>>> l.append(3)
>>> l.append(5)
>>> for valor in l:
...     print(valor)
...
7
3
5
```

No solo eso, sino que además podremos utilizar la lista enlazada en cualquier operación que requiera una secuencia *iterable*:

```
>>> list(l) # convertir a una lista de Python
[7, 3, 5]
>>> sum(l) # sumar los elementos
15
>>> max(l) # obtener el máximo elemento
7
```

16.7 Resumen

- Un **tipo abstracto de datos** (TAD) es un tipo de datos que está definido por las operaciones que contiene y cómo se comportan (su *interfaz*), no por la forma en la que esas operaciones están implementadas.
- Una **lista enlazada** es una implementación del TAD *lista*. Se trata de una lista compuesta por nodos, en la que cada nodo contiene un dato y una referencia al nodo que le sigue.
- En las listas enlazadas, es *barato* insertar o eliminar elementos, ya que simplemente se deben alterar un par de referencias; pero es *caro* acceder a un elemento en particular, ya que es necesario pasar por todos los anteriores para llegar a él.
- Tanto al insertar como al remover elementos de una lista enlazada, se utiliza la técnica de *máquina de parejas*, mediante la cual se va recorriendo la lista hasta encontrar el lugar apropiado donde operar con las referencias.
- Una **lista doblemente enlazada** es aquella cuyos nodos además del dato contienen una referencia al nodo anterior y otra al nodo siguiente, de modo que se la puede recorrer en ambos sentidos.
- Una **lista circular** es aquella en la que el último nodo contiene una referencia al primero, y puede ser recorrida infinitamente.
- Un **iterador** es un objeto que permite recorrer uno a uno los elementos de una secuencia.

Referencia Python

`__iter__(self)`

Método especial que debe devolver un iterador para el objeto. El iterador debe ser un objeto que implementa el método `__next__`.

`__next__(self)`

Devuelve el elemento actual de la iteración y avanza al siguiente. Si se llegó al final de la iteración lanza `StopIteration`.

`iter(objeto)`

Equivalente a invocar `objeto.__iter__()`.

`next(objeto, [valor])`

Equivalente a invocar `objeto.__next__()`. Si se invoca con el parámetro adicional `valor`, al llegar al final de la iteración se devuelve el `valor` en lugar de lanzar `StopIteration`.

16.8 Ejercicios

Ejercicio 16.8.1. Implementar el método `__str__` de `ListaEnlazada`, para que se genere una salida legible de lo que contiene la lista, similar a las listas de python.

Ejercicio 16.8.2. Agregar a `ListaEnlazada` un método `extend` que reciba una `ListaEnlazada` y agregue a la lista actual los elementos que se encuentran en la lista recibida.

Ejercicio 16.8.3. Implementar el método `remover_todos(elemento)` de `ListaEnlazada`, que recibe un elemento y remueve de la lista todas las apariciones del mismo, devolviendo la cantidad de elementos removidos. La lista debe ser recorrida una sola vez.

Ejercicio 16.8.4. Implementar el método `duplicar(elemento)` de `ListaEnlazada`, que recibe un elemento y duplica todas las apariciones del mismo. Ejemplo:

```
L = 1 -> 5 -> 8 -> 8 -> 2 -> 8
L.duplicar(8) => L = 1 -> 5 -> 8 -> 8 -> 8 -> 8 -> 2 -> 8 -> 8
```

Ejercicio 16.8.5. Implementar el método `filter(f)` de `ListaEnlazada`, que recibe una función y devuelve **una nueva lista enlazada** con los elementos para los cuales la aplicación de `f` devuelve `True`. La nueva lista debe ser construida recorriendo los nodos una sola vez (es decir, no se puede llamar a `append`). Ejemplo:

```
L = 1 -> 5 -> 8 -> 8 -> 2 -> 8
L.filter(es_primo) -> L2 = 5 -> 2
```

Ejercicio 16.8.6. Escribir un método de la clase `ListaEnlazada` que invierta el orden de la lista (es decir, el primer elemento queda como último y viceversa).

Ejercicio 16.8.7. Una **lista circular** es una lista cuyo último nodo está ligado al primero, de modo que es posible recorrerla infinitamente. Escribir la clase `ListaCircular`, incluyendo los métodos `insert`, `append`, `remove` y `pop`.

Ejercicio 16.8.8. Una **lista doblemente enlazada** es una lista en la cual cada nodo tiene una referencia al anterior además de al próximo de modo que es posible recorrerla en ambas direcciones. Escribir la clase `ListaDobleEnlazada`, incluyendo los métodos `insert`, `append`, `remove` y `pop`.

Unidad 17

Pilas y colas

En esta unidad veremos dos ejemplos de tipos abstractos de datos, de los más clásicos: *pilas* y *colas*.

17.1 Pilas

Una *pila* es un TAD que tiene las siguientes operaciones:

- `__init__`: Inicializa una pila nueva, vacía.
- `apilar`: Agrega un nuevo elemento a la pila.
- `desapilar`: Elimina el tope de la pila y lo devuelve. El elemento que se devuelve es siempre el último que se agregó.
- `esta_vacia`: Devuelve `True` o `False` según si la pila está vacía o no.

El comportamiento de una pila se puede describir mediante la frase "Lo último que se apiló es lo primero que se usa", que es exactamente lo que uno hace con una pila (de platos por ejemplo): en una pila de platos uno sólo puede ver la apariencia completa del plato de arriba, y sólo puede tomar el plato de arriba (si se intenta tomar un plato del medio de la pila lo más probable es que alguno de sus vecinos, o él mismo, se arruine).

Como ya se dijo, al crear un tipo abstracto de datos, es importante decidir cuál será la representación a utilizar. En el caso de la pila, si bien puede haber más de una representación, por ahora veremos la más sencilla: representaremos una pila mediante una lista de Python.

Sin embargo, para los que construyen programas que usan un TAD vale el siguiente llamado de atención:

> **⚠ Atención**
>
> Al usar esa pila dentro de un programa, deberemos ignorar que se está trabajando sobre una lista: solamente podremos usar los métodos de la *interfaz pila*.

17.1.1 Pilas representadas por listas

Definiremos una clase `Pila` con un atributo `items`, de tipo lista, que contendrá los elementos de la pila. El tope de la pila se encontrará en la última posición de la lista, y cada vez que se apile un nuevo elemento, se lo agregará al final.

El método `__init__` no recibirá parámetros adicionales, ya que deberá crear una pila vacía (que representaremos por una lista vacía):

```
class Pila:
    """Representa una pila con operaciones de apilar, desapilar y
       verificar si está vacía."""

    def __init__(self):
        """Crea una pila vacía."""
        self.items = []
```

El método `esta_vacia` simplemente se fija si la lista de Python está vacía:

```
    def esta_vacia(self):
        """Devuelve True si la lista está vacía, False si no."""
        return len(self.items) == 0
```

El método `apilar` agrega el nuevo elemento al final de la lista:

```
    def apilar(self, x):
        """Apila el elemento x."""
        self.items.append(x)
```

Para implementar `desapilar` se usamos el método `pop` de lista que hace exactamente lo requerido: elimina el último elemento de la lista y devuelve el valor del elemento eliminado. Si la lista está vacía `desapilar` lanza una excepción.

```
    def desapilar(self):
        """Devuelve el elemento tope y lo elimina de la pila.
           Si la pila está vacía levanta una excepción."""
        if self.esta_vacia():
            raise IndexError("La pila está vacía")
        return self.items.pop()
```

Utilizamos los métodos `append` y `pop` de las listas de Python, porque sabemos que estos métodos se ejecutan en tiempo constante. Queremos que el tiempo de apilar o desapilar de la pila no dependa de la cantidad de elementos contenidos.

Construimos algunas pilas y operamos con ellas:

```
>>> p = Pila()
>>> p.esta_vacia()
True
>>> p.apilar(1)
>>> p.esta_vacia()
False
>>> p.apilar(5)
>>> p.apilar("+")
>>> p.apilar(22)
>>> p.desapilar()
22
>>> q = Pila()
>>> q.desapilar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Pila.py", line 24, in desapilar
    raise IndexError:("La pila está vacía")
IndexError: La pila está vacía
```

17.1.2 Uso de pila: calculadora científica

La famosa calculadora portátil HP-35 (de 1972) popularizó la notación polaca inversa (o notación prefijo) para hacer cálculos sin necesidad de usar paréntesis. Esa notación, inventada por el lógico polaco Jan Lukasiewicz en 1920, se basa en el principio de que un operador siempre se escribe a continuación de sus operandos. La operación $(5-3)+8$ se escribirá como `5 3 - 8 +`, que se interpretará como: "restar 3 de 5, y al resultado sumarle 8".

Es posible implementar esta notación de manera sencilla usando una pila de la siguiente manera, a partir de una cadena de entrada de valores separados por blancos:

- Mientras se lean números, se apilan.
- En el momento en el que se detecta una operación binaria `+`, `-`, `*` o `/` se desapilan los dos últimos números apilados, se ejecuta la operación indicada, y el resultado de esa operación se apila.
- Si la expresión está bien formada, tiene que quedar al final un único número en la pila (el resultado).
- Los posibles errores son:
  - Queda más de un número al final (por ejemplo si la cadena de entrada fue `"5 3"`),
  - Ingresa algún caracter que no se puede interpretar ni como número ni como una de las cinco operaciones válidas (por ejemplo si la cadena de entrada fue `"5 3 &"`)
  - No hay suficientes operandos para realizar la operación (por ejemplo si la cadena de entrada fue `"5 3 - +"`).

La siguiente es la estrategia de resolución:

Dada una cadena con la expresión a evaluar, podemos separar sus componentes utilizando el método `split`. Recorreremos luego la lista de componentes realizando las acciones indicadas en el párrafo anterior, utilizando una pila auxiliar para operar. Si la expresión está bien formada devolveremos el resultado, de lo contrario levantaremos una excepción.

En el Código 17.1 está la implementación de la calculadora descripta.

Veamos algunos casos de prueba:

- El caso de una expresión que es sólo un número (es correcta):

```
>>> calculadora_polaca.main()
Ingrese la expresion a evaluar: 5
DEBUG: 5
DEBUG: apila  5.0
5.0
```

- El caso en el que sobran operandos:

```
>>> calculadora_polaca.main()
Ingrese la expresion a evaluar: 4 5
DEBUG: 4
DEBUG: apila  4.0
DEBUG: 5
DEBUG: apila  5.0
Traceback (most recent call last):
```

Código 17.1 `calculadora_polaca.py`: Una calculadora polaca inversa

```
from Pila import Pila

def calculadora_polaca(elementos):
    """Dada una lista de elementos que representan las componentes de
       una expresión en notacion polaca inversa, evalúa dicha expresión.
       Si la expresion está mal formada, levanta ValueError."""

    p = Pila()
    for elemento in elementos:
        print("DEBUG:", elemento)
        # Intenta convertirlo a número
        try:
            numero = float(elemento)
            p.apilar(numero)
            print("DEBUG: apila ", numero)
        # Si no se puede convertir a número, debería ser un operando
        except ValueError:
            # Si no es un operando válido, levanta ValueError
            if elemento not in ('+', '-', '*', '/'):
                raise ValueError("Operando inválido")
            # Si es un operando válido, intenta desapilar y operar
            try:
                a1 = p.desapilar()
                print("DEBUG: desapila ", a1)
                a2 = p.desapilar()
                print("DEBUG: desapila ", a2)
            # Si hubo problemas al desapilar
            except IndexError:
                raise ValueError("Faltan operandos")

            if elemento == "+":
                resultado = a2 + a1
            elif elemento == "-":
                resultado = a2 - a1
            elif elemento == "*":
                resultado = a2 * a1
            elif elemento == "/":
                resultado = a2 / a1
            print("DEBUG: apila ", resultado)
            p.apilar(resultado)
    # Al final, el resultado debe ser lo único en la Pila
    resultado = p.desapilar()
    if not p.esta_vacia():
        raise ValueError("Sobran operandos")
    return resultado

def main():
    expresion = input("Ingrese la expresion a evaluar: ")
    elementos = expresion.split()
    print(calculadora_polaca(elementos))

main()
```

```
  File "<stdin>", line 1, in <module>
  File "calculadora_polaca.py", line 50, in main
    print(calculadora_polaca(elementos))
  File "calculadora_polaca.py", line 44, in calculadora_polaca
    raise ValueError("Sobran operandos")
ValueError: Sobran operandos
```

- El caso en el que faltan operandos:

```
>>> calculadora_polaca.main()
Ingrese la expresion a evaluar: 4 /
DEBUG: 4
DEBUG: apila  4.0
DEBUG: /
DEBUG: desapila  4.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "calculadora_polaca.py", line 50, in main
    print(calculadora_polaca(elementos))
  File "calculadora_polaca.py", line 29, in calculadora_polaca
    raise ValueError("Faltan operandos")
ValueError: Faltan operandos
```

- El caso de un operador inválido:

```
>>> calculadora_polaca.main()
Ingrese la expresion a evaluar: 4 5 &
DEBUG: 4
DEBUG: apila  4.0
DEBUG: 5
DEBUG: apila  5.0
DEBUG: &
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "calculadora_polaca.py", line 50, in main
    print(calculadora_polaca(elementos))
  File "calculadora_polaca.py", line 20, in calculadora_polaca
    raise ValueError("Operando inválido")
ValueError: Operando inválido
```

- `4 + 5`

```
>>> calculadora_polaca.main()
Ingrese la expresion a evaluar: 4 5 +
DEBUG: 4
DEBUG: apila  4.0
DEBUG: 5
DEBUG: apila  5.0
DEBUG: +
DEBUG: desapila  5.0
DEBUG: desapila  4.0
DEBUG: apila  9.0
9.0
```

- `(4 + 5) * 6`:

```
>>> calculadora_polaca.main()
Ingrese la expresion a evaluar: 4 5 + 6 *
DEBUG: 4
DEBUG: apila  4.0
DEBUG: 5
DEBUG: apila  5.0
DEBUG: +
DEBUG: desapila  5.0
DEBUG: desapila  4.0
DEBUG: apila  9.0
DEBUG: 6
DEBUG: apila  6.0
DEBUG: *
DEBUG: desapila  6.0
DEBUG: desapila  9.0
DEBUG: apila  54.0
54.0
```

Ejercicio 17.1. Si se oprime la tecla ⟵Backspace del teclado, se borra el último caracter ingresado. Construir una función `visualizar` para modelar el tipeo de una cadena de caracteres desde un teclado:

La función recibe una cadena de caracteres con todo lo que el usuario ingresó por teclado (incluyendo ⟵Backspace, que se reconoce como `\b`), y devuelve el texto tal como debe presentarse (por ejemplo, `visualizar("Holas\b chau")` debe devolver 'Hola chau').

Atención, que muchas veces la gente aprieta de más la tecla ⟵Backspace, y no por eso la función tiene que lanzar una excepción.

17.1.3 ¿Cuánto cuestan los métodos?

Al elegir de una representación debemos tener en cuenta cuánto nos costarán los métodos implementados. En nuestro caso, el tope de la pila se encuentra en la última posición de la lista, y cada vez que se apila un nuevo elemento, se lo agregará al final.

Por lo tanto se puede implementar el método `apilar` mediante un `append` de la lista, *que se ejecuta en tiempo constante*. También el método `desapilar`, que se implementa mediante `pop` de lista, *se ejecuta en tiempo constante*.

Vemos que la alternativa que elegimos fue barata.

Otra alternativa posible habría sido agregar el nuevo elemento en la posición 0 de la lista, es decir implementar el método `apilar` mediante `self.items.insert(0, x)` y el método `desapilar` mediante `self.items.pop(0)`. Sin embargo, ésta no es una solución inteligente, ya que tanto insertar al comienzo de la lista como borrar al comienzo de la lista de Python *consumen tiempo proporcional a la longitud de la lista*.

Ejercicio 17.2. Diseñar un pequeño experimento para verificar que la implementación elegida es mucho mejor que la implementación con listas en la cual el elemento nuevo se inserta al principio de la lista.

Ejercicio 17.3. Implementar pilas mediante listas enlazadas. Analizar el costo de los métodos a utilizar.

17.2 Colas

El TAD *cola* modela el comportamiento: "el primero que llega es el primero en ser atendido"; los demás elementos se van *encolando* hasta que les toque su turno.

Sus operaciones son:

- `__init__`: Inicializa una cola nueva, vacía.
- `encolar`: Agrega un nuevo elemento al final de la cola.
- `desencolar`: Elimina el primero de la cola y lo devuelve.
- `esta_vacia`: Devuelve `True` o `False` según si la cola está vacía o no.

17.2.1 Colas implementadas sobre listas

Al momento de realizar una implementación de una Cola, deberemos preguntarnos ¿Cómo representamos a las colas? Veamos, en primer lugar, si podemos implementar colas usando listas de Python, como hicimos con la Pila.

Definiremos una clase `Cola` con un atributo, `items`, de tipo lista, que contendrá los elementos de la cola. El primero de la cola se encontrará en la primera posición de la lista, y cada vez que encole un nuevo elemento, se lo agregará al final.

El método `__init__` no recibirá parámetros adicionales, ya que deberá crear una cola vacía (que representaremos por una lista vacía):

```
class Cola:
    """Representa a una cola, con operaciones de encolar y
       desencolar. El primero en ser encolado es también el primero
       en ser desencolado."""

    def __init__(self):
        """Crea una cola vacía."""
        self.items = []
```

El método `encolar` se implementará agregando el nuevo elemento al final de la lista:

```
    def encolar(self, x):
        """Encola el elemento x."""
        self.items.append(x)
```

Para implementar `desencolar`, se eliminará el primer elemento de la lista y se devolverá el valor del elemento eliminado, utilizaremos nuevamente el método `pop`, pero en este caso le pasaremos la posición 0, para que elimine el primer elemento, no el último. Si la cola está vacía se levantará una excepción.

```
    def desencolar(self):
        """Elimina el primer elemento de la cola y devuelve su
           valor. Si la cola está vacía, levanta ValueError."""
        if self.esta_vacia():
            raise ValueError("La cola está vacía")
        return self.items.pop(0)
```

Por último, el método `esta_vacia`, que indicará si la cola está o no vacía.

```
    def esta_vacia(self):
        """Devuelve True si la cola esta vacía, False si no."""
        return len(self.items) == 0
```

Veamos una ejecución de este código:

```
>>> from claseCola import Cola
>>> q = Cola()
>>> q.esta_vacia()
True
>>> q.encolar(1)
>>> q.encolar(2)
>>> q.encolar(5)
>>> q.esta_vacia()
False
>>> q.desencolar()
1
>>> q.desencolar()
2
>>> q.encolar(8)
>>> q.desencolar()
5
>>> q.desencolar()
8
>>> q.esta_vacia()
True
>>> q.desencolar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Cola.py", line 24, in desencolar
    raise ValueError("La cola está vacía")
ValueError: La cola está vacía
```

¿Cuánto cuesta esta implementación? Dijimos en la sección anterior que usar listas comunes para borrar elementos al principio da muy malos resultados. Como en este caso necesitamos agregar elementos por un extremo y quitar por el otro extremo, esta implementación será una buena alternativa sólo si nuestras listas son pequeñas, ya que e medida que la cola crece, el método `desencolar` tardará cada vez más.

Pero si queremos hacer que tanto el `encolar` como el `desencolar` se ejecuten en tiempo constante, debemos apelar a otra implementación.

17.2.2 Colas y listas enlazadas

En la unidad anterior vimos la clase `ListaEnlazada`. La clase presentada ejecutaba la inserción en la primera posición en tiempo constante, pero el `append` se había convertido en lineal.

Sin embargo, como ejercicio, se propuso mejorar el `append`, agregando un nuevo atributo que apunte al último nodo, de modo de poder agregar elementos en tiempo constante.

Si esas mejoras estuvieran hechas, cambiar nuestra clase `Cola` para que utilice la `ListaEnlazada` sería tan simple como cambiar el constructor, para que en lugar de construir una lista de Python construyera una lista enlazada.

```
class Cola:
    def __init__(self):
        self.items = ListaEnlazadaMejorada()
```

Sin embargo, una `Cola` es bastante más sencilla que una lista enlazada con referencia al último, por lo que también podemos implementar una clase `Cola` utilizando las técnicas de referencias, que se vieron en las *listas enlazadas*.

Planteamos otra solución posible para obtener una cola que sea eficiente tanto al encolar como al desencolar, utilizando los nodos de las listas enlazadas, y solamente implementaremos insertar al final y remover al principio.

Para ello, la cola deberá tener dos atributos, `self.primero` y `self.ultimo`, que en todo momento deberán apuntar al primer y último nodo de la cola, es decir que serán los invariantes de esta cola.

En primer lugar los crearemos vacíos, ambos referenciando a `None`.

```
class Cola:
    def __init__(self):
        """Crea una cola vacía."""
        self.primero = None
        self.ultimo = None
```

Al momento de encolar, hay dos situaciones a tener en cuenta:

- Si la cola está vacía (es decir, `self.ultimo` es `None`), tanto `self.primero` como `self.ultimo` deben pasar a referenciar al nuevo nodo, ya que este nodo será a la vez el primero y el último.
- Si ya había nodos en la cola, simplemente hay que agregar el nuevo a continuación del último y actualizar la referencia de `self.ultimo`.

El código resultante es el siguiente.

```
    def encolar(self, x):
        """Encola el elemento x."""
        nuevo = Nodo(x)
        if self.ultimo is not None:
            self.ultimo.prox = nuevo
            self.ultimo = nuevo
        else:
            self.primero = nuevo
            self.ultimo = nuevo
```

Al momento de desencolar, será necesario verificar que la cola no esté vacía, y de ser así levantar una excepción. Si la cola no está vacía, se almacena el valor del primer nodo de la cola y luego se avanza la referencia `self.primero` al siguiente elemento.

Nuevamente hay un caso particular a tener en cuenta y es el que sucede cuando luego de eliminar el primer nodo de la cola, la cola queda vacía. En este caso, además de actualizar la referencia de `self.primero`, también hay que actualizar la referencia de `self.ultimo`.

```
    def desencolar(self):
        """Desencola el primer elemento y devuelve su valor.
           Si la cola está vacía, levanta ValueError."""
        if self.primero is None:
            raise ValueError("La cola está vacía")
        valor = self.primero.dato
        self.primero = self.primero.prox
        if not self.primero:
            self.ultimo = None
        return valor
```

Finalmente, para saber si la cola está vacía, es posible verificar tanto si `self.primero` o `self.ultimo` referencian a `None`.

```
    def esta_vacia(self):
        """Devuelve True si la cola esta vacía, False si no."""
        return self.primero is None
```

Una vez implementada toda la interfaz de la cola, podemos probar el TAD resultante

```
>>> q = Cola()
>>> q.esta_vacia()
True
>>> q.encolar("Manzanas")
>>> q.encolar("Peras")
>>> q.encolar("Bananas")
>>> q.esta_vacia()
False
>>> q.desencolar()
'Manzanas'
>>> q.desencolar()
'Peras'
>>> q.encolar("Guaraná")
>>> q.desencolar()
'Bananas'
>>> q.desencolar()
'Guaraná'
>>> q.desencolar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ColaEnlazada.py", line 42, in desencolar
    raise ValueError("La cola está vacía")
ValueError: La cola está vacía
```

Ejercicio 17.4. Hace un montón de años había una viejísma sucursal del correo en la vereda impar de Av. de Mayo al 800 que tenía un cartel que decía "No se recibirán más de 5 cartas por persona". O sea que la gente entregaba sus cartas (hasta la cantidad permitida) y luego tenía que volver a hacer la cola si tenía más cartas para despachar.

Modelar una cola de correo generalizada, donde en la inicialización se indica la cantidad (no necesariamente 5) de cartas que se reciben por persona.

17.3 Resumen

- Una **pila** es un tipo abstracto de datos que permite agregar elementos y sacarlos en el orden inverso al que se los colocó, de la misma forma que una pila (de platos, libros, cartas, etc) en la vida real.
- Las pilas son útiles en las situaciones en las que se desea operar primero con los últimos elementos agregados, como es el caso de la notación polaca inversa.
- Una **cola** es un tipo abstracto de datos que permite agregar elementos y sacarlos en el mismo orden en que se los colocó, como una cola de atención en la vida real.
- Las colas son útiles en las situaciones en las que se desea operar con los elementos en el orden en el que se los fue agregando, como es el caso de un cola de atención de clientes.

17.4 Ejercicios

Ejercicio 17.4.1. Escribir una clase *TorreDeControl* que modele el trabajo de una torre de control de un aeropuerto con una pista de aterrizaje. Los aviones que están esperando para aterrizar tienen prioridad sobre los que están esperando para despegar. La clase debe funcionar conforme al siguiente ejemplo:

```
>>> torre = TorreDeControl()
>>> torre.nuevo_arribo('AR156')
>>> torre.nueva_partida('KLM1267')
>>> torre.nuevo_arribo('AR32')
>>> torre.ver_estado()
Vuelos esperando para aterrizar: AR156, AR32
Vuelos esperando para despegar: KLM1267
>>> torre.asignar_pista()
El vuelo AR156 aterrizó con éxito.
>>> torre.asignar_pista()
El vuelo AR32 aterrizó con éxito.
>>> torre.asignar_pista()
El vuelo KLM1267 despegó con éxito.
>>> torre.asignar_pista()
No hay vuelos en espera.
```

Ejercicio 17.4.2. Escribir las clases `Impresora` y `Oficina` que permita modelar el funcionamiento de un conjunto de impresoras conectadas en red.
Una impresora:

- Tiene un nombre, y una capacidad máxima de tinta.
- Permite encolar un documento para imprimir (recibiendo el nombre del documento).
- Permite imprimir el documento que está al frente de la cola.
  - Si no hay documentos encolados, se muestra un mensaje informándolo.
  - Si no hay tinta suficiente, se muestra un mensaje informándolo.
  - En caso contrario, se muestra el nombre del documento, y se gasta 1 unidad de tinta.
- Permite cargar el cartucho de tinta

Una oficina:

- Permite agregar una impresora
- Permite obtener una impresora por nombre
- Permite quitar una impresora por nombre
- Permite obtener la impresora que tenga menos documentos encolados.

Ejemplo:

```
>>> o = Oficina()
>>> o.agregar_impresora(Impresora('HP1234', 1))
>>> o.agregar_impresora(Impresora('Epson666', 5))
>>> o.impresora('HP1234').encolar('tp1.pdf')
>>> o.impresora('Epson666').encolar('tp2.pdf')
>>> o.impresora('HP1234').encolar('tp3.pdf')
>>> o.obtener_impresora_libre().encolar('tp4.pdf') # se encola en Epson666
>>> o.impresora('HP1234').imprimir()
tp1.pdf impreso
>>> o.impresora('HP1234').imprimir()
No tengo tinta :(
>>> o.impresora('HP1234').cargar_tinta()
>>> o.impresora('HP1234').imprimir()
```

```
tp3.pdf impreso
```

Ejercicio 17.4.3. En la parada del colectivo 130 pueden ocurrir dos eventos diferentes:

- Llega una persona
- Llega un colectivo con n asientos libres, y se suben al mismo todas las personas que están esperando, en orden de llegada, hasta que no quedan asientos libres.

Cada evento se representa con una tupla que incluye:

- El instante de tiempo (cantidad de segundos desde el inicio del día)
- El tipo de evento, que puede ser `'p'` (persona) o `'c'` (colectivo).
- En el caso de un evento de tipo `'c'` hay un tercer elemento que es la cantidad de asientos libres.

Escribir una función que recibe una lista de eventos, ordenados cronológicamente, y devuelva el promedio de tiempo de espera de los pasajeros en la parada.

Ejemplo:

```
promedio_espera([(35,'p'), (43,'p'), (80,'c',1), (98,'p'), (142,'c',2)])
    -> 62.6667 (calculado como (45+99+44) / 3)
```

Ejercicio 17.4.4. Juego de Cartas

a) Crear una clase `Carta` que contenga un palo y un valor.

b) Crear una clase `Solitario` que permita apilar las cartas una arriba de otra, pero sólo permita apilar una carta si es de un número inmediatamente inferior y de distinto palo a la carta que está en el tope. Si se intenta apilar una carta incorrecta, debe lanzar una excepción.

Ejercicio 17.4.5. Crear una clase `PilaConMaximo` que soporte las operaciones de `Pila` (`apilar(elemento)` y `desapilar()`), y además incluya el método `obtener_maximo()` **en tiempo constante**. Ayuda: usar dos pilas, una para guardar los elementos y otra para guardar los máximos.

Ejercicio 17.4.6. Escribir una función que recibe una expresión matemática (en forma de cadena) y devuelve `True` si los paréntesis (`'()'`), corchetes (`'[]'`) y llaves (`'{}'`) están correctamente balanceados, `False` en caso contrario. Ejemplos:

```
validar('(x+y)/2') -> True
validar('[8*4(x+y)]+{2/5}') -> True
validar('(x+y]/2') -> False
validar('1+)2(+3') -> False
```

Apéndice 17.A Implementación de la pila y cola

A continuación el código completo de la pila y las colas implementadas en esta unidad.

Código 17.2 `Pila.py`**:** Implementación básica de una pila

```
class Pila:
    """Representa una pila con operaciones de apilar, desapilar y
       verificar si está vacía."""

    def __init__(self):
        """Crea una pila vacía."""
        self.items = []

    def apilar(self, x):
        """Apila el elemento x."""
        self.items.append(x)

    def desapilar(self):
        """Desapila el elemento x y lo devuelve.
           Si la pila está vacía levanta una excepción."""
        if self.esta_vacia():
            raise ValueError("La pila está vacía")
        return self.items.pop()

    def esta_vacia(self):
        """Devuelve True si la lista está vacía, False si no."""
        return len(self.items) == 0
```

Código 17.3 `Cola.py`**:** Implementación básica de una cola

```
class Cola:
    """Representa a una cola, con operaciones de encolar y
       desencolar. El primero en ser encolado es también el primero
       en ser desencolado."""

    def __init__(self):
        """Crea una cola vacía."""
        self.items = []

    def encolar(self, x):
        """Agrega el elemento x como último de la cola."""
        self.items.append(x)

    def desencolar(self):
        """Desencola el primer elemento y devuelve su
           valor. Si la cola está vacía, levanta ValueError."""
        if self.esta_vacia():
            raise ValueError("La cola está vacía")
        return self.items.pop(0)

    def esta_vacia(self):
        """Devuelve True si la cola esta vacía, False si no."""
        return len(self.items) == 0
```

Código 17.4 `ColaEnlazada.py`: Implementación de una cola enlazada

```
from Nodo import Nodo

class Cola:
    """Representa a una cola, con operaciones de encolar y
       desencolar.  El primero en ser encolado es también el primero
       en ser desencolado. """

    def __init__(self):
        """Crea una cola vacía."""
        self.primero = None
        self.ultimo = None

    def encolar(self, x):
        """Encola el elemento x."""
        nuevo = Nodo(x)
        if self.ultimo:
            self.ultimo.prox = nuevo
            self.ultimo = nuevo
        else:
            self.primero = nuevo
            self.ultimo = nuevo

    def desencolar(self):
        """Desencola el primer elemento y devuelve su
           valor. Si la cola está vacía, levanta ValueError."""
        if self.primero is None:
            raise ValueError("La cola está vacía")
        valor = self.primero.dato
        self.primero = self.primero.prox
        if not self.primero:
            self.ultimo = None
        return valor

    def esta_vacia(self):
        """Devuelve True si la cola esta vacía, False si no."""
        return self.primero is None
```

Unidad 18

Modelo de ejecución de funciones y recursión

18.1 La pila de ejecución de las funciones

Si miramos el siguiente segmento de código y su ejecución podemos comprobar que, pese a tener el mismo nombre, la variable de `x` de la función `f` y la variable de `x` de la función `g` no tienen nada que ver: una y otra se refieren a valores distintos, y modificar una no modifica a la otra.

```
def f():
    x = 50
    a = 20
    print("En f, x vale", x)

def g():
    x = 10
    b = 45
    print("En g, antes de llamar a f, x vale", x)
    f()
    print("En g, después de llamar a f, x vale", x)
```

Esta es la ejecución de `g()`:

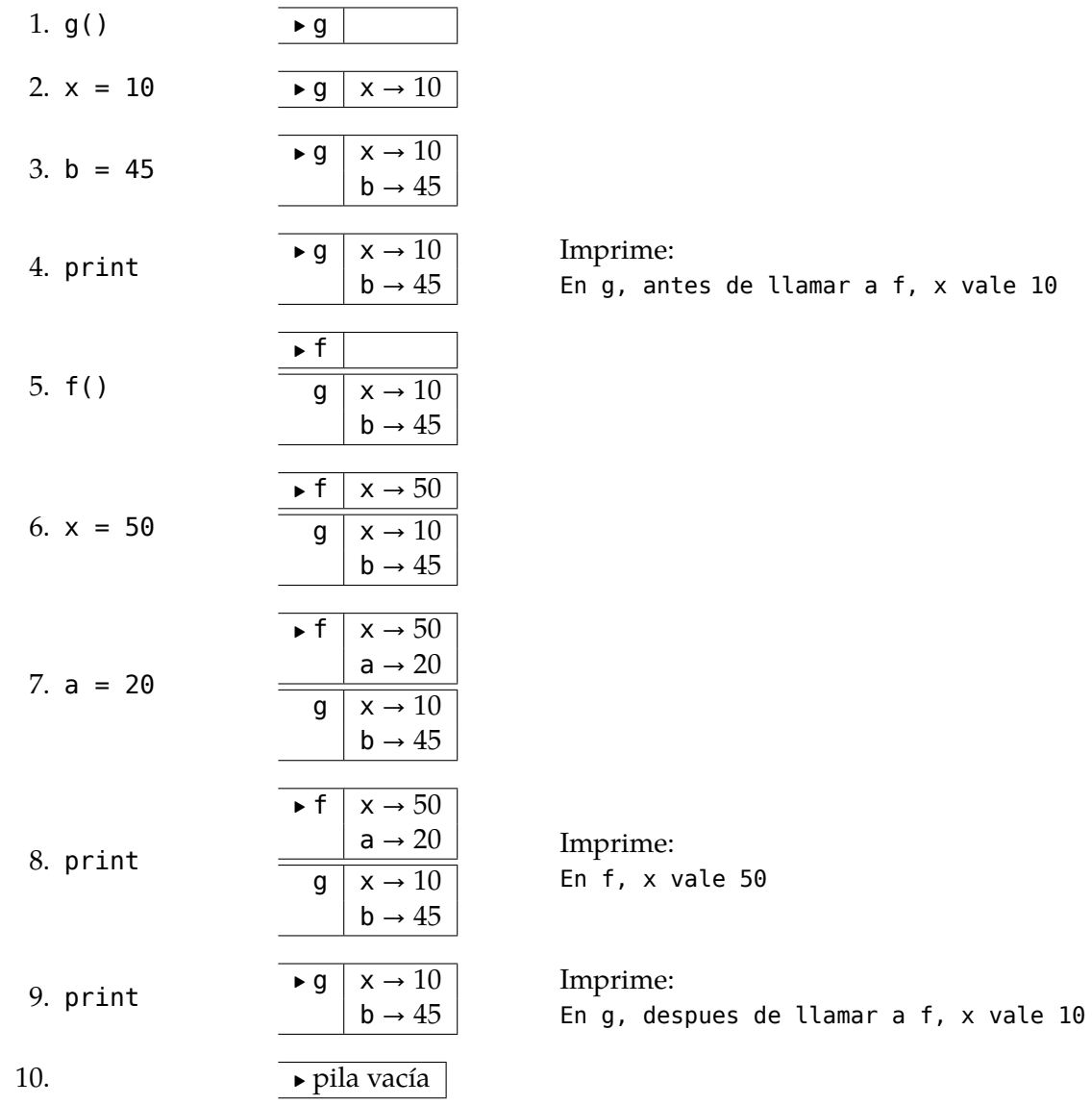
```
>>> g()
En g, antes de llamar a f, x vale 10
En f, x vale 50
En g, después de llamar a f, x vale 10
```

Este comportamiento lo dimos por sentado desde el principio, pero nunca nos detuvimos a pensar por qué sucede. Vamos a ver en esta sección cómo se ejecutan las llamadas a funciones, para comprender cuál es la razón de este comportamiento.

Cada función tiene asociado por un lado un código (el texto del programa) que se ejecutará, y por el otro un conjunto de variables que le son propias (en este caso `x` y `a` se asocian con `f` y `x` y `b` se asocian con `g`) y que no se confunden entre sí pese a tener el mismo nombre. No debería llamarnos la atención, ya que después de todo conocemos a muchas personas que tienen el mismo nombre.

Estos nombres asociados a una función los va *descubriendo* el intérprete de Python a medida que va ejecutando el programa (hay otros lenguajes en los que los nombres se descubren todos juntos antes de iniciar la ejecución).

La ejecución del programa se puede modelar por el siguiente diagrama, en el cual los nombres asociados a cada función se encerrarán en una caja o *marco*:



Se puede observar que:

- Cuando se invoca a `g`, se arma un *marco* vacío para contener las referencias a las variables asociadas con `g`. Ese marco se apila sobre una *pila vacía*. El marco que está en el tope de la pila es el *marco actual* y se marca con ▸ en el diagrama.
- Cuando se ejecuta dentro de `g` la invocación `f()` (en el paso 5) se *apila* un marco vacío que va a alojar las variables asociadas con `f`, que pasa a ser el marco actual, y se transfiere el control del programa a la primera instrucción de `f`. El marco de `g` queda debajo del tope de la pila, y por lo tanto el intérprete no lo ve.
- Mientras se ejecuta `f`, el intérprete busca los valores que necesita usando el marco que está en el tope de la pila. Si alguna línea de código de `f` intentara acceder a una variable llamada `b`, el intérprete no la encontraría y lanzaría una excepción.

- Después de ejecutar 8, se encuentra el final de la ejecución de `f`. Se desapila el marco de `f` y reaparece el marco de `g` en el tope de la pila. Sigue ejecutando `g` a partir de donde se suspendió por la invocación a `f`. `g` sólo ve su marco en el tope de la pila.
- Después de ejecutar 9, se encuentra el final de la ejecución de `g`. Se desapila el marco de `g` y queda la pila vacía.

El **ámbito de definición** de una variable está constituido por todas las partes del programa desde donde esa variable *es visible*.

18.2 Pasaje de parámetros

Un parámetro es una variable más dentro del marco de una función. Sólo hay que tener en cuenta que si en la invocación se le pasa un valor a ese parámetro, en el marco inicial esa variable ya aparecerá ligada a un valor. Analicemos el siguiente código de ejemplo:

```
def fun1(a):
    print(a + 1)

def fun2(b):
    fun1(b + 5)
    print("Volvio a fun2")
```

Con la siguiente ejecución:

```
>>> fun2(43)
49
Volvio a fun2
```

En este caso, la ejecución se puede representar de la siguiente manera:

1. `fun2(43)`

▸ fun2	b → 43

2. `fun1(b+5)`

▸ fun1	a → 48
fun2	b → 43

3. `print(a+1)`

▸ fun1	a → 48
fun2	b → 43

Imprime:
`49`

4. `print`

▸ fun2	b → 43

Imprime:
`Volvio a fun2`

5.

▸ pila vacía

Cuando se pasan objetos como parámetros, las dos variables hacen referencia al *mismo* objeto. Eso significa que si el objeto pasado es mutable, cualquier modificación que la función invocada realice sobre su parámetro se reflejará en el argumento de la función llamadora, como se puede ver en el siguiente ejemplo:

```
def modif(lista):
    lista[0] = 5

def main():
    ls = [1, 2, 3, 4]
```

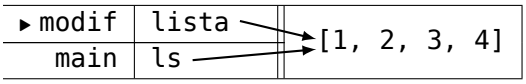
```
    print(ls)
    modif(ls)
    print(ls)
```

Y esta es la ejecución:

```
>>> main()
[1, 2, 3, 4]
[5, 2, 3, 4]
```

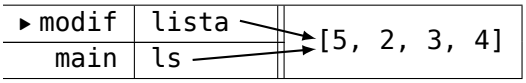
- Cuando se invoca a `modif(ls)` desde `main`, el esquema de la pila es el siguiente:

`modif(ls)`



- Cuando se modifica la lista desde `modif`, el esquema de la pila es el siguiente:

`lista[0] = 5`



- Cuando la ejecución vuelve a `main`, `ls` seguirá apuntando a la lista `[5, 2, 3, 4]`.

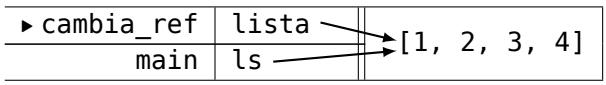
En cambio, cuando el parámetro cambia la referencia que se le pasó por una referencia a otro objeto, la función `main` no se entera:

```
def cambia_ref(lista):
    lista = [5, 2, 3, 4]

def main():
    ls = [1, 2, 3, 4]
    print(ls)
    cambia_ref(ls)
    print(ls)
```

```
>>> main()
[1, 2, 3, 4]
[1, 2, 3, 4]
```

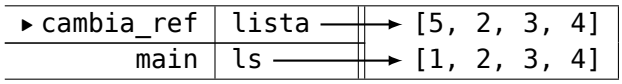
- Cuando se invoca a `cambia_ref(ls)` desde `main`, el esquema de la pila es el siguiente:

`cambia_ref()`



- Cuando se cambia referencia a la lista desde `cambia_ref`, **se crea una nueva instancia de lista**, y el esquema de la pila es el siguiente:

`lista = [...]`



- Cuando la ejecución vuelve a `main`, `ls` seguirá apuntando a la lista `[1, 2, 3, 4]`.

18.3 Devolución de resultados

Finalmente, para completar los distintos seguimientos, debemos tener en cuenta que los resultados que devuelve la función invocada, se *reciben* en la expresión correspondiente de la función invocante.

```
def cuad(valor):
    c = valor * valor
    return c

def main():
    c = cuad(6)
    print(c)
```

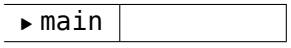
En este caso, si hacemos el seguimiento de la funcion invocada:

```
>>> main()
36
```

Veremos algo como lo siguiente:

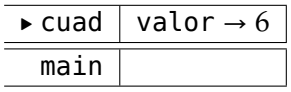
1. `main()`

2. `c = cuad(6)`

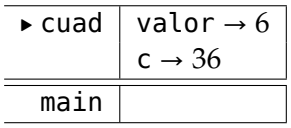


Se suspende la ejecución.
Se invoca a `cuad(6)`.

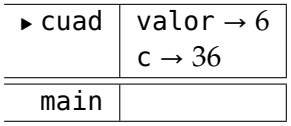
3. `cuad(6)`



4. `c = valor * valor`

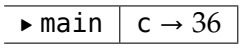


5. `return c`

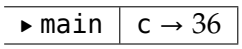


`cuad` devuelve 36.

6. `c = cuad(6)`

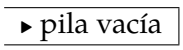


7. `print(c)`



Imprime:
`36`.

8.



Según se ve en el paso 6, al momento de devolver un valor, el valor de retorno correspondiente a la función `cuad` es el que se asigna a la variable `cuad`, a la vez que la llamada a la función se elimina de la pila.

18.4 La recursión y cómo puede ser que funcione

Estamos acostumbrados a escribir funciones que llaman a otras funciones. Pero lo cierto es que nada impide que en Python (y en muchos otros lenguajes) una función se llame a sí misma. Y lo más interesante es que esta propiedad, que se llama *recursión*, permite en muchos casos encontrar soluciones muy elegantes para determinados problemas.

En materias de matemática se estudian los razonamientos por inducción para probar propiedades de números enteros, la recursión no es más que una generalización de la inducción a más estructuras: las listas, las cadenas de caracteres, las funciones, etc.

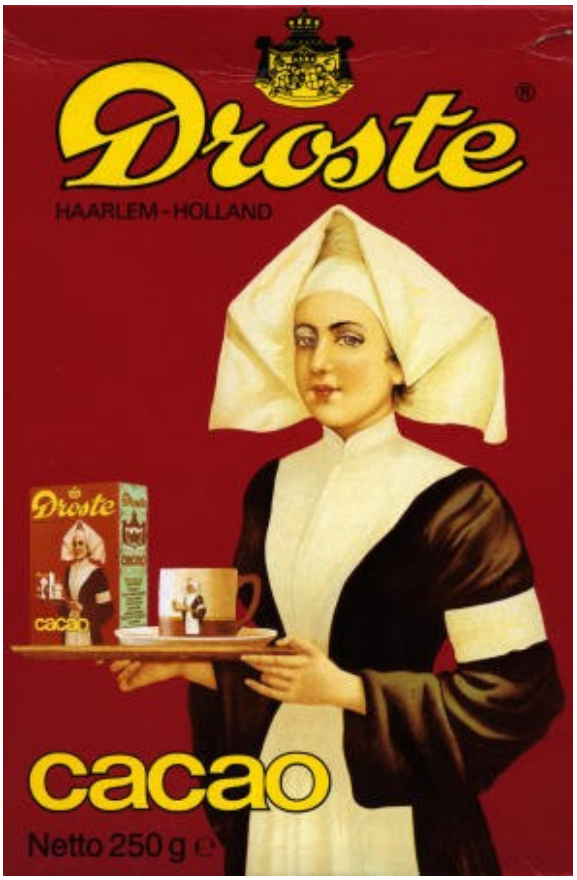


Figura 18.1: Una imagen recursiva: la publicidad de Cacao Droste.

A continuación estudiaremos diversas situaciones en las cuales aparece la recursión, veremos cómo es que esto puede funcionar, algunas situaciones en las que es conveniente utilizarla y otras situaciones en las que no.

18.5 Una función recursiva matemática

Es muy común tener definiciones inductivas de operaciones, como por ejemplo:

$$0! = 1$$
$$x! = x\,(x-1)! \quad \text{si } x > 0$$

Este tipo de definición se traduce naturalmente en una función en Python:

```
def factorial(n):
    """Precondición: n entero >= 0
       Devuelve: n!"""
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Esta es la ejecución del factorial para `n = 0` y para `n = 3`.

```
>>> factorial(0)
1
>>> factorial(3)
6
```

El sentido de la instrucción `n * factorial(n - 1)` es exactamente el mismo que el de la definición inductiva: para calcular el factorial de n se debe multiplicar n por el factorial de $n-1$.

Dos piezas fundamentales para garantizar el funcionamiento de este programa son:

- Que se defina un *caso base* (en este caso la indicación *no recursiva* de cómo calcular `factorial(0)`).
- Que el argumento de la función respete la precondición de que `n` debe ser un entero mayor o igual que 0.

Dado que ya vimos la pila de evaluación y cómo funciona, no debería llamarnos la atención que esto pueda funcionar adecuadamente en un lenguaje de programación que utilice pila para evaluar.

Para poder analizar qué sucede a cada paso de la ejecución de la función, utilizaremos una versión más detallada del mismo código, en la que el resultado de cada paso se asigna a una variable.

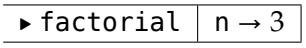
```
def factorial(n):
    if n == 0:
        r = 1
        return r

    f = factorial(n-1)
    r = n * f
    return r
```

Esta porción de código funciona exactamente igual que la anterior, pero nos permite ponerles nombres a los resultados intermedios de cada operación para poder estudiar qué sucede a cada paso. Analicemos entonces la ejecución de `factorial(3)` mediante la pila de evaluación:

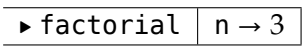
1. `factorial(3)`

▸ `factorial`	`n` → 3

2. `if n == 0:`

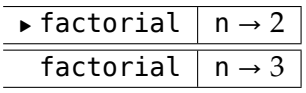


3. `f = factorial(n - 1)`

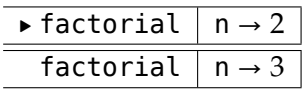


Se suspende el cálculo.
Se invoca a `factorial(2)`.

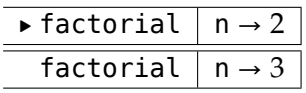
4. `factorial(2)`



5. `if n == 0:`

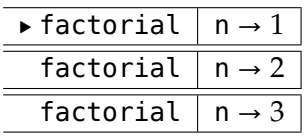


6. `f = factorial(n - 1)`

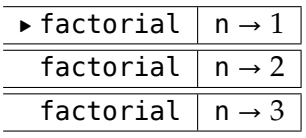


Se suspende el cálculo.
Se invoca a `factorial(1)`.

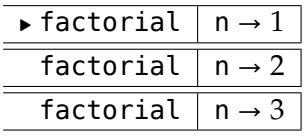
7. `factorial(1)`



8. `if n == 0:`

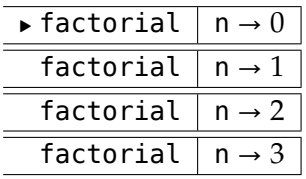


9. `f = factorial(n - 1)`

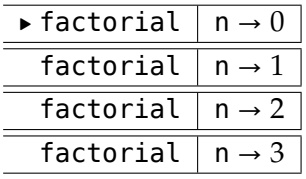


Se suspende el cálculo.
Se llama a `factorial(0)`.

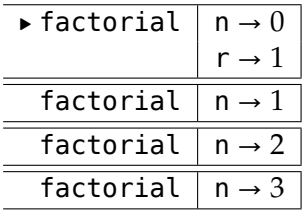
10. `factorial(0)`



11. `if n == 0:`

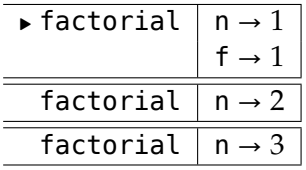


12.
```
r = 1
return r
```



`factorial(0)` devuelve 1

13. `f = factorial(n - 1)`



14.
```
r = n * f
return r
```

▸ factorial	n → 1 f → 1 r → 1
factorial	n → 2
factorial	n → 3

`factorial(1)` devuelve 1

15. `f = factorial(n - 1)`

▸ factorial	n → 2 f → 1
factorial	n → 3

16.
```
r = n * f
return r
```

▸ factorial	n → 2 f → 1 r → 2
factorial	n → 3

`factorial(2)` devuelve 2

17. `f = factorial(n - 1)`

▸ factorial	n → 3 f → 2

18.
```
r = n * f
return r
```

▸ factorial	n → 3 f → 2 r → 6

`factorial(3)` devuelve 6

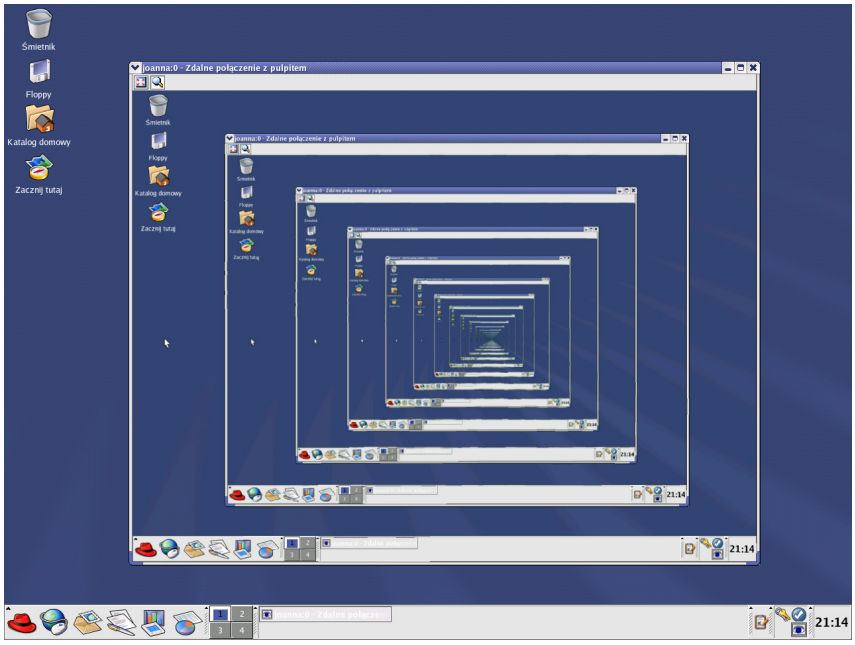


Figura 18.2: Otra imagen recursiva: captura de pantalla de RedHat.

18.6 Algoritmos recursivos y algoritmos iterativos

Llamaremos *algoritmos recursivos* a aquellos que realizan llamadas recursivas para llegar al resultado, y *algoritmos iterativos* a aquellos que llegan a un resultado a través de una iteración mediante un ciclo definido o indefinido.

Todo algoritmo recursivo puede expresarse como iterativo y viceversa. Sin embargo, según las condiciones del problema a resolver podrá ser preferible utilizar la solución recursiva o la iterativa.

Una posible implementación iterativa de la función `factorial` vista anteriormente sería:

```
def factorial(n):
    """Precondición: n entero >= 0
       Devuelve: n!"""
    fact = 1
    for num in range(n, 1, -1):
        fact *= num
    return fact
```

Se puede ver que en este caso no es necesario incluir un caso base, ya que el mismo ciclo incluye una condición de corte, pero que sí es necesario incluir un acumulador, que en el caso recursivo no era necesario.

Por otro lado, si hiciéramos el seguimiento de esta función, como se hizo para la versión recursiva, veríamos que la pila de ejecución siempre tiene un único marco, en el cual se van modificando los valores de `num` y `fact`.

Es por esto que, en general, las versiones recursivas de los algoritmos utilizan más memoria (ya que la pila de ejecución se guarda en memoria) pero suelen ser más elegantes.

18.7 Un ejemplo de recursión elegante

Consideremos ahora otro problema que puede ser resuelto de forma elegante mediante un algoritmo recursivo.

La función `potencia(b, n)`, vista en unidades anteriores, realizaba `n` iteraciones para poder obtener el valor de b^n. Sin embargo, es posible optimizarla teniendo en cuenta que:

$$b^n = b^{n/2} \cdot b^{n/2} \qquad \text{si } n \text{ es par.}$$
$$b^n = b^{(n-1)/2} \cdot b^{(n-1)/2} \cdot b \qquad \text{si } n \text{ es impar.}$$

Antes de programar cualquier función recursiva es necesario decidir cuál será el *caso base* y cuál el *caso recursivo*. Para esta función, tomaremos $n = 0$ como el caso base, en el que devolveremos 1; y el caso recursivo tendrá dos partes, correspondientes a los dos posibles grupos de valores de n.

```
def potencia(b,n):
    """Precondición: n >= 0
       Devuelve: b^n."""

    if n <= 0:
        # caso base
        return 1

    if n % 2 == 0:
        # caso n par
        p = potencia(b, n // 2)
        return p * p
    else:
```

```
        # caso n impar
        p = potencia(b, (n - 1) // 2)
        return p * p * b
```

El uso de la variable `p` en este caso no es optativo, ya que es una de las ventajas principales de esta implementación: se aprovecha el resultado calculado en lugar de tener que calcularlo dos veces. Vemos que este código funciona correctamente:

```
>>> potencia(2, 10)
1024
>>> potencia(3, 3)
27
>>> potencia(5, 0)
1
```

El orden de las llamadas, haciendo un seguimiento simplificado de la función será:

1.	`potencia(2, 10)`			
2.	`p = potencia(2, 5)`	b → 2	n → 10	
3.	`p = potencia(2, 2)`	b → 2	n → 5	
4.	`p = potencia(2, 1)`	b → 2	n → 2	
5.	`p = potencia(2, 0)`	b → 2	n → 1	
6.	`return 1`	b → 2	n → 0	
7.	`return 1 * 1 * 2`	b → 2	n → 1	p → 1
8.	`return 2 * 2`	b → 2	n → 2	p → 2
9.	`return 4 * 4 * 2`	b → 2	n → 5	p → 4
10.	`return 32 * 32`	b → 2	n → 10	p → 32

Se puede ver, entonces, que para calcular 2^{10} se realizaron 5 llamadas a `potencia`, mientras que en la implementación más sencilla se realizaban 10 iteraciones. Y esta optimización será cada vez más importante a medida que aumenta `n`: por ejemplo para $n = 100$ se realizarán 8 llamadas recursivas, y para $n = 1000$ 11 llamadas.

Para transformar este algoritmo recursivo en un algoritmo iterativo, es necesario *simular* la pila de llamadas a funciones mediante una pila que almacene los valores que sean necesarios. En este caso, lo que apilaremos será si el valor de `n` es par o no.

```
def potencia(b, n):
    """Precondición: n >= 0
       Devuelve: b^n."""

    pila = []
    while n > 0:
        if n % 2 == 0:
            pila.append(True)
            n //= 2
        else:
            pila.append(False)
```

```
            n = (n - 1) // 2

    p = 1
    while pila:
        es_par = pila.pop()
        if es_par:
            p *= p
        else:
            p *= p * b

    return p
```

Como se puede ver, este código es mucho más complejo que la versión recursiva. Esto se debe a que utilizando recursión el uso de la pila de llamadas a funciones oculta el proceso de apilado y desapilado y permite concentrarse en la parte importante del algoritmo.

18.8 Un ejemplo de recursión poco eficiente

Del ejemplo anterior se podría deducir que siempre es mejor utilizar algoritmos recursivos; sin embargo —como ya se dijo— cada situación debe ser analizada por separado.

Un ejemplo clásico en el cual la recursión tiene un resultado muy poco eficiente es el de los números de Fibonacci. La sucesión de Fibonacci está definida por la siguiente relación:

$$
\begin{aligned}
F_n &= 0 && \text{si } n = 0 \\
F_n &= 1 && \text{si } n = 1 \\
F_n &= F_{n-1} + F_{n-2} && \text{si } n > 1
\end{aligned}
$$

Los primeros números de esta sucesión son: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

Dada la definición recursiva de la sucesión, puede resultar muy tentador escribir una función que calcule en valor de `fib(n)` de la siguiente forma:

```
def fib(n):
    """Precondición: n >= 0.
       Devuelve: el número de Fibonacci número n."""
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Si bien esta implementación es muy sencilla y elegante, también es extremadamente poco eficiente: para calcular `fib(n - 1)` es necesario calcular `fib(n - 2)`, que luego volverá a ser calculado para obtener el valor `fib(n)`.

Por ejemplo, una simple llamada a `fib(5)`, generaría recursivamente todas las llamadas ilustradas en la Figura 18.3. Puede verse que muchas de estas llamadas están repetidas, generando un total de 15 llamadas a la función `fib`, sólo para devolver el valor F_5.

En este caso, será mucho más conveniente utilizar una versión iterativa, que vaya almacenando los valores de las dos variables anteriores a medida que los va calculando.

```
def fib(n):
    """Precondición: n >= 0.
       Devuelve: el número de Fibonacci número n."""
    if n == 0 or n == 1:
```

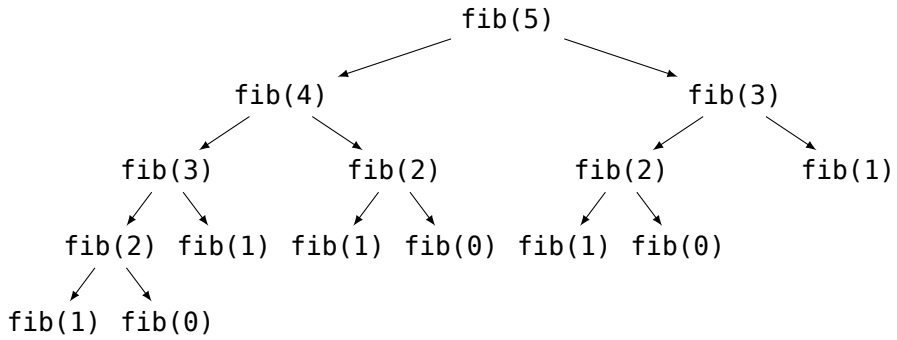


Figura 18.3: Árbol de llamadas para `fib(5)`

```
        return n
    ant2 = 0
    ant1 = 1
    for i in range(2, n + 1):
        fibn = ant1 + ant2
        ant2 = ant1
        ant1 = fibn
    return fibn
```

Vemos que el caso base es el mismo para ambos algoritmos, pero que en el caso iterativo se calcula el número de Fibonacci de forma incremental, de modo que para obtener el valor de `fib(n)` se harán $n - 1$ iteraciones.[1]

> **Atención**
>
> En definitiva, vemos que un algoritmo recursivo **no** es mejor que uno iterativo, ni viceversa. En cada situación será conveniente analizar cuál algoritmo provee la solución al problema de forma más clara y eficiente.

18.9 Diseño de algoritmos recursivos

Hasta el momento vimos que hay muchas funciones matemáticas que se definen o que pueden desarrollarse de forma recursiva, pero puede aplicarse recursividad a muchos problemas que no sean explicitamente recursivos. Diseñar un algoritmo recursivo es un proceso sistematizable.

En general en el proceso para plantear un algoritmo recursivo necesitamos resolver estos tres problemas:

Caso base: Necesitamos definir uno o más casos bases de acuerdo a nuestro problema. Como regla general tratamos de pensar como caso base a las condiciones sobre las cuales es más fácil resolver nuestro problema. Por ejemplo, si estruviéramos trabajando sobre listas o cadenas probablemente sepamos la respuesta a nuestro problema si tuviéramos una secuencia vacía, o si estuviéramos trabajando sobre conjuntos de elementos probablemente la respuesta fuera evidente cuando tengamos un solo elemento.

[1] ¿Y es la iterativa la mejor implementación? Particularmente para el caso de la sucesión de Fibonacci, puede resolverse matemáticamente la ecuación de recurrencia para llegar a una fórmula cerrada. Así se obtiene $F_n = \left[\frac{\varphi^n}{\sqrt{5}}\right]$, con $\varphi = \frac{1+\sqrt{5}}{2}$ el "número de oro" y el operador $[\dots]$ como el redondeo entero.

Caso recursivo o caso general: Este es el caso que va a efectuar la llamada recursiva. La idea de este caso es reducir el problema a un problema más sencillo, del cual se hará cargo la llamada recursiva, y luego poder ensamblar la solución al problema original. Ampliaremos esto más adelante.

Convergencia: Necesitamos que la reducción que se haga en el caso recursivo converja hacia los casos bases, de modo que la recursión alguna vez termine. Esto es, si dijimos que el caso base se resolvía cuando teníamos una lista vacía, las operaciones del caso recursivo tienen que reducir reiteradamente la lista hasta que la misma quede vacía.

Si podemos hacer estas tres cosas, tendremos un algoritmo recursivo para nuestro problema.

18.10 Un primer diseño recursivo

Supongamos que queremos programar una función `sumar(lista)` que determine en forma recursiva la suma de una secuencia `lista` de números.

Como caso base debemos elegir un caso sencillo de verificar. El caso más sencillo de verificar es uno en el que ni siquiera necesitamos computar algo: Si la lista está vacía es evidente que la suma da cero.

Nuestro caso base será algo así como:

```
    if len(lista) == 0:
        return 0
```

Queremos converger a que dada cualquier lista de la reducción de nuestro problema terminemos en el caso base. Hay muchas maneras de reducir una lista para terminar teniendo cero elementos pero para este caso vamos a proponer la más fácil: si cada llamada recursiva saca un elemento, tarde o temprano covergeremos a una lista vacía.

Nuestra llamada recursiva podría ser algo así como:

```
   sumar(lista[1:])
```

Lo más complejo ahora es pensar el caso general.

Dijimos que íbamos a retirar un elemento de la lista por vez y hacer una llamada recursiva. Olvidémonos por un momento de la recursividad e imaginemos que *ya* tenemos una función `sumar2` que sabe sumar los elementos de una `lista` y que lo hace bien. Intentemos resolver el problema inverso: si agregamos un elemento `x` al principio de la lista (obteniendo `[x] + lista`), ¿podemos calcular la suma de la nueva lista? ¿Podemos resolver el problema más grande con la solución al problema más pequeño? La solución es sencilla: La suma de la lista ampliada será `x` más la suma de la lista original (que podemos calcular como `sumar2(lista)`).

Es decir, la solución al problema este que planteamos sería así:

```
def sumar(lista):
   """Precondición: len(lista) >= 1.
      Devuelve: La suma de los elementos en la lista."""
   return lista[0] + sumar2(lista[1:])
```

Podemos ver que si tuviéramos implementada `sumar2` entonces `sumar` funcionaría bien. Volvamos ahora a recursividad: Si sabemos resolver el caso general en función a la solución del caso simplificado de la llamada recursiva, si existen casos bases que cortan la recursión y si además la recursión converge hacia los casos bases tenemos resuelto el problema completo. La función que asumimos que funcionaba *es la misma* que acabamos de implementar.

Cuando diseñamos una función recursiva tenemos que dar este *salto de fé*: asumir que la función del paso recursivo ya funciona; nosotros lo que vamos a implementar es una función que logra concatenar el resultado del subproblema y ensamblarlo con nuestro problema mayor. Si hacemos esto bien entonces todo funciona.

Finalmente nuestra primera función recursiva quedaría:

```
def sumar(lista):
   """Devuelve la suma de los elementos en la lista."""
   if len(lista) == 0:
       return 0
   return lista[0] + sumar(lista[1:])
```

18.11 Pasaje de la información

Dentro de los problemas recursivos no siempre es inmediato establecer cómo se va a propagar la información entre las llamadas recursivas, es decir, la reducción de la solución de los subproblemas en la solución del problema general.

En todos los ejemplos presentados hasta el momento la información del resultado se propagó desde las hojas del árbol de llamadas (los casos bases) hacia las funciones invocantes (mediante la instrucción `return`). Por ejemplo, para resolver el resultado de Fibonacci F_5 se utilizan únicamente los resultados computados por F_4 y F_3, y no se recibe ningún dato adicional de la función invocante (más allá del parámetro `n=5`). Esto no siempre es así, en algunos problemas sí se hace necesario propagar información "hacia abajo". Y en otros casos, si bien no es necesario, puede tener ventajas adicionales.

Por ejemplo, podríamos reescribir la función `sumar` de esta forma:

```
def sumar(lista, suma=0):
    """Devuelve la suma de los elementos en la lista."""
    if len(lista) == 0:
        return suma
    return sumar(lista[1:], lista[0] + suma)
```

Puede observarse que en esta implementación en vez de *esperar* a que se resuelva el cómputo de la parte recursiva para ensamblar la solución e ir resolviendo los cálculos parciales desde el final de la lista hacia el principio, le *pasamos* la solución parcial a la llamada recursiva. Finalmente el caso base devuelve la suma de los cálculos que se realizaron de principio a final y cada llamada recursiva devuelve este resultado.

No profundizaremos más en el tema, pero la particularidad de que lo último que se realice en el caso general sea la llamada recursiva (sin realizar ninguna operación adicional sobre el resultado de esta llamada) se conoce como *recursividad de cola*. La recursividad de cola es de interés porque implica muy poco esfuerzo reescribir una versión iterativa y no recursiva del algoritmo. Esto es inmediato: como lo último que se hace es la llamada recursiva entonces no hace falta seguir *recordando* el contexto de la llamada anterior cuando se hace la siguiente, entonces no es necesario utilizar la pila de ejecución. El código anterior puede reescribirse como

```
def sumar(lista):
    """Devuelve la suma de los elementos en la lista."""
    suma = 0
    while True:
        if len(lista) == 0:
            return suma
        lista, suma = lista[1:], lista[0] + suma
```

tan solo reemplazando la recursión por un bucle y actualizando las variables según los parámetros de la llamada recursiva.

18.12 Modificación de la firma

La *firma* de una función es su nombre, más los parámetros que recibe, más los valores que devuelve. Para invocar una función cualquiera, es suficiente con saber cómo es su firma, y no es necesario saber cómo es la implementación interna. Ahora bien, si cambiamos la lista de parámetros o el tipo de dato del valor de retorno de la función, estamos cambiando su firma, y eso nos obliga a cambiar cualquier lugar del código que contenga alguna llamada a la función.

En el ejemplo de `sumar` implementada con recursividad de cola nos vimos obligados a modificar la firma de la función agregando el parámetro `suma` que no formaba parte del problema inicial. Pudimos hacerlo elegantemente utilizando un valor por omisión (`suma=0`), pero la firma de todos modos quedó confusa.

Hay casos en los que no podemos salvar un cambio en la firma. Por ejemplo, supongamos que queremos diseñar una función recursiva que calcule el promedio de una secuencia de números.

Como ya sabemos diseñar funciones recursivas intuimos que el caso base será cuando la lista esté vacía y que la reduciremos sacando de a un elemento por vez. El cuerpo de nuestra función será algo así:

```
def promediar(lista):
    if len(lista) == 0:
        return ???
    promediar(lista[1:]) ???
```

Ahora bien, con esto no alcanza para resolver el problema.

Para calcular un promedio necesitamos computar tanto una suma como contar la cantidad de elementos. Las funciones van a estar computando dos valores cuando el resultado del problema es evidentemente uno solo. Si bien puede elaborarse una solución similar a la que ya ensayamos con `sumar` complicaría innecesariamente el código. Es preferible modificar la firma de la función.

Implementemos el problema resolviendo primero la llamada recursiva (en una función diferente que llamaremos `_promediar`) y luego ensamblando:

```
def _promediar(lista):
    if len(lista) == 0:
        return 0, 0
    suma, cantidad = _promediar(lista[1:])
    return lista[0] + suma, cantidad + 1
```

Puede verse que esta función cumple con las reglas de diseño de recursividad que describimos antes. Con lo que no cumple esta función es con la firma natural de la función `promediar` que queríamos diseñar, ya que `_promediar` devuelve dos cosas y no una.

Esto no invalida nuestra solución, pero la misma está incompleta. Lo que debemos hacer es implementar una función *wrapper* (envoltorio) que lo que haga es operar como *cara visible* para el usuario de la función que hace realmente el trabajo. A esta función sí la vamos a llamar `promediar`, ya que va a cumplir con la firma deseada:

```
def promediar(lista):
   """Devuelve el promedio de los elementos de una lista de números."""
   suma, cantidad = _promediar(lista)
```

```
    return suma / cantidad
```

Notar que si bien la función visible `promediar` no es recursiva, sí lo es la función `_promediar` que es la que realiza el trabajo, por lo que el conjunto se considera recursivo.

Además de para adaptar la firma de la función recursiva, las funciones wrapper se suelen utilizar para simplificar el código de las funciones recursivas. Por ejemplo, si quisiéramos hacer validaciones de los parámetros, no querríamos que las mismas se reiteraran en cada iteración recursiva porque consumirían recursos innecesarios. Entonces las podemos resolver en la función wrapper, antes de empezar la recursión.

Por ejemplo, en la sección 18.7 implementamos la potencia en forma recursiva con la restricción $n \geq 0$. Pero dado que $b^n = \left(\frac{1}{b}\right)^{-n}$ podemos aprovechar el código implementado para resolver para cualquier n entero. Podríamos modificar el código de `potencia` para incluir este caso, pero se reiteraría la comprobación en cada recursión. Para un caso así sería más sencillo construir una función wrapper e incluir ahí todo lo que consideremos necesario[2]. Habiendo renombrado la función original como `_potencia`, nuestro wrapper sería:

```
def potencia(b, n):
    """Precondición: n es entero
       Devuelve: b^n."""
    if n < 0:
        b = 1 / b
        n = -n
    return _potencia(b, n)
```

18.13 Limitaciones

Si creamos una función sin *caso base*, obtendremos el equivalente recursivo de un bucle infinito. Sin embargo, como cada llamada recursiva agrega un elemento a la pila de llamadas a funciones y la memoria de nuestras computadoras no es infinita, el ciclo deberá terminarse cuando se agote la memoria disponible.

En particular, en Python, para evitar que la memoria se termine, la pila de ejecución de funciones tiene un límite. Es decir, que si se ejecuta un código como el que sigue:

```
def inutil(n):
    return inutil(n - 1)
```

Se obtendrá un resultado como el siguiente:

```
>>> inutil(1)
  File "<stdin>", line 2, in inutil
  File "<stdin>", line 2, in inutil
  (...)
  File "<stdin>", line 2, in inutil
RecursionError: maximum recursion depth exceeded
```

El límite por omisión es de 1000 llamadas recursivas. Es posible modificar el tamaño máximo de la pila de recursión mediante la instrucción `sys.setrecursionlimit(n)`. Sin embargo, si se está alcanzando este límite suele ser una buena idea pensar si realmente el algoritmo recursivo es el que mejor resuelve el problema.

[2] Y más allá de lo que se mencionó, ¿hace falta resolver la recursión si $b = 0$, $b = 1$, $b = -1$?

Sabías que…

Existen algunos lenguajes *funcionales*, como Haskell, ML, o Scheme, en los cuales la recursión es la única forma de realizar un ciclo. Es decir, no existen construcciones `while` ni `for`.

Estos lenguajes cuentan con optimización de recursión de cola, una optimización para que cuando se identifique que la recursión es de cola, no se apile el estado de la función innecesariamente, evitando el consumo adicional de memoria mencionado anteriormente.

La ejecución de todas las funciones con recursión de cola vistas en esta unidad podrían ser optimizada por el compilador o intérprete del lenguaje.

18.14 Resumen

- A medida que se realizan llamadas a funciones, el estado cada función se almacena en la *pila de ejecución*.
- Esto permite que sea posible que una función se llame a sí misma, pero que las variables dentro de la función tomen distintos valores.
- La **recursión** es el proceso en el cual una función se invoca a sí misma. Este proceso permite crear un nuevo tipo de ciclos.
- Siempre que se escribe una función recursiva es importante considerar el **caso base** (el que detendrá la recursión) y el **caso recursivo** (el que realizará la llamada recursiva). Una función recursiva sin caso base es equivalente a un bucle infinito.
- Una función no es mejor ni peor por ser recursiva. En cada situación a resolver puede ser conveniente utilizar una solución recursiva o una iterativa. Para elegir una o la otra será necesario analizar las características de elegancia y eficiencia.
- Al diseñar funciones recursivas muchas veces puede ser útil implementar una función **wrapper**, por ejemplo para adaptar la firma de la función, validar parámetros, inicializar datos o manejar excepciones.

18.15 Ejercicios

Ejercicio 18.15.1. Escribir una función recursiva que reciba un número positivo n y devuelva la cantidad de dígitos que tiene.

Ejercicio 18.15.2. Escribir una función recursiva que simule el siguiente experimento: Se tiene una rata en una jaula con 3 caminos, entre los cuales elige al azar (cada uno tiene la misma probabilidad), si elige el *1* luego de 3 minutos vuelve a la jaula, si elige el *2* luego de 5 minutos vuelve a la jaula, en el caso de elegir el *3* luego de 7 minutos sale de la jaula. La rata no aprende, siempre elige entre los 3 caminos con la misma probabilidad, pero quiere su libertad, por lo que recorrerá los caminos hasta salir de la jaula.

La función debe devolver el tiempo que tarda la rata en salir de la jaula.

Ejercicio 18.15.3. Escribir una función recursiva que reciba 2 enteros n y b y devuelva `True` si n es potencia de b.

Ejemplos:

```
es_potencia(8, 2) -> True
es_potencia(64, 4) -> True
es_potencia(70, 10) -> False
```

Ejercicio 18.15.4. Escribir una funcion recursiva que reciba como parámetros dos cadenas a y b, y devuelva una lista con las posiciones en donde se encuentra b dentro de a.

Ejemplo:

```
posiciones_de("Un tete a tete con Tete", "te") -> [3, 5, 10, 12, 21]
```

Ejercicio 18.15.5. Escribir dos funciones mutualmente recursivas `par(n)` e `impar(n)` que determinen la paridad del numero natural dado, conociendo solo que:

- 1 es impar.
- Si un número es impar, su antecesor es par; y viceversa.

Ejercicio 18.15.6. Escribir una función recursiva que calcule recursivamente el n-ésimo número triangular (el número $1 + 2 + 3 + \cdots + n$).

Ejercicio 18.15.7. Escribir una función que calcule recursivamente cuántos elementos hay en una pila, suponiendo que la pila sólo tiene los métodos `apilar` y `desapilar`, y no altere el contenido de la pila.

Ejercicio 18.15.8. Escribir una funcion recursiva que encuentre el mayor elemento de una lista.

Ejercicio 18.15.9. Escribir una función recursiva para replicar los elementos de una lista una cantidad n de veces. Por ejemplo:

```
replicar([1, 3, 3, 7], 2) -> ([1, 1, 3, 3, 3, 3, 7, 7])
```

Ejercicio 18.15.10. Escribir una funcion recursiva que dada una cadena determine si en la misma hay más letras A o letras E.

Ejercicio 18.15.11. El triángulo de Pascal es un arreglo triangular de números que se define de la siguiente manera: Las filas se enumeran desde $n = 0$, de arriba hacia abajo. Los valores de cada fila se enumeran desde $k = 0$ (de izquierda a derecha). Los valores que se encuentran en los bordes del triángulo son 1. Cualquier otro valor se calcula sumando los dos valores contiguos de la fila de arriba.

$n = 0$							1						
$n = 1$						1		1					
$n = 2$					1		2		1				
$n = 3$				1		3		3		1			
$n = 4$			1		4		6		4		1		
$n = 5$		1		5		10		10		5		1	
$n = 6$	1		6		15		20		15		6		1

Escribir la función recursiva `pascal(n, k)` que calcula el valor que se encuentra en la fila `n` y la columna `k`. Ejemplo: `pascal(5, 2) -> 10`

Ejercicio 18.15.12. Ya sabemos que la implementación recursiva del cálculo del número de Fibonacci ($F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$) es ineficiente porque muchas de las ramas calculan reiteradamente los mismos valores.

Escribir una función `fibonacci(n)` que calcule el enésimo número de Fibonacci de forma recursiva pero que utilice un diccionario para almacenar los valores ya computados y no computarlos más de una vez.

Nota: Será necesario implementar una función wrapper para cumplir con la firma de la función pedida.

Unidad 19

Ordenar listas

Al estudiar las listas de Python, vimos que poseen un método `sort` que las ordena de menor a mayor de acuerdo a una clave arbitraria.

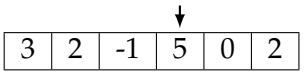
Sin embargo, no todas las estructuras cuentan con un método `sort` que las ordene. Es por ello que en esta unidad nos plantearemos cómo se hace para ordenar cuando no hay un método `sort`, y cuánto cuesta ordenar.

Ante todo una advertencia: hay varias maneras de ordenar, y no todas cuestan lo mismo. Vamos a empezar viendo las más sencillas de escribir (que en general suelen ser las más caras).

19.1 Ordenamiento por selección

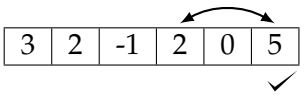
El método de *ordenamiento por selección* se basa en la siguiente idea:

- **Paso 1.1**: Buscar el mayor de todos los elementos de la lista.



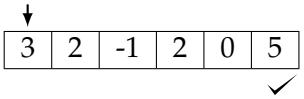
Encuentra el valor 5 en la posición 3.

- **Paso 1.2**: Poner el mayor al final (intercambiar el que está en la última posición de la lista con el mayor encontrado).



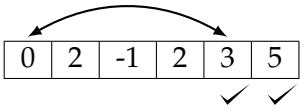
Intercambia el elemento de la posición 3 con el de la posición 5.
En la última posición de la lista está el mayor de todos.

- **Paso 2.1**: Buscar el mayor de todos los elementos del segmento de la lista entre la primera y la anteúltima posición.



Encuentra el valor 3 en la posición 0.

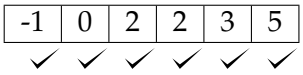
- **Paso 2.2**: Poner el mayor al final del segmento (intercambiar el que está en la última posición del segmento –o sea anteúltima posición de la lista– con el mayor encontrado).



Intercambia el elemento de la posición 0 con el valor de la posición 4.
En la anteúltima y última posición de la lista están los dos mayores en su posición definitiva.

...

- **Paso n**: Se termina cuando queda un único elemento sin tratar: el que está en la primera posición de la lista, y que es el menor de todos porque todos los mayores fueron reubicados.



La lista se encuentra ordenada.

La implementación en Python puede verse en el Código 19.1.

Código 19.1 `seleccion.py`**:** Ordena una lista por selección

```
def ord_seleccion(lista):
    """Ordena una lista de elementos según el método de selección.
       Pre: los elementos de la lista deben ser comparables.
       Post: la lista está ordenada."""

    # posición final del segmento a tratar
    n = len(lista) - 1

    # mientras haya al menos 2 elementos para ordenar
    while n > 0:
        # posición del mayor valor del segmento
        p = buscar_max(lista, 0, n)

        # intercambiar el valor que está en p con el valor que
        # está en la última posición del segmento
        lista[p], lista[n] = lista[n], lista[p]
        print("DEBUG: ", p, n, lista)

        # reducir el segmento en 1
        n = n - 1

def buscar_max(lista, a, b):
    """Devuelve la posición del máximo elemento en un segmento de
       lista de elementos comparables.
       La lista no debe ser vacía.
       a y b son las posiciones inicial y final del segmento"""

    pos_max = a
    for i in range(a + 1, b + 1):
        if lista[i] > lista[pos_max]:
            pos_max = i
    return pos_max
```

La función principal, `ord_seleccion` es la encargada de recorrer la lista, ubicando el mayor elemento al final del segmento y luego reduciendo el segmento a analizar.

La función `buscar_max` busca el mayor elemento de un segmento de la lista y devuelve su posición.

A continuación, algunas una ejecuciones de prueba de ese código:

```
>>> l = [3, 2, -1, 5, 0, 2]
>>> ord_seleccion(l)
DEBUG:  3 5 [3, 2, -1, 2, 0, 5]
DEBUG:  0 4 [0, 2, -1, 2, 3, 5]
DEBUG:  1 3 [0, 2, -1, 2, 3, 5]
DEBUG:  1 2 [0, -1, 2, 2, 3, 5]
DEBUG:  0 1 [-1, 0, 2, 2, 3, 5]
>>> l
[-1, 0, 2, 2, 3, 5]
>>> l = []
>>> ord_seleccion(l)
>>> l = [1]
>>> ord_seleccion(l)
>>> l
[1]
>>> l = [1, 2, 3, 4, 5]
>>> ord_seleccion(l)
DEBUG:  4 4 [1, 2, 3, 4, 5]
DEBUG:  3 3 [1, 2, 3, 4, 5]
DEBUG:  2 2 [1, 2, 3, 4, 5]
DEBUG:  1 1 [1, 2, 3, 4, 5]
```

Puede verse que aun cuando la lista está ordenada, se la recorre buscando los mayores elementos y ubicándolos en la misma posición en la que se encuentran.

19.1.1 Invariante en el ordenamiento por selección

Todo ordenamiento tiene un invariante que permite asegurarse de que cada paso que se toma va en la dirección de obtener una lista ordenada.

En el caso del ordenamiento por selección, el invariante es que los elementos desde $n + 1$ hasta el final de la lista están ordenados y son mayores que los elementos de 0 a n; es decir que ya están en su posición definitiva.

19.1.2 ¿Cuánto cuesta ordenar por selección?

Como se puede ver en el código de la función `buscar_max`, para buscar el máximo elemento en un segmento de lista se debe recorrer todo ese segmento, por lo que en nuestro caso debemos recorrer en el primer paso N elementos, en el segundo paso $N - 1$ elementos, en el tercer paso $N - 2$ elementos, etc. Cada visita a un elemento implica una cantidad constante y pequeña de comparaciones (que no depende de N). Por lo tanto tenemos que

$$T(N) \approx c \cdot (2 + 3 + \cdots + N) \approx c \cdot N \cdot (N + 1)/2 \sim N^2$$

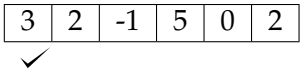
O sea que ordenar por selección una lista de tamaño N insume tiempo del orden de N^2. Como ya se vio, este tiempo es independiente de si la lista estaba previamente ordenda o no.

En cuanto al espacio utilizado, sólo se tiene en memoria la lista que se desea ordenar y algunas variables de tamaño 1.

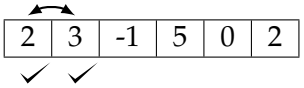
19.2 Ordenamiento por inserción

El método de *ordenamiento por inserción* se basa en la siguiente idea:

- **Paso 0**: Partimos de la misma lista de ejemplo utilizada para el ordenamiento por selección.

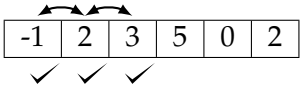


- **Paso 1**: Considerar el segundo elemento de la lista, y ordenarlo respecto del primero, deplazándolo hasta la posición correcta, si corresponde.



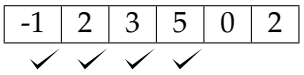
Se desplaza el valor 2 antes de 3.

- **Paso 2**: Considerar el tercer elemento de la lista, y ordenarlo respecto del primero y el segundo, deplazándolo hasta la posición correcta, si corresponde.



Se desplaza el valor -1 antes de 2 y de 3.

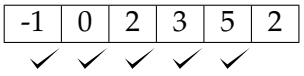
- **Paso 3**: Considerar el cuarto elemento de la lista, y ordenarlo respecto del primero, el segundo y el tercero, deplazándolo hasta la posición correcta, si corresponde.



El 5 está correctamente ubicado respecto de -1,2 y 3 (como el segmento hasta la tercera posición está ordenado, basta con comparar con el tercer elemento del segmento para verificarlo).

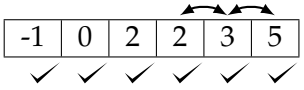
...

- **Paso N-1**:



Todos los elementos excepto el ante-último ya se encuentran ordenados.

- **Paso N**: Considerar el N–ésimo elemento de la lista, y ordenarlo respecto del segmento formado por el primero hasta el $N-1$–ésimo, deplazándolo hasta la posición correcta, si corresponde.



Se desplaza el valor 2 antes de 3 y de 5.

Una posible implementación en Python de este algoritmo se incluye en el Código 19.2.

La función principal, `ord_insercion`, recorre la lista desde el segundo elemento hasta el último, y cuando uno de estos elementos no está ordenado con respecto al anterior, llama a la función auxiliar `reubicar`, que se encarga de colocar el elemento en la posición que le corresponde.

Código 19.2 `insercion.py`**:** Ordena una lista por Inserción

```
def ord_insercion(lista):
    """Ordena una lista de elementos según el método de inserción.
       Pre: los elementos de la lista deben ser comparables.
       Post: la lista está ordenada."""

    for i in range(len(lista) - 1):
        # Si el elemento de la posición i+1 está desordenado respecto
        # al de la posición i, reubicarlo dentro del segmento [0:i]
        if lista[i + 1] < lista[i]:
            reubicar(lista, i + 1)
        print("DEBUG: ", lista)

def reubicar(lista, p):
    """Reubica al elemento que está en la posición p de la lista
       dentro del segmento [0:p-1].
       Pre: p tiene que ser una posicion válida de lista."""

    v = lista[p]

    # Recorrer el segmento [0:p-1] de derecha a izquierda hasta
    # encontrar la posición j tal que lista[j-1] <= v < lista[j].
    j = p
    while j > 0 and v < lista[j - 1]:
        # Desplazar los elementos hacia la derecha, dejando lugar
        # para insertar el elemento v donde corresponda.
        lista[j] = lista[j - 1]
        j -= 1

    lista[j] = v
```

En la función `reubicar` se busca la posición correcta donde debe colocarse el elemento, a la vez que se van corriendo todos los elementos un lugar a la derecha, de modo que cuando se encuentra la posición, el valor a insertar reemplaza al valor que se encontraba allí anteriormente.

En las siguientes ejecuciones puede verse que funciona correctamente.

```
>>> l=[3, 2, -1, 5, 0, 2]
>>> ord_insercion(l)
DEBUG:  [2, 3, -1, 5, 0, 2]
DEBUG:  [-1, 2, 3, 5, 0, 2]
DEBUG:  [-1, 2, 3, 5, 0, 2]
DEBUG:  [-1, 0, 2, 3, 5, 2]
DEBUG:  [-1, 0, 2, 2, 3, 5]
>>> l
[-1, 0, 2, 2, 3, 5]
>>> l = []
>>> ord_insercion(l)
>>> l = [1]
>>> ord_insercion(l)
>>> l
[1]
>>> l=[1, 2, 3, 4, 5, 6]
```

```
>>> ord_insercion(l)
DEBUG:  [1, 2, 3, 4, 5, 6]
DEBUG:  [1, 2, 3, 4, 5, 6]
DEBUG:  [1, 2, 3, 4, 5, 6]
DEBUG:  [1, 2, 3, 4, 5, 6]
DEBUG:  [1, 2, 3, 4, 5, 6]
>>> l
[1, 2, 3, 4, 5, 6]
```

19.2.1 Invariante del ordenamiento por inserción

En el ordenamiento por inserción, en cada paso se considera que los elementos que se encuentran en el segmento de 0 a i están ordenados, de manera que agregar un nuevo elemento implica colocarlo en la posición correspondiente y el segmento seguirá ordenado.

19.2.2 ¿Cuánto cuesta ordenar por inserción?

Del Código 19.2 se puede ver que la función principal avanza por la lista de izquierda a derecha, mientras que la función `reubicar` cambia los elementos de lugar de derecha a izquierda.

Lo peor que le puede pasar a un elemento que está en la posición j es que deba ser ubicado al principio de la lista. Y lo peor que le puede pasar a una lista es que todos sus elementos deban ser reubicados.

Por ejemplo, en la lista `[10, 8, 6, 2, -2, -5]`, todos los elementos deben ser reubicados al principio de la lista.

En el primer paso, el segundo elemento se debe intercambiar con el primero; en el segundo paso, el tercer elemento se compara con el segundo y el primer elemento, y se ubica adelante de todo; en el tercer paso, el cuarto elemento se compara con el tercero, el segundo y el primer elemento, y se ubica adelante de todo; etc…

$$T(N) \approx c \cdot (2 + 3 + \cdots + N) \approx c \cdot N \cdot (N + 1)/2 \sim N^2$$

Es decir que ordenar por inserción una lista de tamaño N puede insumir (en el peor caso) tiempo del orden de N^2. En cuanto al espacio utilizado, nuevamente sólo se tiene en memoria la lista que se desea ordenar y algunas variables de tamaño 1.

19.2.3 Inserción en una lista ordenada

Sin embargo, algo interesante a notar es que cuando la lista se encuentra ordenada, este algoritmo no hace ningún movimiento de elementos, simplemente compara cada elemento con el anterior, y si es mayor sigue adelante.

Es decir que para el caso de una lista de N elementos que se encuentra ordenada, el tiempo que insume el algoritmo de inserción es:

$$T(N) \sim N \tag{19.1}$$

19.3 Resumen

- El **ordenamiento por selección** es uno de los más sencillos, pero es bastante ineficiente: se basa en la idea de *buscar el máximo* en una secuencia, ubicarlo al final y seguir analizando la secuencia sin el último elemento.

Tiene como ventaja que hace una baja cantidad de intercambios (N), pero como desventaja que necesita una alta cantidad de comparaciones (N^2). Siempre tiene el mismo comportamiento.

- El **ordenamiento por inserción** es un algoritmo bastante intuitivo y se suele usar para ordenar en la vida real. Se basa en la idea de ir *insertando ordenadamente*: en cada paso se considera la inserción de un elemento más de secuencia y la inserción se empieza a hacer desde el final de los datos ya ordenados.

  Tiene como ventaja que en el caso de tener los datos ya ordenados no hace ningún intercambio (y hace sólo $N - 1$ comparaciones). En el peor caso, cuando la secuencia está invertida, se hace una gran cantidad de intercambios y comparaciones (N^2). Si bien es un algoritmo ineficiente, para secuencias cortas el tiempo de ejecución es bastante bueno.

19.4 Ejercicios

Ejercicio 19.4.1. Implementar una función que reciba una lista y devuelva otra lista con los mismos elementos que la primera, ordenados de mayor a menor mediante el método de inserción.

Unidad 20

Algunos ordenamientos recursivos

Los métodos de ordenamiento vistos en la unidad anterior eran métodos iterativos cuyo tiempo estaba relacionado con N^2.

En esta unidad veremos dos métodos de ordenamiento basados en un planteo recursivo del problema, que nos permitirán obtener el mismo resultado de forma más eficiente.

20.1 Ordenamiento por mezcla, o *Merge sort*

Merge sort se basa en la siguiente idea:

1. Si la lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. De lo contrario hacer lo siguiente:
2. Dividir la lista al medio, formando dos sublistas de (aproximadamente) el mismo tamaño cada una.
3. Ordenar cada una de esas dos sublistas (usando este mismo método).
4. Una vez que se ordenaron ambas sublistas, intercalarlas de manera ordenada.

Por ejemplo, si la lista original es `[6, 7, -1, 0, 5, 2, 3, 8]` deberemos ordenar recursivamente `[6, 7, -1, 0]` y `[5, 2, 3, 8]` con lo cual obtendremos `[-1, 0, 6, 7]` y `[2, 3, 5, 8]`. Si intercalamos ordenadamente las dos listas ordenadas obtenemos la solución buscada: `[-1, 0, 2, 3, 5, 6, 7, 8]`.

Diseñamos la función `merge_sort(lista)`:

1. Si lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. Se devuelve lista tal cual.
2. De lo contrario:
   (a) `medio = len(lista) / 2`
   (b) `izq = merge_sort(lista[:m])`
   (c) `der = merge_sort(lista[m:])`
   (d) Se devuelve `merge(izq, der)`.

Falta sólo diseñar la función `merge`: dadas dos listas ordenadas debe obtener una nueva lista que resulte de intercalar a ambas de manera ordenada:

1. Utilizaremos dos índices, `i` y `j`, para recorrer cada una de las dos listas.
2. Utilizaremos una tercera lista, `resultado`, donde almacenaremos el resultado.
3. Mientras `i` sea menor que el largo de `lista1` y `j` menor que el largo de `lista2`, significa que hay elementos para comparar en ambas listas.
   (a) Si el menor es el de `lista1`:
      i. Agregar el elemento `i` de `lista1` al final del resultado.
      ii. Avanzar el índice `i`.
   (b) de lo contrario:
      i. Agregar el elemento `j` de `lista2` al final del resultado.
      ii. Avanzar el índice `j`.
4. Una vez que una de las dos listas se termina, simplemente hay que agregar todo lo que queda en la otra al final del resultado.

El código resultante del diseño de ambas funciones puede verse en el Código 20.1.

Sabías que…

El método que hemos usado para resolver este problema se llama **División y Conquista**, y se aplica en las situaciones en las que vale el siguiente principio:

Para obtener una solución es posible partir el problema en varios subproblemas de tamaño menor, resolver cada uno de esos subproblemas por separado aplicando la misma técnica (en nuestro caso ordenar por mezcla cada una de las dos sublistas), y finalmente juntar estas soluciones parciales en una solución completa del problema mayor (en nuestro caso la intercalación ordenada de las dos sublistas ordenadas).

Como siempre sucede con las soluciones recursivas, debemos encontrar un caso base en el cual no se aplica la llamada recursiva (en nuestro caso la base sería el paso 1: Si la lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer). Además debemos asegurar que siempre se alcanza el caso base, y en nuestro caso aseguramos eso porque las lista original se divide siempre en mitades cuando su longitud es mayor que 1.

20.1.1 ¿Cuánto cuesta el *Merge sort*?

Sea N la longitud de la lista. Observamos lo siguiente:

- Para intercalar dos listas de longitud $N/2$ hace falta recorrer ambas listas que en total tienen N elementos, por lo que es proporcional a N. Llamemos $a \cdot N$ a ese tiempo.
- Si llamamos $T(N)$ al tiempo que tarda el algoritmo en ordenar una lista de longitud N, vemos que $T(N) = 2\,T(N/2) + a\,N$.
- Además, cuando la lista es pequeña, la operación es de tiempo constante: $T(1) = T(0) = b$.

Para simplificar la cuenta vamos a suponer que $N = 2^k$.

Código 20.1 `mergesort.py`**:** Una implementación de *Merge sort*

```
def merge_sort(lista):
    """Ordena lista mediante el método merge sort.
       Pre: lista debe contener elementos comparables.
       Devuelve: una nueva lista ordenada."""
    if len(lista) < 2:
        return lista
    medio = len(lista) // 2
    izq = merge_sort(lista[:medio])
    der = merge_sort(lista[medio:])
    return merge(izq, der)

def merge(lista1, lista2):
    """Intercala los elementos de lista1 y lista2 de forma ordenada.
       Pre: lista1 y lista2 deben estar ordenadas.
       Devuelve: una lista con los elementos de lista1 y lista2."""

    i, j = 0, 0
    resultado = []

    while(i < len(lista1) and j < len(lista2)):
        if (lista1[i] < lista2[j]):
            resultado.append(lista1[i])
            i += 1
        else:
            resultado.append(lista2[j])
            j += 1

    # Agregar lo que falta
    resultado += lista1[i:]
    resultado += lista2[j:]

    return resultado
```

$$\begin{aligned} T(N) = T(2^k) &= 2\,T(2^{k-1}) + a\,2^k \\ &= 2\,\left(2\,T(2^{k-2}) + a\,2^{k-1}\right) + a\,2^k \\ &= 2^2\,T(2^{k-2}) + a\,2^k + a\,2^k \\ &\;\;\vdots \\ &= 2^i\,T(2^{k-i}) + i\,a\,2^k \\ &\;\;\vdots \\ &= 2^k\,T(1) + k\,a\,2^k \\ &= b\,2^k + k\,a\,2^k \end{aligned}$$

Pero si $N = 2^k$ entonces $k = \log_2 N$, y por lo tanto hemos demostrado que:

$$T(N) = b\,N + a\,N\,\log_2 N. \tag{20.1}$$

Como lo que nos interesa es aproximar el valor, diremos (despreciando el término de menor orden) que

$$T(N) \sim N\,\log_2 N$$

Hemos mostrado entonces un algoritmo que se porta mucho mejor que los que vimos en la unidad pasada (ya que $\log_2 N$ es un número mucho más pequeño que N).

Si analizamos el espacio que consume, vemos que a cada paso la función `merge` genera una nueva lista cuya longitud es la suma de los tamaños de las dos listas, por lo que `merge_sort` duplica el espacio consumido.

20.2 Ordenamiento rápido o *Quick sort*

Quick sort es tal vez el más famoso de los algoritmos recursivos de ordenamiento. Su fama radica en que en la práctica, con casos reales, es uno de los algoritmos más eficientes para ordenar.

Este método se basa en la siguiente idea:

1. Si la lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. De lo contrario hacer lo siguiente:
2. Tomar un elemento de la lista (por ejemplo el primero) al que llamaremos **pivote** y armar a partir de esa lista tres sublistas: la de todos los elementos de la lista menores al pivote, la formada sólo por el pivote, y la de los elementos mayores o iguales al pivote, pero sin contarlo al pivote.
3. Ordenar cada una de esas tres sublistas (usando este mismo método).
4. Concatenar las tres sublistas ya ordenadas.

Por ejemplo, si la lista original es `[6, 7, -1, 0, 5, 2, 3, 8]` consideramos que el pivote es el primer elemento (el 6) y armamos las sublistas `[-1, 0, 5, 2, 3]`, `[6]` y `[7,8]`. Se ordenan recursivamente `[-1, 0, 5, 2, 3]` (obtenemos `[-1, 0, 2, 3, 5]`) y `[7, 8]` (obtenemos la misma) y concatenamos en el orden adecuado, y así obtenemos `[-1, 0, 2, 3, 5, 6, 7, 8]`.

Para diseñar, vemos que lo más importante es conseguir armar las tres listas en las que se parte la lista original. Para eso definiremos una función auxiliar `_partition` que recibe una lista no vacía y devuelve las tres sublistas `menores`, `medio` y `mayores` (incluye los iguales, de haberlos) en las que se parte la lista original usando como pivote al primer elemento.

Contando con la función `_partition`, el diseño del *Quick sort* es muy simple:

1. Si lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. Se devuelve lista tal cual.
2. De lo contrario:
   (a) Dividir la lista en tres, usando `_partition`.
   (b) Llamar a `quick_sort(menores)`, `quick_sort(mayores)`, y concatenarlo con `medio` en el medio.

Por otro lado, en cuanto a la función `_partition(lista)`:

1. Tiene como precondición que la lista es no vacía.
2. Se elige el primer elemento como pivote.
3. Se inicializan como vacías las listas `menores` y `mayores`.
4. Para cada elemento de la lista después del primero:
   (a) Si es menor que el pivote, se lo agrega a `menores`.
   (b) De lo contrario, se lo a agrega a `mayores`.
5. Devolver `menores, [pivote], mayores`

Una primera aproximación a este código se puede ver en el Código 20.2.

20.2.1 ¿Cuánto cuesta el *Quick sort*?

A primera vista, la ecuación del tiempo consumido parece ser la misma que en el *Mergesort*: Una partición que se hace en tiempo lineal más dos llamadas recursivas a mitades de la lista original.

Pero el problema acá es que la partición tomando como pivote `lista[0]` no siempre parte la lista en mitades: puede suceder (y ese es el peor caso) que parta a la lista en (`[]`, `[lista[0]]` , `lista[1:]`) (esto es lo que pasa cuando la lista está ordenada de entrada), y en ese caso se comporta como *selección*.

En cambio, cuando la lista tiene números ubicados de manera aleatoria dentro de ella, podemos imaginar en promedio un comportamiento parecido al del Mergesort, y por lo tanto ahí sí

$$T(N) \sim N \log_2 N$$

Si analizamos el espacio que consume, el código mostrado en 20.2 crea nuevas listas a cada paso, con lo cual al igual que el *Merge sort* utiliza el doble de memoria.

Código 20.2 `quicksort_copia.py`**:** Una primera aproximación al *Quick sort*

```
def quick_sort(lista):
    """Ordena la lista de forma recursiva.
       Pre: los elementos de la lista deben ser comparables.
       Devuelve: una nueva lista con los elementos ordenados."""
    if len(lista) < 2:
        return lista
    menores, medio, mayores = _partition(lista)
    return quick_sort(menores) + medio + quick_sort(mayores)

def _partition(lista):
    """Pre: lista no vacía.
       Devuelve: tres listas: menores, medio y mayores."""
    pivote = lista[0]
    menores = []
    mayores = []
    for x in range(1, len(lista)):
        if lista[x] < pivote:
            menores.append(lista[x])
        else:
            mayores.append(lista[x])
    return menores, [pivote], mayores
```

20.2.2 Una versión mejorada de *Quick sort*

Sin embargo, es posible hacer la partición de otra forma, operando sobre la misma lista recibida, reubicando los elementos en su interior, de modo que no se consuma el doble de memoria.

En este caso, tendremos una función `_quick_sort`, que será muy similar al de la vista anteriormente, con la particularidad de que en lugar de recibir listas cada vez más pequeñas, recibirá los índices de inicio y fin que indican la porción de la lista sobre la que debe operar.

Habrá además una función `quick_sort`, que recibirá la lista, y se encargará de llamar `_quick_sort` con los índices correspondientes.

Por otro lado, la función `_partition` recibirá también los índices de inicio y fin. En este caso, la función se encargará de cambiar de lugar los elementos de la lista, de modo que todos los menores al pivote se encuentren antes de él y todos los mayores se encuentren después.

Existen varias formas de llevar esto a cabo. Este es un posible diseño para la función `_partition`:

1. Elegir el pivote como el primero de los elementos a procesar.
2. Inicializar un índice `menores` con el valor del primer elemento de la porción a procesar.
3. Recorrer los elementos desde el segundo hasta el último a procesar:
   (a) Si el elemento es menor al pivote, incrementar el índice `menores` y de ser necesario, intercambiar el elemento para que pase a ser el último de los menores.
4. Intercambiar el pivote con el último de los menores
5. Devolver la posición del pivote.

Código 20.3 `quicksort.py`**:** Una versión más eficiente de *Quicksort*

```
def quick_sort(lista):
    """Ordena la lista de forma recursiva.
       Pre: los elementos de la lista deben ser comparables.
       Post: la lista está ordenada. """
    _quick_sort(lista, 0, len(lista) - 1)

def _quick_sort(lista, inicio, fin):
    """Función quick_sort recursiva.
       Pre: los índices inicio y fin indican sobre qué porción operar.
       Post: la lista está ordenada."""
    if inicio >= fin:
        return
    menores = _partition(lista, inicio, fin)
    _quick_sort(lista, inicio, menores - 1)
    _quick_sort(lista, menores + 1, fin)

def _partition(lista, inicio, fin):
    """Función partición que trabaja sobre la misma lista.
       Pre: los índices inicio y fin indican sobre qué porción operar.
       Post: los menores están antes que el pivote, los mayores después.
       Devuelve: la posición en la que quedó ubicado el pivote."""
    pivote = lista[inicio]
    menores = inicio

    # Ubicar menores a la izquierda, mayores a la derecha
    for i in range(inicio + 1, fin + 1):
        if lista[i] < pivote:
            menores += 1
            if i != menores:
                _swap(lista, i, menores)
    # Ubicar el pivote al final de los menores
    if inicio != menores:
        _swap(lista, inicio, menores)
    return menores

def _swap(lista, i, j):
    """Intercambia los elementos i y j de lista."""
    lista[j], lista[i] = lista[i], lista[j]
```

El código resultante de este nuevo diseño se reproduce en el Código 20.3.

Este código, si bien más complejo, cumple con el objetivo de proveer un algoritmo de ordenamiento que en el caso promedio tarda $T(N) \sim N\,log_2 N$, sin consumir memoria adicional (más allá de la memoria utilizada por la pila de ejecución).

20.3 Resumen

- Los ordenamientos de selección e inserción, presentados en la unidad anterior son ordenamientos sencillos pero costosos en cantidad de intercambios o de comparaciones. Sin embargo, es posible conseguir ordenamientos con mejor orden utilizando algoritmos recursivos.
- El algoritmo **Merge Sort** consiste en dividir la lista a ordenar hasta que tenga 1 ó 0 elementos y luego combinar la lista de forma ordenada. De esta manera se logra un tiempo proporcional a $N\,logN$. Tiene como desventaja que siempre utiliza el doble de la memoria requerida por la lista a ordenar.
- El algoritmo **Quick Sort** consiste en elegir un elemento, llamado *pivote* y ordenar los elementos de tal forma que todos los menores queden a la izquierda y todos los mayores a la derecha, y a continuación ordenar de la misma forma cada una de las dos sublistas formadas. Puede implementarse de tal forma que opere sobre la misma lista, sin necesidad de utilizar más memoria. Tiene como desventaja que si bien en el caso promedio tarda $N\,logN$, en el peor caso (según cuál sea el pivote elegido) puede llegar a tardar N^2.

20.4 Ejercicios

Ejercicio 20.4.1. Escribir una función `merge_sort_3` que funcione igual que el merge sort pero en lugar de dividir los valores en dos partes iguales, los divida en tres (asumir que se cuenta con la función `merge(lista_1, lista_2, lista_3)`). ¿Cómo te parece que se va a comportar este método con respecto al merge sort original?

Ejercicio 20.4.2. Mostrar los pasos del ordenamiento de la lista: `6 0 3 2 5 7 4 1` con los métodos de inserción y con merge sort. ¿Cuáles son las principales diferencias entre los métodos? ¿Cuál usarías en qué casos?

Ejercicio 20.4.3. Ordenar la lista `6 0 3 2 5 7 4 1` usando el método quicksort. Mostrar el árbol de recursividad explicando las llamadas que se hacen en cada paso, y el orden en el que se realizan.

Licencia y Copyright

Copyright © Rosita Wachenchauzer <rositaw@gmail.com>
Copyright © Margarita Manterola <margamanterola@gmail.com>
Copyright © Maximiliano Curia <maxy@gnuservers.com.ar>
Copyright © Marcos Medrano <mmedrano@fi.uba.ar>
Copyright © Nicolás Paez <nicopaez@computer.org>
Copyright © Diego Essaya <dessaya@gmail.com>
Copyright © Dato Simó <dato@net.com.org.es>
Copyright © Sebastián Santisi <s@ntisi.com.ar>

Esta obra se distribuye bajo la Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional.

Los íconos utilizados fueron diseñados por Freepik.

El logo de Python es una marca registrada de la Python Software Foundation.

La publicidad de Cacao Droste es de dominio público, y fue descargada de Wikipedia.