Algoritmos y Programación I
Con lenguaje Python

9 de marzo de 2011

Contenidos

1. Conceptos básicos **7**
1.1. Computadoras y programas 7
1.2. El mito de la máquina todopoderosa 8
1.3. Cómo darle instrucciones a la máquina usando Python 9
1.4. Devolver un resultado 13
1.5. Una instrucción un poco más compleja: el ciclo definido 13
1.5.1. Ayuda desde el intérprete 15
1.6. Construir programas y módulos 15
1.7. La forma de un programa Python 17
1.8. Estado y computación 18
1.9. Depuración de programas 19
1.10. Ejercicios 20

2. Programas sencillos **21**
2.1. Construcción de programas 21
2.2. Realizando un programa sencillo 22
2.3. Piezas de un programa Python 24
2.3.1. Nombres 24
2.3.2. Expresiones 25
2.4. No sólo de números viven los programas 26
2.5. Instrucciones 27
2.6. Ciclos definidos 27
2.7. Una guía para el diseño 28
2.8. Ejercicios 29

3. Funciones **30**
3.1. Documentación de funciones 30
3.2. Imprimir versus Devolver 31
3.3. Cómo usar una función en un programa 32
3.4. Más sobre los resultados de las funciones 34
3.5. Un ejemplo completo 35
3.6. Devolver múltiples resultados 38
3.7. Resumen 39

4. Decisiones **41**
4.1. Expresiones booleanas 41
4.1.1. Expresiones de comparación 42
4.1.2. Operadores lógicos 42

4.2. Comparaciones simples 43
4.3. Múltiples decisiones consecutivas 46
4.4. Ejercicios 48
4.5. Resumen 48

5. Más sobre ciclos 50
5.1. Ciclos indefinidos 51
5.2. Ciclo interactivo 51
5.3. Ciclo con centinela 53
5.4. Cómo romper un ciclo 54
5.5. Ejercicios 56
5.6. Resumen 57

6. Cadenas de caracteres 58
6.1. Operaciones con cadenas 58
6.1.1. Obtener el largo de una cadena 58
6.1.2. Recorrer una cadena 59
6.1.3. Acceder a una posición de la cadena 59
6.2. Segmentos de cadenas 60
6.3. Las cadenas son inmutables 61
6.4. Procesamiento sencillo de cadenas 62
6.5. Nuestro primer juego 65
6.6. Ejercicios 71
6.7. Resumen 71

7. Tuplas y listas 72
7.1. Tuplas 72
7.1.1. Elementos y segmentos de tuplas 72
7.1.2. Las tuplas son inmutables 73
7.1.3. Longitud de tuplas 73
7.1.4. Empaquetado y desempaquetado de tuplas 74
7.1.5. Ejercicios con tuplas 75
7.2. Listas 75
7.2.1. Longitud de la lista. Elementos y segmentos de listas 76
7.2.2. Cómo mutar listas 76
7.2.3. Cómo buscar dentro de las listas 77
7.3. Ordenar listas 83
7.4. Listas y cadenas 83
7.4.1. Ejercicios con listas y cadenas 84
7.5. Resumen 84

8. Algoritmos de búsqueda 87
8.1. El problema de la búsqueda 87
8.2. Cómo programar la búsqueda lineal a mano 88
8.3. Búsqueda lineal 88
8.4. Buscar sobre una lista ordenada 90
8.5. Búsqueda binaria 90
8.6. Resumen 94

9. Diccionarios **95**
9.1. Qué es un diccionario 95
9.2. Utilizando diccionarios en Python 96
9.3. Algunos usos de diccionarios 97
9.4. Resumen 98

10. Contratos y Mutabilidad **99**
10.1. Pre y Postcondiciones 99
10.1.1. Precondiciones 99
10.1.2. Postcondiciones 99
10.1.3. Aseveraciones 100
10.1.4. Ejemplos 100
10.2. Invariantes de ciclo 101
10.2.1. Comprobación de invariantes desde el código 102
10.3. Mutabilidad e Inmutabilidad 102
10.3.1. Parámetros mutables e inmutables 103
10.4. Resumen 104
10.5. Apéndice - Acertijo MU 105

11. Manejo de archivos **107**
11.1. Cerrar un archivo 107
11.2. Ejemplo de procesamiento de archivos 108
11.3. Modo de apertura de los archivos 109
11.4. Escribir en un archivo 109
11.5. Agregar información a un archivo 110
11.6. Manipular un archivo en forma binaria 112
11.7. Persistencia de datos 113
11.7.1. Persistencia en archivos CSV 115
11.7.2. Persistencia en archivos binarios 117
11.8. Directorios 118
11.9. Resumen 118
11.10Apéndice 121

12. Manejo de errores y excepciones **124**
12.1. Errores 124
12.2. Excepciones 124
12.2.1. Manejo de excepciones 125
12.2.2. Procesamiento y propagación de excepciones 127
12.2.3. Acceso a información de contexto 128
12.3. Validaciones 128
12.3.1. Comprobaciones por contenido 129
12.3.2. Entrada del usuario 129
12.3.3. Comprobaciones por tipo 130
12.3.4. Comprobaciones por características 132
12.4. Resumen 132
12.5. Apéndice 134

13. Procesamiento de archivos **135**
13.1. Corte de control . . . 135
13.2. Apareo . . . 137
13.3. Resumen . . . 138

14. Objetos **139**
14.1. Tipos . . . 139
14.2. Qué es un objeto . . . 140
14.3. Definiendo nuevos tipos . . . 141
14.3.1. Nuestra primera clase: Punto . . . 141
14.3.2. Agregando validaciones al constructor . . . 142
14.3.3. Agregando operaciones . . . 143
14.4. Métodos especiales . . . 145
14.4.1. Un método para mostrar objetos . . . 145
14.4.2. Métodos para operar matemáticamente . . . 145
14.5. Creando clases más complejas . . . 146
14.5.1. Métodos para comparar objetos . . . 148
14.5.2. Ordenar de menor a mayor listas de hoteles . . . 149
14.5.3. Otras formas de comparación . . . 150
14.5.4. Comparación sólo por igualdad o desigualdad . . . 151
14.6. Ejercicios . . . 152
14.7. Resumen . . . 152

15. Polimorfismo, Herencia y Delegación **154**
15.1. Polimorfismo . . . 154
15.1.1. Interfaz . . . 154
15.1.2. Redefinición de métodos . . . 155
15.1.3. Un ejemplo de polimorfismo . . . 155
15.2. Herencia . . . 157
15.3. Delegación . . . 159
15.4. Resumen . . . 162

16. Listas enlazadas **163**
16.1. Una clase sencilla de *vagones* . . . 163
16.1.1. Caminos . . . 164
16.1.2. Referenciando el principio de la lista . . . 165
16.2. Tipos abstractos de datos . . . 165
16.3. La clase `ListaEnlazada` . . . 166
16.3.1. Construcción de la lista . . . 167
16.3.2. Eliminar un elemento de una posición . . . 168
16.3.3. Eliminar un elemento por su valor . . . 169
16.3.4. Insertar nodos . . . 171
16.4. Invariantes de objetos . . . 175
16.5. Otras listas enlazadas . . . 175
16.6. Iteradores . . . 176
16.7. Resumen . . . 178

17. Pilas y colas **179**
17.1. Pilas . . . 179
17.1.1. Pilas representadas por listas . . . 179
17.1.2. Uso de pila: calculadora científica . . . 181
17.1.3. ¿Cuánto cuestan los métodos? . . . 186
17.2. Colas . . . 186
17.2.1. Colas implementadas sobre listas . . . 186
17.2.2. Colas y listas enlazadas . . . 188
17.3. Resumen . . . 190
17.4. Apéndice . . . 191

18. Modelo de ejecución de funciones y recursividad **194**
18.1. La pila de ejecución de las funciones . . . 194
18.2. Pasaje de parámetros . . . 196
18.3. Devolución de resultados . . . 198
18.4. La recursión y cómo puede ser que funcione . . . 199
18.5. Una función recursiva matemática . . . 199
18.6. Algoritmos recursivos y algoritmos iterativos . . . 202
18.7. Un ejemplo de recursividad elegante . . . 203
18.8. Un ejemplo de recursividad poco eficiente . . . 205
18.9. Limitaciones . . . 206
18.10Resumen . . . 207

19. Ordenar listas **208**
19.1. Ordenamiento por selección . . . 208
19.1.1. Invariante en el ordenamiento por selección . . . 209
19.1.2. ¿Cuánto cuesta ordenar por selección? . . . 211
19.2. Ordenamiento por inserción . . . 211
19.2.1. Invariante del ordenamiento por inserción . . . 214
19.2.2. ¿Cuánto cuesta ordenar por inserción? . . . 214
19.2.3. Inserción en una lista ordenada . . . 214
19.3. Resumen . . . 214

20. Algunos ordenamientos recursivos **216**
20.1. Ordenamiento por mezcla, o *Merge sort* . . . 216
20.2. ¿Cuánto cuesta el *Merge sort*? . . . 217
20.3. Ordenamiento rápido o *Quick sort* . . . 219
20.4. ¿Cuánto cuesta el *Quick sort*? . . . 220
20.5. Una versión mejorada de *Quick sort* . . . 222
20.6. Resumen . . . 222

A. Licencia y Copyright **224**

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 ...

Existen 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.

Todas estas aplicaciones son software libre, por lo que se puede obtener y estudiar el código con el que están hechas

1.2. El mito de la máquina todopoderosa

Muchas veces la gente se imagina que con la computadora se puede hacer cualquier cosa, que no hay tareas imposibles de realizar. Más aún, se imaginan que si bien hubo cosas que eran imposibles de realizar hace 50 años, ya no lo son más, o no lo serán dentro de algunos años, cuando las computadoras crezcan en poder (memoria, velocidad), y la computadora se vuelva una máquina todopoderosa.

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 $N \times N \times \ldots \times N$, que involucra 32 multiplicaciones.

Otra solución, mucho más eficiente es:

- Calcular $N \times N$.
- 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 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. Para orientarnos, el intérprete presenta una

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 versión 3.0, cuya versión actual es la 3.2, de febrero de 2011. Sin embargo, debido a que estas versiones introducen importantes cambios y no son totalmente compatibles con las versiones anteriores, todavía no se la utiliza extensamente.

línea de comandos (los comandos son las órdenes) que identifica al comienzo con los símbolos `>>>`, y que llamaremos *prompt*. En esta línea, a continuación del *prompt* podemos escribir diferentes órdenes.

Algunas órdenes sencillas, por ejemplo, permiten utilizar la línea de comandos como una calculadora simple con números enteros. Para esto escribimos la expresión que queremos resolver en el *prompt* y presionamos la tecla <ENTER>. El intérprete de Python "responde" el resultado de la operación en la línea siguiente, sin *prompt*, y luego nos presenta nuevamente el cursor para escribir la siguiente orden.

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

Python permite utilizar las operaciones `+`, `-`, `*`, `/` (división entera), y `**` (potenciación). La sintaxis es la convencional (valores intercalados con operaciones), y se pueden 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
>>> 10/5
2
>>> 5**2
25
>>>
```

Otra orden sencilla de Python permite indicarle al intérprete que escriba o "imprima" por pantalla una palabra o frase, que llamaremos *cadena de texto*.

```
>>> print 'Hola'
Hola
>>> print 'Como estan?'
Como estan?
>>> print "Bienvenidos y bienvenidas a este curso!"
Bienvenidos y bienvenidas  a este curso!
```

```
>>>
```

print es una instrucción de Python: aquélla que le indica a la máquina que debe imprimir un texto en pantalla, que deberá ser ingresado entre comillas simples `'` o dobles `"` indistintamente. Ya veremos con qué otras instrucciones viene equipado Python.

Pero ya dijimos 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 instrucciones que se le dan a la máquina, escritas por uno o muchos programadores.

Llamaremos *función* a una instrucción escrita por un programador.

Si queremos escribir una función (que llamaremos `holaMar`) que escribe en una línea el texto “Hola Marta!” y en la línea siguiente el texto “Estoy programando en Python.”, lo que debemos hacer es ingresar el siguiente conjunto de líneas en Python:

Código 1.1 `holaMar`: Saluda a Marta

```
>>> def holaMar():
        print "Hola Marta!"
        print "Estoy programando en Python."

>>>
```

def `holaMar():` le indica a Python que estamos escribiendo una instrucción cuyo nombre es `holaMar`. ¿Por qué se ponen esos dos paréntesis? Lo veremos dentro de unos párrafos. La sangría con la que se escriben las dos instrucciones **print** le indican a Python que estamos escribiendo el *cuerpo* (es decir las instrucciones que la componen) de la función en cuestión. Las dos teclas <ENTER> que tecleamos después de ingresar el texto `"Estoy programando en Python."` le indican a Python que se acabó el cuerpo de la función (y por eso aparece nuevamente el cursor).

Si ahora queremos que la máquina ejecute la instrucción `holaMar`, debemos escribir `holaMar()` a continuación del cursor de Python:

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

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

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

Código 1.2 `holaPab`: Saluda a Pablo

```
>>> def holaPab():
        print "Hola Pablo!"
        print "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 querramos. Para eso están precisamente los paréntesis.

Las funciones tienen partes variables, llamadas *parámetros*, que se ponen dentro de los paréntesis. Escribimos por ejemplo una función `hola` general que nos sirva para saludar a cualquiera, de la siguiente manera:

Código 1.3 `hola`**: Saluda a quien sea necesario**

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

En este caso, `alguien` es un parámetro cuyo valor será reemplazado por un texto (nombre en este caso) en cada invocación. 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 1.3.1. Escribir un programa que calcule el cuadrado de un número dado.

Solución. Para resolver este problema, se combinan los recursos utilizados hasta ahora.

Código 1.4 `cuad1`**: Eleva un número al cuadrado y lo imprime**

```
def cuad1(num):
        print num*num
```

Para invocarlo, deberemos hacer:

```
1 >>> cuad1(5)
2 25
3 >>>
```

Problema 1.3.2. Permitir que el usuario ingrese el valor a elevar al cuadrado.

Solución. Para esto utilizaremos una nueva función `input` que permite leer valores ingresados por el usuario.

La ejecución del programa será la siguiente:

```
>>> cuad2()
Ingrese un numero: 5
25
>>>
```

Código 1.5 `cuad2`**:** Pide un número al usuario e imprime su cuadrado

```
def cuad2():
        n = input("Ingrese un número: ")
        cuad1(n)
```

1.4. Devolver un resultado

Las funciones que vimos hasta ahora muestran mensajes, pero no hemos visto funciones que se comporten como las funciones que conocemos, las de la matemática, que se usan para calcular resultados.

Queremos también poder hacer cosas del estilo $y = f(x)$ en nuestros programas. Para ello introduciremos la instrucción **`return`** `<expresion>` que indica cuál es el valor que tiene que devolver nuestra función.

En este ejemplo escribimos una función que eleva al cuadrado un número.

Código 1.6 `cuadrado`**:** Eleva un número al cuadrado y lo devuelve

```
>>> def cuadrado (x):
...         cua = x * x
...         return cua
...
>>> y = cuadrado (5)
>>> y
25
>>>
```

1.5. Una instrucción un poco más compleja: el ciclo definido

Problema 1.5.1. Ahora que sabemos construir una función que calcula el cuadrado de un número, nos piden que imprimamos los cuadrados de los números del 2 al 8.

Solución. Por supuesto que podemos hacer:

```
>>> def cuad3():
        print cuadrado(2)
        print cuadrado(3)
        print cuadrado(4)
        print cuadrado(5)
        print cuadrado(6)
        print cuadrado(7)
        print cuadrado(8)

>>> cuad3()
```

```
4
9
16
25
36
49
64
>>>
```

¿Se puede hacer algo mejor que esto? Con lo que sabemos de Python hasta el momento, no.

Para resolver este tipo de problema (repetir un cálculo para los valores contenidos en un intervalo dado) de manera 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>*.

Vemos cómo, usando esta construcción, el problema anterior se puede resolver de manera más compacta:

Solución. En nuestro caso lo que hay que hacer es invocar a la instrucción `print cuadrado(x)` que usa la función definida en 1.6 para calcular el cuadrado de `x`, y luego imprime el resultado, cuando `x` toma los valores $2, 3, \ldots, 8$.

La instrucción que describe qué tipo de repeticiones se deben realizar es el *encabezado del ciclo*, y las instrucciones que describen la acción que se repite se llaman *cuerpo del ciclo*.

No nos olvidemos que en nuestro ejemplo el ciclo debe recorrer todos los valores enteros entre 2 y 8, por lo tanto:

- `for x in range(2, 9):` será el encabezado del ciclo y
- `print cuadrado(x)` será el cuerpo del ciclo.

Vemos entonces cómo resulta el ciclo completo (con su ejecución):

Atención

Todas las instrucciones que describen el cuerpo del ciclo deben tener una sangría mayor que el encabezado del ciclo.

Esta sangría puede ingresarse mediante espacios o tabuladores, pero es importante que sea la misma para todas las instrucciones del ciclo.

Código 1.7 `cuadrados`: Imprime los cuadrados del 2 al 8

```
>>> for x in range(2, 9):
        print cuadrado(x)

4
9
16
25
36
49
64
>>>
```

1.5.1. Ayuda desde el intérprete

El intérprete de python nos provee una ayuda en línea, es decir, nos puede dar la documentación de una función, intrucción, etc, para obtenerla llamamos a `help()`. Por ejemplo podemos pedir `help(input)` y nos dara la documentación de esa función.

Para obtener la documentación de las instrucciones las debemos poner entre comillas, es decir `help('return')`, de la misma forma se puede pedir ayuda sobre variables o valores.

Otra forma de obtener ayuda es mediante `dir(variable)`, que nos va a listar todas las funciones que tiene asociadas esa variable. Por ejemplo, mostramos las funciones asociadas a una cadena.

```
>>> dir("hola")
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Como se puede ver son muchas las funciones asociadas a las cadenas, pero no necesitamos conocerlas todas, a medida que avancemos veremos algunas.

1.6. Construir programas y módulos

El interprete es muy útil para probar cosas, acceder a la ayuda, inspeccionar el lenguaje, etc, pero si escribimos nuestras funciones frente al cursor de Python como hasta ahora, perdemos todas las definiciones cuando salimos de Python. Para conservar los programas que vamos escribiendo, debemos usar algún editor de texto, y guardar el archivo con la extensión `.py`.

Por convención, la primer línea del archivo deberá contener:

```
#!/usr/bin/env python
```

Estas dos convenciones indican que se trata de un *módulo* o *programa* Python, es decir un archivo separado que deberá ser ejecutado por Python.

Sabías que ...

Python es un lenguaje multiplataforma, esto quiere decir, que está pensado para que pueda utilizarse en una gran variedad de sistemas operativos (Windows, Mac, Linux, etc).
Las dos convenciones mencionadas le permiten a distintos sistemas "darse cuenta" que este archivo en particular lo tiene que procesar con Python. En particular, Windows, tradicionalmente, sólo se fija en la extensión del archivo para saber cómo procesarlo (por eso el `.py`), mientras que todos los sistemas derivados de Unix (OS X, Linux, Solaris, etc) analizan los permisos del archivo (en particular, el permiso de ejecución) y los primeros caracteres, para saber como procesarlos (por ello es necesario incluir esa primera línea mágica).

Problema 1.6.1. Escribir en Python un programa que haga lo siguiente:

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

Solución. Para la resolución del problema, escribiremos nuestro primer módulo de Python, que guardaremos en el archivo `cuad.py`, el código de este programa se encuentra en el Código 1.8.

Código 1.8 `cuad.py`**: Imprime los cuadrados solicitados**

```
1 #!/usr/bin/env python
2 """ Un programa sencillo, para calcular cuadrados de números """
3
4 def main():
5     print "Se calcularán cuadrados de números"
6
7     n1 = input("Ingrese un número entero: ")
8     n2 = input("Ingrese otro número entero: ")
9
10    for x in range(n1, n2):
11        print x*x
12
13    print "Es todo por ahora"
14
15 main()
```

Para ejecutar este módulo, podemos iniciar Python y luego importarlo. Lo ejecutaremos con valores 5 y 8 de la siguiente manera:

```
>>> import cuad
Se calcularán cuadrados de números
Ingrese un número entero: 5
Ingrese otro número entero: 8
25
36
49
Es todo por ahora
>>>
```

La orden **import** cuad le indica a Python que debe traer a la memoria el módulo `cuad.py`, tal como lo habíamos guardado, y ejecutar su contenido. Al hacer esto, sucenden las siguientes operaciones:

- Se carga en memoria la función `main` del módulo `cuad` (a la que se le asigna el nombre `cuad.main`), según la definición que está en el archivo, y
- se inicia su ejecución inmediatamente, dado que luego de la definición se encuentra la invocación `main()`.

Una vez importado el módulo, `cuad.main` queda en memoria, y se puede volver a invocar sin necesidad de importar nuevamente:

```
>>> cuad.main()
Se calcularán cuadrados de números
Ingrese un número entero: 3
Ingrese otro número entero: 5
9
16
Es todo por ahora
>>>
```

Por otro lado, habiendo cumplido con las convenciones nombradas anteriormente, es posible ejecutar el archivo como un programa normal, y el sistema se encargará de llamar a Python y darle nuestro archivo para que lo procese y ejecute.

1.7. La forma de un programa Python

La primera instrucción de `cuad.main` es

```
5    print "Se calcularán cuadrados de números"
```

que lo que hace es mostrar un mensaje por pantalla.

Las instrucciones segunda y tercera

```
7    n1 = input("Ingrese un número entero: ")
8    n2 = input("Ingrese otro número entero: ")
```

Sabías que . . .

En los programas Python que escribimos, podemos operar con cadenas de texto o con números. Las representaciones dentro de la computadora de un número y una cadena son muy distintas, el número `12345678` se almacena en forma binaria y utiliza unos pocos bytes, mientras que la cadena `"12345678"`, es una sucesión de caracteres, en la que cada número es un caracter que ocupa un byte.

La función `input` toma valores numéricos, y si se desea ingresar una cadena, debe hacérselo entre comillas: `"hola"`. Existe, por otro lado, la función `raw_input`, para la cual los valores ingresados son siempre cadenas de caracteres.

son instrucciones de entrada: se despliega el texto que está entre comillas y se espera que el usuario ingrese un valor numérico y oprima la tecla <ENTER>.

¿Cómo hacer para que los valores que provee el usuario se recuerden a lo largo de todo el programa? Al valor ingresado se le dará un nombre, de la misma manera que a otros valores calculados durante la ejecución. Aparece el concepto de *variables* de un programa: una variable se usa para darle un nombre a un valor dado y poder de esa manera referirnos al mismo a lo largo del programa.

En estas dos instrucciones, `n1` y `n2` son los nombres con los que se mencionarán el primer y el segundo entero tipeados por el usuario.

En el ejemplo de la última corrida, se asociará el valor 3 con la variable `n1` y el valor 5 con la variable `n2`.

Luego de leer esos valores, se procede a ejecutar el ciclo

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

Si el valor asociado con `n1` es 3, y el valor asociado con `n2` es 5, se asociará a `x` sucesivamente con los valores 3 y 4, y en cada caso se ejecutará el cuerpo del ciclo indicado (mostrará en pantalla los valores de los cuadrados de 3 y 4).

Finalmente, cuando se terminan las repeticiones indicadas en el ciclo, se ejecuta la instrucción

```
13         print "Es todo por ahora"
```

que, como ya se ha visto, muestra el mensaje `Es todo por ahora` por pantalla.

1.8. 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 $\texttt{n} \to 13$ 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 `cuad`:

Instrucción	Qué sucede	Estado
`print "Se calcularán cuadrados de números"`	Se despliega el texto "Se calcularán cuadrados de números" en la pantalla.	
`n1 = 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` $\rightarrow$ 3
`n2 = 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` $\rightarrow$ 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` $\rightarrow$ 3 `n2` $\rightarrow$ 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` $\rightarrow$ 3 `n2` $\rightarrow$ 5 `x` $\rightarrow$ 3
`print x*x`	Se imprime por pantalla el valor de `x` * `x` (9)	`n1` $\rightarrow$ 3 `n2` $\rightarrow$ 5 `x` $\rightarrow$ 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` $\rightarrow$ 3 `n2` $\rightarrow$ 5 `x` $\rightarrow$ 4
`print x*x`	Se imprime por pantalla el valor de `x` * `x` (16)	`n1` $\rightarrow$ 3 `n2` $\rightarrow$ 5 `x` $\rightarrow$ 4
`for x in range(n1, n2):`	Como no quedan más valores por tratar en `[n1,n2)`, se sale del ciclo.	`n1` $\rightarrow$ 3 `n2` $\rightarrow$ 5 `x` $\rightarrow$ 4
`print "Es todo por ahora"`	Se despliega por pantalla el mensaje "Es todo por ahora"	`n1` $\rightarrow$ 3 `n2` $\rightarrow$ 5 `x` $\rightarrow$ 4

1.9. 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.

1.10. Ejercicios

Ejercicio 1.1. Correr tres veces el programa `cuad` con valores de entrada (3,5), (3,3) y (5,3) respectivamente. ¿Qué sucede en cada caso?

Ejercicio 1.2. Insertar instrucciones de depuración que permitan ver el valor asociado con la variable `x` en el cuerpo del ciclo **`for`** y después que se sale de tal ciclo. Volver a correr tres veces el programa `cuad` con valores de entrada (3,5), (3,3) y (5,3) respectivamente, y explicar lo que sucede.

Ejercicio 1.3. La salida del programa `cuad` es poco informativa. Escribir un programa `nom_cuad` que ponga el número junto a su cuadrado. Ejecutar el programa nuevo.

Ejercicio 1.4. Si la salida sigue siendo poco informativa seguir mejorándola hasta que sea lo suficientemente clara.

Unidad 2

Programas sencillos

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

2.1. Construcción de programas

Cuando nos piden que hagamos un programa debemos seguir una cierta cantidad de pasos para asegurarnos de que tendremos éxito en la tarea. La acción irreflexiva (me piden algo, me siento frente a la computadora y escribo rápidamente y sin pensarlo 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, dado.

Existen muchas metodologías para construir programas, pero en este curso aplicaremos una metodología 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 el *depurador* 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     millas = input("Cuántas millas?: ")
4     pies = input("Y cuántos pies?: ")
5     pulgadas = input("Y cuántas pulgadas?: ")
6
7     metros = 1609.344 * millas + 0.3048 * pies + 0.0254 * pulgadas
8     print "La longitud es de ", metros, " metros"
9 main()
```

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

   - 1 milla, 0 pies, 0 pulgadas.
   - 0 millas, 1 pie, 0 pulgada.
   - 0 millas, 0 pies, 1 pulgada.

   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.

Atención

Al entregar un ejercicio, se deberá presentar el desarrollo completo con todas las etapas, desde el análisis hasta las pruebas (y el mantenimiento, si hubo cambios).

2.3. Piezas de un programa Python

Para poder empezar a programar en Python es necesario que 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*. La lista completa de las palabras reservadas de Python aparecen en el cuadro 2.1.

and	**del**	**from**	**not**	**while**
as	**elif**	**global**	**or**	with
assert	**else**	**if**	**pass**	yield
break	**except**	**import**	**print**	
class	**exec**	**in**	**raise**	
continue	**finally**	**is**	**return**	
def	**for**	**lambda**	**try**	

Cuadro 2.1: Las palabras reservadas de Python. Estas palabras no pueden ser usadas como identificadores.

2.3.2. Expresiones

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

- Un valor es una expresión (de hecho es la expresión más sencilla). Por ejemplo el resultado de la expresión `111` es precisamente el número `111`.
- Una variable es una expresión, y el valor que produce es el que tiene asociado en el estado (si `x` $\rightarrow$ 5 en el estado, entonces el resultado de la expresión `x` es el número 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` $\rightarrow$ 1, `pies` $\rightarrow$ 0 y `pulgadas` $\rightarrow$ 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)) /`
    - Igual que en la matemática, si no hay paréntesis en la expresión primero se agrupan las exponenciaciones, luego los productos y cocientes, y luego las sumas y restas.
    - Sin embargo, hay que tener cuidado con lo que sucede con los cocientes, porque si `x` e `y` son números enteros, entonces `x / y` se calcula como la *división entera* entre `x` e `y`:
      Si `x` se refiere al valor 12 e `y` se refiere al valor 9 entonces `x / y` se refiere al valor 1.
    - Si `x` e `y` son números enteros, entonces `x % y` se calcula como el *resto de la división entera* entre `x` e `y`:
      Si `x` se refiere al valor 12 e `y` se refiere al valor 9 entonces `x % y` se refiere al valor 3.

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.

- Conocemos también dos expresiones muy particulares:
  - `input`, que devuelve el valor ingresado por teclado tal como se lo digita (en particular sirve para ingresar valores numéricos).
  - `raw_input`, que devuelve lo ingresado por teclado como si fuera un texto.

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) `1 / 2 * b`

f) `b / 2`.

Ejercicio 2.2. En Python hagan lo siguiente: Denle a `a`, `b` y `c` los valores 10, 100 y 1000 respectivamente y evalúen las expresiones del ejercicio anterior.

Ejercicio 2.3. En Python hagan lo siguiente: Denle a `a`, `b` y `c` los valores 10.0, 100.0 y 1000.0 respectivamente y evalúen las expresiones del punto anterior.

2.4. No sólo de números viven los programas

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

```
def hola(alguien):
    print "Hola", alguien, "!"
    print "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"`.

Python usa también una notación con comillas simples para referirse a las cadenas de caracteres, y habla de `'Ana'`.

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

- Un valor también acá es una expresión. Por ejemplo el resultado de la expresión `'Ana'` es precisamente `'Ana'`.
- Una variable es una expresión, y el valor que produce es el que tiene asociado en el estado (si `amiga` → `'Ana'` en el estado, entonces el resultado de la expresión `amiga` es la cadena `'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 pueden sumar cadenas más 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 `*` se usa para indicar cuántas veces se repite una cadena: `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'
```

2.5. Instrucciones

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

- hemos mostrado valores por pantalla mediante la instrucción **`print`**,
- hemos retornado valores de una función mediante la instrucción **`return`**,
- hemos asociado valores con variables y
- hemos usado un ciclo para repetir un cálculo.

2.6. 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.

2.7. 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:

- ¿Han visto este problema antes, aunque sea de manera ligeramente diferente?
- ¿Conocen un problema relacionado? ¿Conocen un programa que puede ser útil?
- Fíjense en la especificación. Traten de encontrar un problema que les resulte familiar y que tenga la misma especificación o una parecida.
- Acá hay un problema relacionado con el que ustedes tienen y que ya fue resuelto. ¿Lo pueden usar? ¿Pueden usar sus resultados? ¿Pueden usar sus métodos? ¿Pueden agregarle alguna parte auxiliar a ese programa del que ya disponen?
- Si no pueden resolver el problema propuesto, traten de resolver uno relacionado. ¿Pueden imaginarse uno relacionado que sea más fácil de resolver? ¿Uno más general? ¿Uno más específico? ¿Un problema análogo?
- ¿Pueden resolver una parte del problema? ¿Pueden sacar algo útil de los datos de entrada? ¿Pueden pensar qué información es útil para calcular las salidas? ¿De qué manera se pueden manipular las entradas y las salidas de modo tal que estén "más cerca" unas de las otras?
- ¿Usaron todos los datos de entrada? ¿Usaron las condiciones especiales sobre los datos de entrada que aparecen en el enunciado? ¿Han tenido en cuenta todos los requisitos que se enuncian en la especificación?

2.8. Ejercicios

Ejercicio 2.4. Escribir un ciclo definido para imprimir por pantalla todos los números entre 10 y 20.

Ejercicio 2.5. Escribir un ciclo definido que salude por pantalla a sus cinco mejores amigos/as.

Ejercicio 2.6. Escribir un programa que use un ciclo definido con rango numérico, que pregunte los nombres de sus cinco mejores amigos/as, y los salude.

Ejercicio 2.7. Escribir un programa que use un ciclo definido con rango numérico, que pregunte los nombres de sus seis mejores amigos/as, y los salude.

Ejercicio 2.8. 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.

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 salude a todos a quienes queramos saludar:

```
def hola(alguien):
    print "Hola ", alguien,"!"
    print "Estoy programando en Python."
```

Dijimos en esa ocasión que las funciones tienen partes variables, llamadas parámetros, que se asocian a un valor distinto en cada invocación. El valor con el que se asocia un parámetro se llama *argumento*. En nuestro caso la invocamos 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:

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

Una función puede tener ninguno, uno o más parámetros. La función `hola` tiene un parámetro. Ya vimos también ejemplos de funciones sin parámetros:

```
def holaPab():
    print "Hola Pablo!"
    print "Estoy programando en Python."
```

En el caso de tener más de un parámetro, éstos se separan entre sí por comas, y en la invocación también se separan por comas los argumentos.

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 saber exactamente qué hace una 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):
    """ Imprime por pantalla un saludo, dirigido a la persona que
        se indica por parámetro. """
    print "Hola ", alguien,"!"
    print "Estoy programando en Python."
```

Cuando una función definida está correctamente documentada, es posible acceder a su documentación mediante la función `help` provista por Python:

```
>>> help(hola)
Help on function hola in module __main__:

hola(alguien)
    Imprime por pantalla un saludo, dirigido a la persona que
    se indica 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

A continuación se define una función `print_asegundos (horas, minutos, segundos)` con tres parámetros (`horas`, `minutos` y `segundos`) que imprime por pantalla la transformación a segundos de una medida de tiempo expresada en horas, minutos y segundos:

```
1 def print_asegundos (horas, minutos, segundos):
2     """ Transforma en segundos una medida de tiempo expresada en
3         horas, minutos y segundos """
4     segsal = 3600 * horas + 60 * minutos + segundos  # regla de transformación
5     print "Son", segsal, "segundos"
```

Para ver si realmente funciona, podemos ejecutar la función de la siguiente forma:

```
>>> print_asegundos (1, 10, 10)
Son 4210 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.

```
1 def calc_asegundos (horas, minutos, segundos):
2     """ Transforma en segundos una medida de tiempo expresada en
3         horas, minutos y segundos """
4     segsal = 3600 * horas + 60 * minutos + segundos  # regla de transformacion
5     return segsal
```

De esta forma, es posible realizar distintas operaciones con el valor obtenido luego de hacer la cuenta:

```
>>> print calc_asegundos (1, 10, 10)
4210
>>> print "Son", calc_asegundos (1, 10, 10), "segundos"
Son 4210 segundos
>>> y = calc_asegundos(1, 10, 10)
>>> z = calc_asegundos(2, 20, 20)
>>> y+z
12630
```

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 retorne 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` retorne 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.

Para utilizar las funciones definidas anteriormente, escribiremos un programa que pida tres duraciones, y en los tres casos las transforme a segundos y las muestra por pantalla.

1. **Análisis:** El programa debe pedir 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 3600 * h + 60 * m + 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 hs)

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

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

Mostrar por pantalla 3600 * hs + 60 * min + seg
```

  Pero convertir y mostrar por pantalla es exactamente lo que hace nuestra función `print_asegundos`, por lo que podemos hacer que el cuerpo del ciclo se diseñe como:

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

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

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

Invocar la función print_asegundos(hs, min, seg)
```

- El pseudocódigo final queda:

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

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

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

        Invocar la función print_asegundos(hs, min, seg)
```

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`.

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

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

```
1 def print_asegundos (horas, minutos, segundos):
2     """ Transforma en segundos una medida de tiempo expresada en
3         horas, minutos y segundos """
4     segsal = 3600 * horas + 60 * minutos + segundos
5     print "Son",segsal, "segundos"
6
7 def main():
8     """ Lee tres tiempos expresados en hs, min y seg, y usa
9         print_asegundos para mostrar en pantalla la conversión a
10        segundos """
11     for x in range(3):
12         hs = raw_input("Cuantas horas?: ")
13         min = raw_input("Cuantos minutos?: ")
14         seg = raw_input("Cuantos segundos?: ")
15         print_asegundos(hs, min, seg)
16
17 main()
```

```
>>> import tres_tiempos
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
>>>
```

Ejercicio 3.5. Resolver el problema anterior usando ahora la función `calc_asegundos`.

3.4. Más sobre los resultados de las funciones

Ya hemos visto cómo hacer para que las funciones que se comporten como las funciones que conocemos, las de la matemática, que se usan para calcular resultados.

Veremos ahora varias cuestiones a tener en cuenta al escribir funciones. Para ello volvemos a escribir una función que eleva al cuadrado un número.

```
>>> def cuadrado (x):
...         cua = x * x
...         return cua
```

```
...
>>> y = cuadrado (5)
>>> y
25
>>>
```

¿Por qué no usamos dentro del programa el valor `cua` calculado dentro de la función?

```
>>> def cuadrado (x):
...     cua = x * x
...     return cua
...
>>> cua
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cua' is not defined
>>>
```

Las variables y los parámetros que se declaran dentro de una función no existen fuera de ella, no se los conoce. Fuera de la función se puede ver sólo el valor que retorna y es por eso que es necesario introducir la instrucción **`return`**.

¿Para qué hay que introducir un **`return`** en la función? ¿No alcanza con el valor que se calcula dentro de la misma para que se considere que la función retorna un valor? En Python no alcanza (hay otros lenguajes en los que se considera que el último valor calculado en una función es el valor de retorno de la misma).

```
>>> def cuadrado (x):
...     cua = x * x
...
>>> y = cuadrado (5)
>>> y
>>>
```

Cuando se invoca la función `cuadrado` mediante la instrucción `y = cuadrado (5)` lo que sucede es lo siguiente:

- Se invoca a `cuadrado` con el argumento `5`, y se ejecuta el cuerpo de la función.
- El valor que devuelve la función se asocia con la variable `y`.

Es por eso que si la función no devuelve ningún valor, no queda ningún valor asociado a la variable `y`.

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 `f`).
   - ¿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 `f` 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:

$$3600 * h + 60 * m + s, f * (3600 * h + 60 * m + s).$$

3. **Diseño:**

   Siguiendo a Thompson, buscamos un programa que haga algo análogo, y vemos si se lo puede modificar para resolver nuestro problema. El programa `tres_tiempos` que hicimos anteriormente, se parece bastante a lo que necesitamos. Veamos las diferencias entre sus especificaciones.

tres_tiempos.py	tarifador.py
repetir 3 veces: <hacer cosas>	leer el valor de f leer el valor de n repetir n veces: <hacer cosas>
El cuerpo del ciclo: Leer el valor de hs Leer el valor de min Leer el valor de seg Invocar print_asegundos(hs, min, seg)	El cuerpo del ciclo: Leer el valor de hs Leer el valor de min Leer el valor de seg Asignar segcalc = asegundos(hs, min, seg) Calcular costo = segcalc * f Mostrar por pantalla segcalc y costo
print_asegundos (hs, min, seg): segsal = 3600*hs+60*min+seg print segsal	asegundos (hs, min, seg): segsal = 3600*hs+60*min+seg return segsal

En primer lugar se observa que el `tarifador` debe leer el valor de la tarifa (`f`) y que en `tres_tiempos` se conoce la cantidad de ternas (3), mientras que en `tarifador` la cantidad de ternas es un dato a ingresar.

Además, se puede ver que en el cuerpo del ciclo de `tres_tiempos`, se lee una terna y se llama a `print_asegundos` que calcula, imprime y no devuelve ningún valor. Si hiciéramos lo mismo en `tarifador`, no podríamos calcular el costo de la comunicación. Es por ello que en lugar de usar `print_asegundos` se utiliza la función `asegundos`, que calcula el valor transformado y lo devuelve en lugar de imprimirlo y en el cuerpo principal del programa se imprime el tiempo junto con el costo asociado.

4. **Implementación:** El siguiente es el programa resultante:

Código 3.2 `tarifador.py`**:** Factura el tiempo de uso de un teléfono

```
1  def main():
2      """ El usuario ingresa la tarifa por segundo, cuántas
3          comunicaciones se realizaron, y la duracion de cada
4          comunicación expresada en horas, minutos y segundos. Como
5          resultado se informa la duración en segundos de cada
6          comunicación y su costo. """
7
8      f = input("¿Cuánto cuesta 1 segundo de comunicacion?: ")
9      n = input("¿Cuántas comunicaciones hubo?: ")
10     for x in range(n):
11         hs = input("¿Cuántas horas?: ")
12         min = input("¿Cuántos minutos?: ")
13         seg = input("¿Cuántos segundos?: ")
14         segcalc = asegundos(hs, min, seg)
15         costo = segcalc * f
16         print "Duracion: ", segcalc, "segundos. Costo: ",costo, "$."
17
18 def asegundos (horas, minutos, segundos):
19            segsal = 3600 * horas + 60 * minutos + segundos
20            return segsal
21
22 main()
```

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)`. Ésta es la corrida:

```
>>> import tarifador
Cuanto cuesta 1 segundo de comunicacion?: 0.40
Cuantas comunicaciones hubo?: 3
Cuantas horas?: 1
Cuantos minutos?: 0
Cuantos segundos?: 0
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 3.6. Corregir el programa para que:

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

3.6. Devolver múltiples resultados

Ahora nos piden que escribamos una función que dada una duración en segundos sin fracciones (representada por un número entero) calcule la misma duración en horas, minutos y segundos.

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 `hs`, `min` y `seg`, lo que debemos retornar es la terna `(hs, min, seg)`:

```
1 def aHsMinSeg (x):
2     """ Dada una duración en segundos sin fracciones
3         (la función debe invocarse con números enteros)
4         se la convierte a horas, minutos y segundos """
5     hs = x / 3600
6     min = (x % 3600) / 60
7     seg = (x % 3600 ) % 60
8     return (hs, min, seg)
```

Esto es lo que sucede al invocar esta función:

```
>>> (h, m, s) = aHsMinSeg(3661)
>>> print "Son",h,"horas",m,"minutos",s,"segundos"
Son 1 horas 1 minutos 1 segundos
>>> (h, m, s) = aHsMinSeg(3661.0)  # aca violamos la especificacion
>>> print "Son",h,"horas",m,"minutos",s,"segundos" # y esto es lo que pasa:
Son 1.0169444444444444 horas 1.0166666666666666 minutos 1.0 segundos
>>>
```

Sabías que ...

Cuando la función debe retornar múltiples resultados se empaquetan todos juntos en una n-upla del tamaño adecuado.
Esta característica está presente en Python, Haskell, y algunos otros pocos lenguajes. En los lenguajes en los que esta característica no está presente, como C, Pascal, Java o PHP, 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:

```
>>> t=aHsMinSeg(3661)
>>> print t
(1, 1, 1)
>>>
```

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 retornan.

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

3.7. Resumen

- Una función puede tener ninguno, uno o más parámetros. En el caso de tener más de uno, se separan por comas tanto en la declaración de la función como en la invocación.
- 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.
- Las funciones pueden imprimir mensajes para comunicarlos al usuario, y/o devolver valores. Cuando una función realice un cálculo o una operación con sus parámetros, es

recomendable que devuelva el resultado en lugar de imprimirlo, permitiendo realizar otras operaciones con él.

- 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.
- Si una función no devuelve nada, por más que se la asigne a una variable, no quedará ningún valor asociado a esa variable.

Referencia del lenguaje 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 indentación 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 *e*mpaqueta en una tupla de valores. Esta tupla 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)
```

Unidad 4

Decisiones

Nos plantean el siguiente problema:

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

Es claro que la primera línea se puede traducir como

```
x = input("Ingrese un numero: ")
```

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* que 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 números y los textos que vimos hasta ahora, Python introduce las constantes `True` y `False` para representar los valores de verdad *verdadero* y *falso* respectivamente.

Vimos que una expresión es un trozo de código Python que produce o calcula un valor (resultado). Una *expresión booleana* o *expresión lógica* es una expresión que vale o bien `True` o bien `False`.

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
>>>
```

Los 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` de lo contrario el resultado es `False`
`not a`	El resultado es `True` si `a` es `False` de lo contrario el resultado es `False`

- `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 `"Numero positivo"`.

Utilizando 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>` 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 `es_positivo()` que hace lo pedido:

```
def es_positivo():
    x = input("Ingrese un numero: ")
    if x > 0:
        print "Numero positivo"
```

y la probamos:

```
>>> es_positivo()
Ingrese un numero: 4
Numero positivo
>>> es_positivo()
Ingrese un numero: -25
>>> es_positivo()
Ingrese un numero: 0
>>>
```

Problema 4.2. En la etapa de mantenimiento nos dicen que, en realidad, también se necesitaría un mensaje `"Numero 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 `"Numero positivo"`
3. Si no se cumple $x > 0$, imprimir `"Numero 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 = input("Ingrese un numero: ")
    if x > 0:
       print "Numero positivo"
    if not (x > 0):
       print "Numero no positivo"
```

Probamos la nueva solución y obtenemos el resultado buscado:

```
>>> positivo_o_no()
Ingrese un numero: 4
Numero positivo
>>> positivo_o_no()
Ingrese un numero: -25
Numero no positivo
>>> positivo_o_no()
Ingrese un numero: 0
Numero no positivo
>>>
```

Sin embargo hay algo que nos preocupa: si ya averiguamos una vez, en la segunda línea del cuerpo, si `x > 0`, ¿Es realmente necesario volver a preguntarlo en la cuarta?.

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` (verdadero) se ejecutan las acciones indicadas como *<hacer algo si se da la condición>*, y si el resultado es `False` (falso) 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 `"Numero positivo"`

3. De lo contrario, imprimir `"Numero no positivo"`

Este diseño se implementa como:

```
def positivo_o_no_nue():
    x = input("Ingrese un numero: ")
    if x > 0:
       print "Numero positivo"
    else:
       print "Numero no positivo"
```

y lo probamos:

```
>>> positivo_o_no_nue()
Ingrese un numero: 4
Numero positivo
>>> positivo_o_no_nue()
Ingrese un numero: -25
Numero no positivo
>>> positivo_o_no_nue()
Ingrese un numero: 0
Numero no positivo
>>>
```

Es importante destacar que, en general, negar la condición del **`if`** y poner **`else`** no son intercambiables, 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 pn():
...     x = input("Ingrese un numero: ")
...     if x > 0:
...        print "Numero positivo"
...        x = -x
...     if x < 0:
...        print "Numero no positivo"
...
>>> pn()
Ingrese un numero: 25
Numero positivo
Numero no positivo
>>>
```

```
>>> def pn1():
...     x = input("Ingrese un numero: ")
...     if x > 0:
...        print "Numero positivo"
...        x = -x
...     else:
...        print "Numero no positivo"
...
>>> pn1()
Ingrese un numero: 25
Numero positivo
>>>
```

4.3. Múltiples decisiones consecutivas

La decisión de incluir una alternativa 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 `"Numero positivo"`, de lo contrario escribir un mensaje `"Numero 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 `"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"`.

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 `"Numero positivo"`
3. De lo contrario:
   a) Si $x = 0$, imprimir `"Igual a 0"`
   b) De lo contrario, imprimir `"Numero no positivo"`

Este diseño se implementa como:

```
def pcn1():
    x = input("Ingrese un numero: ")
    if x > 0:
       print "Numero positivo"
    else:
       if x == 0:
          print "Igual a 0"
       else:
          print "Numero 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 pcn2():
    x = input("Ingrese un numero: ")
    if x > 0:
       print "Numero positivo"
    elif x == 0:
       print "Igual a 0"
    else:
       print "Numero 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 0 o vacíos se consideran *falsos*.

Así, el ejemplo anterior también podría escribirse de la siguiente manera:

```
def pcn2():
    x = input("Ingrese un numero: ")
    if x > 0:
       print "Numero positivo"
    elif not x:
       print "Igual a 0"
    else:
       print "Numero negativo"
```

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 del lenguaje Python

if <condición>:

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

```
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 indentación 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 indentación 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 de lo contrario el resultado es False
not a	El resultado es True si a es False de lo contrario el resultado es False

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:

```
1  def muchos_pcn():
2      i = input("Cuantos numeros quiere procesar: ")
3      for j in range(0,i):
4          x = input("Ingrese un numero: ")
5          if x > 0:
6              print "Numero positivo"
7          elif x == 0:
8              print "Igual a 0"
9          else:
10             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 al usuario considera que este programa no es muy intuitivo, porque lo obliga 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, la condición es una expresión booleana, igual que en las instrucciones **if**. Y el cuerpo es, como siempre, una o más instrucciones de Python.

El sentido 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 `hayMasDatos`, 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:

```
1 def pcn_loop():
2     while hayMasDatos == "Si":
3         x = input("Ingrese un numero: ")
4         if x > 0:
5             print "Numero positivo"
6         elif x == 0:
7             print "Igual a 0"
8         else:
9             print "Numero negativo"
10
11        hayMasDatos = raw_input("¿Quiere seguir? <Si-No>: ")
```

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

```
>>> pcn_loop()

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

El problema que se presentó en este caso, es que `hayMasDatos` 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: al menos tiene que tener algún valor la expresión booleana que lo controla.

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 `hayMasDatos`.

Acá encararemos la segunda posibilidad:

```
1  def pcn_loop():
2      hayMasDatos = "Si"
3      while hayMasDatos == "Si":
4          x = input("Ingrese un numero: ")
5          if x > 0:
6              print "Numero positivo"
7          elif x == 0:
8              print "Igual a 0"
9          else:
10             print "Numero negativo"
11
12         hayMasDatos = raw_input("Quiere seguir? <Si-No>: ")
```

El esquema del ciclo interactivo es el siguiente:

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

Ésta es una ejecución:

```
>>> pcn_loop()
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. Se puede usar el método del *centinela*: un valor distinguido que, si se lee, le indica al programa que el usuario desea salir del ciclo. En este caso, podemos suponer que si 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.

En nuestro caso, pedir datos corresponde a lo siguiente:

- Pedir número.

El programa resultante es el siguiente:

```
1 def pcn_loop2():
2     x=input("Ingrese un numero ('*' para terminar): ")
3
4     while x <>"*":
5         if x > 0:
6             print "Numero positivo"
7         elif x == 0:
8             print "Igual a 0"
9         else:
10            print "Numero negativo"
11
12        x=input("Ingrese un numero ('*' para terminar): ")
```

Y ahora lo ejecutamos:

```
>>> pcn_loop2()
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): '*'
>>>
```

5.4. Cómo romper un ciclo

El ciclo con centinela es muy claro pero tiene un problema: hay dos lugares (la primera línea del cuerpo y la última línea del ciclo) donde se ingresa el mismo dato. Si en la etapa de mantenimiento tuviéramos que realizar un cambio en el ingreso del dato (cambio de mensaje, por ejemplo) deberíamos estar atentos y hacer dos correcciones iguales.

Sería preferible poder leer el dato `x` en un único punto del programa. A continuación, tratamos de diseñar una solución con esa restricción.

Es claro que en ese caso la lectura tiene que estar dentro del ciclo para poder leer más de un número, pero entonces la condición del ciclo no puede depender del valor leído, ni tampoco de valores calculados dentro del ciclo.

Pero 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>:
    <hacer algo_1>
    if <condif>:
        break
    <hacer algo_2>
```

el sentido del **`break`** es el siguiente:

1. Se evalúa <*condición*> y si es falsa se sale del ciclo.
2. Se ejecuta <*hacer* $algo_1$ >.
3. Se evalúa <*condif*> y si es verdadera se sale del ciclo (con **`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:

```
1 def pcn_loop3():
2     while True:
3         x = input("Ingrese un numero ('*' para terminar): ")
4         if x == '*':
5             break
6         elif x > 0:
7             print "Numero positivo"
8         elif x == 0:
9             print "Igual a 0"
10        else:
11            print "Numero negativo"
```

Y la probamos:

```
>>> pcn_loop3()
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): '*'
>>>
```

Sabías que ...

Desde hace mucho tiempo los ciclos infinitos vienen trayéndoles dolores de cabeza a los programadores. Cuando un programa deja de responder y se queda utilizando todo el procesador de la computadora, suele deberse a que el programa entró en un ciclo del que no puede salir.
Estos ciclos 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 telefono. Para ello se informa la tarifa por segundo y la duracion de cada comunicacion expresada en horas, minutos y segundos. Como resultado se informa la duracion en segundos de cada comunicacion y su costo. Resolver este problema usando

1. Ciclo definido.
2. Ciclo interactivo.
3. Ciclo con centinela.
4. Ciclo "infinito" que se rompe.

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. Ciclo interactivo.
2. Ciclo con centinela.
3. Ciclo "infinito" que se rompe.

¿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 utiliza 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 del lenguaje 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.

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.

6.1. Operaciones con cadenas

Ya vimos en la sección 2.4 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 el largo de una cadena

Además, se puede averiguar la longitud de 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 x in "programas ":
...     print x
...
p
r
o
g
r
a
m
a
s

>>>
```

6.1.3. 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 `a[i]` para hablar de la posición i-ésima de la cadena `a`.

Trataremos de averiguar con qué letra empieza una cadena.

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

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

```
>>> a[0]
'V'
>>>
```

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

Algunos ejemplos de acceso a distintas posiciones en una cadena.

Atención

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

Los índices entre `0` y `len(a) - 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: `a[-1]` es el último carácter de `a`, `a[-2]` es el penúltimo carácter de `a`, `a[-len(a)]` es el primer carácter de `a`.

```
>>> a="Veronica"
>>> len(a)
8
>>> a[0]
'V'
>>> a[7]
'a'
>>> a[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> a[-1]
'a'
>>> a[-8]
'V'
>>> a[-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: `a[0:2]` se refiere a la subcadena formada por los caracteres cuyos índices están en el rango `[0,2)`:

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

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

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

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

```
>>> a[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 `a[:]`.

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

6.3. Las cadenas son inmutables

Nos dicen que la persona sobre la que estamos hablando en realidad se llama "Veronika"(sí, con "k"). Como conocemos la notación de corchetes, tratamos de corregir sólo el carácter correspondiente de la variable `a`:

```
>>> a[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 cadena `a` se refiera a otro valor:

```
>>> a="Veronika"
>>> a
'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 `x`.

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

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 `x`, o sea que estamos frente a un esquema:

   para cada letra de `x`
       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 `x`. 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 `x`
       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 `x`
       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 muy poderoso para recorrer una cadena: una instrucción **for** que nos brinda un carácter por vez, del primero al último.

   Proponemos la siguiente solución:

```
1 def contarA(x):
2     for letra in x:
3         if letra == "A":
4             contador = contador + 1
5     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 `x`
       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 `x`. 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 `x`.

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

   El valor inicial que debemos darle a `contador` debe reflejar la cantidad de "A" que contiene la parte izquierda de `x` cuando aún no iniciamos el recorrido, es decir cuando esta

parte izquierda es `x[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 `x`
    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)**

```
1 def contarA (x):
2     """ La funcion contarA(x) cuenta cuántas
3         letras "A" aparecen en la cadena x ."""
4     contador = 0
5     for letra in x:
6         if letra == "A":
7            contador = contador + 1
8     return(contador)
```

6. Prueba

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

7. **Mantenimiento:**

Esta función resulta un poco limitada. Cuando nos pidan que contemos 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(l, x)` que cuente cuántas veces aparece un carácter `l` dado en una cadena `x`.

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. 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 devuelve un elemento al azar de una n-upla, y toma como parámetro la n-upla de la que tiene que elegir. Vamos a usarla entonces para elegir cifras. Para eso tenemos que construir una n-upla que tenga todas las cifras, lo hacemos de la misma manera que en la parte 3.6:

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

```
digitos = ('0','1','2','3','4','5','6','7','8','9')
```

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:

a) Tomar una cadena vacía

b) 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 3*b*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, el programa quedaría más o menos así:

```
1 # ARCHIVO: mastermind.py
2
3 # modulo que va a permitir elegir numeros aleatoriamente
4 import random
5
6 # el conjunto de simbolos validos en el codigo
7 digitos = ('0','1','2','3','4','5','6','7','8','9')
8
9 # "elegimos" el codigo
10 codigo = ''
11 for i in range(4):
12         candidato = random.choice(digitos)
13         # vamos eligiendo digitos no repetidos
14         while candidato in codigo:
15                 candidato = random.choice(digitos)
16         codigo = codigo + candidato
```

```
17
18 # iniciamos interaccion con el usuario
19 print "Bienvenido/a al Mastermind!"
20 print "Tenes que adivinar un numero de", 4, "cifras distintas"
21 propuesta = raw_input("Que codigo propones?: ")
22
23 # procesamos las propuestas e indicamos aciertos y coincidencias
24 intentos = 1
25 while propuesta != codigo:
26         intentos = intentos + 1
27         aciertos = 0
28         coincidencias = 0
29
30         # recorremos la propuesta y verificamos en el codigo
31         for i in range(4):
32                 if propuesta[i] == codigo[i]:
33                         aciertos = aciertos + 1
34                 elif propuesta[i] in codigo:
35                         coincidencias = coincidencias + 1
36         print "Tu propuesta (", propuesta, ") tiene", aciertos, \
37                 "aciertos y ", coincidencias, "coincidencias."
38         # pedimos siguiente propuesta
39         propuesta = raw_input("Propone otro codigo: ")
40
41 print "Felicitaciones! Adivinaste el codigo en", intentos, "intentos."
```

Cuando lo que queremos escribir es demasiado largo como para una sola línea que entre cómodamente en el editor o en el campo visual, le indicamos al intérprete que queremos seguir en la siguiente línea por medio de la barra invertida (como al final de la línea 36).

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

```
jugador@casino:~$ 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.
```

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 enlas pruebas.

```
1  # "elegimos" el codigo
2  codigo = ''
3  for i in range(4):
4          candidato = random.choice(digitos)
5          # vamos eligiendo digitos no repetidos
6          while candidato in codigo:
7                  print 'DEBUG: candidato =', candidato
8                  candidato = random.choice(digitos)
9          codigo = codigo + candidato
10         print 'DEBUG: el codigo va siendo =', 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:

```
jugador@casino:~$ python master_debug.py
DEBUG: el codigo va siendo = 8
DEBUG: candidato = 8
DEBUG: el codigo va siendo = 81
DEBUG: candidato = 1
DEBUG: el codigo va siendo = 814
DEBUG: el codigo va siendo = 8145
Bienvenido/a al Mastermind!
Tenes que adivinar un numero de 4 cifras distintas
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 11 del programa por un 5, indicando que hay que elegir 5 dígitos al azar. Pero además, el ciclo en la línea 31 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 20.

   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 y cambiarla sólo ahí:

```
1 # "elegimos" el codigo
2 cant_digitos = 5
3 codigo = ''
4 for i in range(cant_digitos):
5         candidato = random.choice(digitos)
6         # vamos eligiendo digitos no repetidos
```

El mensaje al usuario queda entonces:

```
1 # iniciamos interaccion con el usuario
2 print "Bienvenido/a al Mastermind!"
3 print "Tenes que adivinar un numero de", cant_digitos, \
4         "cifras distintas"
```

Y el chequeo de aciertos y coincidencias:

```
1         # recorremos la propuesta y verificamos en el codigo
2         for i in range(cant_digitos):
3                 if propuesta[i] == codigo[i]:
```

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"`.

```
1 # procesamos las propuestas e indicamos aciertos y coincidencias
2 intentos = 1
3 while propuesta != codigo and propuesta != "Me doy":
4         intentos = intentos + 1
5         aciertos = 0
```

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:

```
1 if propuesta == "Me doy":
2         print "El codigo era", codigo
3         print "Suerte la proxima vez!"
4 else:
5         print "Felicitaciones! Adivinaste el codigo en", \
6                 intentos, "intentos."
```

El código de todo el programa queda entonces así:

```
# ARCHIVO: mastermind5.py

# modulo que va a permitir elegir numeros aleatoriamente
import random

# el conjunto de simbolos validos en el codigo
digitos = ('0','1','2','3','4','5','6','7','8','9')

# "elegimos" el codigo
cant_digitos = 5
codigo = ''
for i in range(cant_digitos):
        candidato = random.choice(digitos)
        # vamos eligiendo digitos no repetidos
        while candidato in codigo:
                candidato = random.choice(digitos)
        codigo = codigo + candidato

# iniciamos interaccion con el usuario
print "Bienvenido/a al Mastermind!"
print "Tenes que adivinar un numero de", cant_digitos, \
        "cifras distintas"
propuesta = raw_input("Que codigo propones?: ")

# procesamos las propuestas e indicamos aciertos y coincidencias
intentos = 1
while propuesta != codigo and propuesta != "Me doy":
        intentos = intentos + 1
        aciertos = 0
        coincidencias = 0

        # recorremos la propuesta y verificamos en el codigo
        for i in range(cant_digitos):
                if propuesta[i] == codigo[i]:
                        aciertos = aciertos + 1
                elif propuesta[i] in codigo:
                        coincidencias = coincidencias + 1
        print "Tu propuesta (", propuesta, ") tiene", aciertos, \
                "aciertos y ", coincidencias, "coincidencias."
        # pedimos siguiente propuesta
        propuesta = raw_input("Propone otro codigo: ")
if propuesta == "Me doy":
        print "El codigo era", codigo
        print "Suerte la proxima vez!"
else:
        print "Felicitaciones! Adivinaste el codigo en", \
                intentos, "intentos."
```

6.6. Ejercicios

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!

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 del lenguaje Python

`len(cadena)`

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

`for letra in cadena`

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

`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.

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 querramos 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))`.

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 `(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 `(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
```

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 `sumaTiempo` que reciba dos tiempos dados y devuelva su suma.

Ejercicio 7.5. Escribir una función `diaSiguienteE` 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 `diaSiguienteT` 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: i `[78455, 89211, 66540, 45750]`. Al abrirse la inscripción, antes de que hubiera inscriptos, la lista de inscriptos se representará por una lista vacía: `[]`.

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.

```
>>> xs=[78455, 89211, 66540, 45750]
>>> xs[0]
78455
>>> len(xs)
4
>>> xs[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> xs[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:

```
>>> xs[1:2]
[89211]
```

  Para obtener la lista que contiene al segundo y tercer inscriptos podemos escribir:

```
>>> xs[1:3]
[89211, 66540]
```

  Para obtener la lista que contiene al primero y segundo inscriptos podemos escribir:

```
>>> xs[: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:

```
>>> xs[1]=79211
>>> xs
[78455, 79211, 66540, 45750]
```

- Para agregar un nuevo valor al final de la lista se utiliza la operación `append()`. Escribimos `xs.append(47890)` para agregar el padrón 47890 al final de `xs`.

```
>>> xs.append(47890)
>>> xs
[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 `xs.insert(2, 54988)` para insertar el padrón 54988 en la tercera posición de `xs`.

  ```
  >>> xs.insert(2, 54988)
  >>> xs
  [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.

  ```
  >>> xs.insert(1,78455)
  >>> xs
  [78455, 78455, 79211, 54988, 66540, 45750, 47890]
  ```

- Para eliminar un valor de una lista se utiliza la operación `remove()`.

  Escribimos `xs.remove(45750)` para borrar el padrón 45750 de la lista de inscriptos:

  ```
  >>> xs.remove(45750)
  >>> xs
  [78455, 78455, 79211, 54988, 66540, 47890]
  ```

  Si el valor a borrar está repetido, se borra sólo su primera aparición:

  ```
  >>> xs.remove(78455)
  >>> xs
  [78455, 79211, 54988, 66540, 47890]
  ```

Atención

Si el valor a borrar no existe, se produce un error:

```
>>> xs.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**:

```
>>> xs
[78455, 79211, 54988, 66540, 47890]
>>> 78 in xs
False
>>> 66540 in xs
True
>>>
```

Esta operación 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()`.

```
>>> xs.index(78455)
0
>>> xs.index(47890)
4
```

Atención

Si el valor no se encuentra en la lista, se producirá un error:

```
>>> xs.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:

```
>>> ys=[10,20,10]
>>> ys.index(10)
0
```

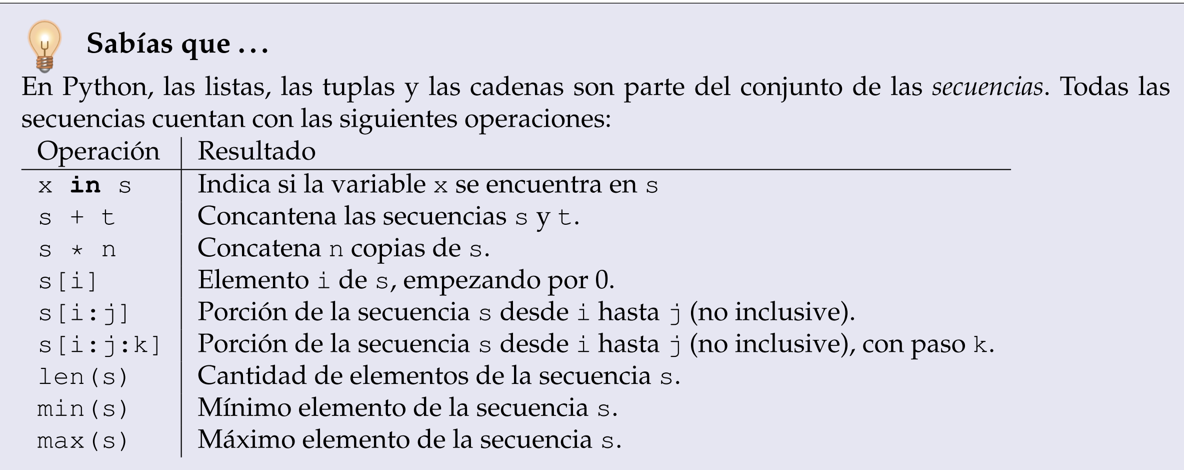
Esta operación está disponible en cadenas, pero no en tuplas.

- Para iterar sobre todos los elementos de una lista usaremos una construcción **for**:

```
>>> zs = [5, 3, 8, 10, 2]
>>> for x in zs:
...     print x
...
5
3
```

```
8
10
2
```

Esta construcción se puede utilizar sobre cualquier secuencia, incluyendo tuplas y cadenas.



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 la variable `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`.

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 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 distinguido (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:

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

   ◦ Pedir datos.

   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.
   - Pedir padrón.
   - Mientras el padrón sea positivo:
     - ◦ Agregar el padrón a la lista.
     - ◦ Pedir padrón.

4. **Implementación:** De acuerdo a lo diseñado en el párrafo anterior, el programa quedaría como se muestra en el Código 7.1.

Código 7.1 `inscriptos.py`**:** Permite ingresar padrones de alumnos inscriptos

```
1  #!/usr/bin/env python
2  # encoding: latin1
3  """ Módulo para inscribir alumnos al curso - versión 0 """
4
5  # Iniciamos la interacción con el usuario
6  print "Inscripcion en el curso 04 de 75.40"
7
8  # Leemos el primer padrón
9  padron=input("Ingresa un padrón (<=0 para terminar): ")
10
11 # Procesamos los padrones
12 # Inicialmente no hay inscriptos
13 ins = []
14 while padron > 0:
15        # Agregamos el padrón leído a la lista de inscriptos
16        ins.append(padron)
17
18        # Leemos otro padrón más
19        padron=input("Ingresá un padrón (<=0 para terminar): ")
20
21 # Mostramos el resultado
22 print "Esta es la lista de inscriptos: ", ins
```

5. **Prueba:** Para probarlo lo ejecutamos con algunos lotes de prueba (inscripción de tres alumnos, inscripción de cero alumnos, inscripción de alumnos repetidos):

```
Inscripción en el curso 04 de 75.40
Ingresá un padrón (<=0 para terminar): 30
Ingresá un padrón (<=0 para terminar): 40
Ingresá un padrón (<=0 para terminar): 50
```

```
Ingresá un padrón (<=0 para terminar): 0
Esta es la lista de inscriptos:  [30, 40, 50]
>>>
Inscripción en el curso 04 de 75.40
Ingresá un padrón (<=0 para terminar): 0
Esta es la lista de inscriptos:  []
>>>
Inscripción en el curso 04 de 75.40
Ingresá un padrón (<=0 para terminar): 30
Ingresá un padrón (<=0 para terminar): 40
Ingresá un padrón (<=0 para terminar): 40
Ingresá un padrón (<=0 para terminar): 30
Ingresá un padrón (<=0 para terminar): 50
Ingresá un padrón (<=0 para terminar): 0
Esta es la lista de inscriptos:  [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.
   - Pedir padrón.
   - Mientras el padrón sea positivo:
     - Si el padrón no está en la lista:
       - Agregar el padrón a la lista
     - pero si está en la lista:
       - Avisar que el padrón ya está en la lista
     - Pedir padrón.

8. **Nueva implementación:** De acuerdo a lo diseñado en el párrafo anterior, el programa ahora quedaría como se muestra en el Código 7.2.

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):

```
Inscripción en el curso 04 de 75.40
Ingresá un padrón (<=0 para terminar): 30
Ingresá un padrón (<=0 para terminar): 40
Ingresá un padrón (<=0 para terminar): 50
Ingresá un padrón (<=0 para terminar): 0
Esta es la lista de inscriptos:  [30, 40, 50]
>>>
Inscripción en el curso 04 de 75.40
Ingresá un padrón (<=0 para terminar): 0
```

Código 7.2 `inscriptos.py`: Permite ingresar padrones, sin repetir

```
1  #!/usr/bin/env python
2  # encoding: latin1
3  """ Módulo para inscribir alumnos al curso - versión 1 """
4
5  # Iniciamos la interacción con el usuario
6  print "Inscripcion en el curso 04 de 75.40"
7
8  # Leemos el primer padrón
9  padron=input("Ingresa un padrón (<=0 para terminar): ")
10
11 # Procesamos los padrones
12 # Inicialmente no hay inscriptos
13 ins = []
14 while padron > 0:
15         # Si todavía no está, agregamos el padrón a la lista de inscriptos,
16         if padron not in ins:
17                 ins.append(padron)
18         # de lo contrario avisamos que ya figura
19         else:
20                 print "Ya figura en la lista"
21
22         # Leemos otro padrón mas
23         padron=input("Ingresá un padrón (<=0 para terminar): ")
24
25 # Mostramos el resultado
26 print "Esta es la lista de inscriptos: ", ins
```

```
Esta es la lista de inscriptos:  []
>>>
Inscripción en el curso 04 de 75.40
Ingresá un padrón (<=0 para terminar): 30
Ingresá un padrón (<=0 para terminar): 40
Ingresá un padrón (<=0 para terminar): 40
Ya figura en la lista
Ingresá un padrón (<=0 para terminar): 30
Ya figura en la lista
Ingresá un padrón (<=0 para terminar): 50
Ingresá un padrón (<=0 para terminar): 0
Esta es la lista de inscriptos:  [30, 40, 50]
```

Pero 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 `xs` escribiremos simplemente `xs.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.

Referencia del lenguaje Python

`(valor1, valor2, valor3)`

Las tuplas se definen como una sucesión de valores encerrados entre paréntesis. 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 suceción de valores encerrados entre corchetes. 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 = tupla`

Para *desempaquetar* una secuencia, es posible asignar la variable que contiene la tupla a tantas variables como elementos tenga, cada variable tomará el valor del elemento que se encuentra en la misma posición.

`len(secuencia)`

Devuelve el largo de la secuencia, 0 si está vacía.

`for elemento in secuencia:`

Itera uno a uno por los elementos de la secuencia.

`if 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 (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)`
: Borra la primera aparción de elemento, si se encuentra en la lista. De no encontrarse en la lista, 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.

Unidad 8

Algoritmos de búsqueda

8.1. 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 muestra en el Código 8.1.

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
```

Código 8.1 `busqueda_con_index.py`: Busca utilizando index e in provistos por Python

```
1  #!/usr/bin/env python
2  # encoding: latin1
3
4  def busqueda_con_index(xs, x):
5      """Busca un elemento x en una lista xs
6
7      si x está en xs devuelve xs.index(x)
8      de lo contrario devuelve -1
9      """
10
11     if x in xs:
12         return(xs.index(x))
13     else:
14         return(-1)
```

```
>>> 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?

La pregunta del título se refiere a ¿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 funciones **`in`** e `index()`. La pregunta queda planteada por ahora pero daremos un método para averiguarlo más adelante en esta unidad.

8.2. Cómo programar la búsqueda lineal a mano

No 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 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.

8.3. Búsqueda lineal

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 en el Código 8.2.

Código 8.2 `busqueda_lineal.py`: Función de búsqueda lineal

```
1 #!/usr/bin/env python
2 # encoding: latin1
3
4 def busqueda_lineal(lista, x):
5     """ Búsqueda lineal.
6         Si x está en lista devuelve su posición en lista, de lo
7     contrario devuelve -1.
8     """
9
10    # Estrategia: se recorren uno a uno los elementos de la lista
11    # y se los compara con el valor x buscado.
12
13    i=0 # i tiene la posición actual en la lista, comienza en 0
14
15    # el ciclo for recorre todos los elementos de lista:
16    for z in lista:
17        # estamos en la posicion i, z contiene el valor de lista[i]
18
19        # si z es igual a x, devuelve i
20        if z == x:
21            return i
22
23        # si z es distinto de x, incrementa i, y continúa el ciclo
24        i=i+1
25
26    # si salió del ciclo sin haber encontrado el valor, devuelve -1
27    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([], 0)
-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 texto de `busqueda_lineal`:

- La **línea 16** del código es un ciclo que recorre uno a uno los elementos de la lista, y en el cuerpo de ese ciclo, en la **línea 20** se compara cada elemento con el valor buscado. En el caso de encontrarlo (**línea 21**) se devuelve la posición.
- Si el valor no está en la lista se recorrerá la lista entera, haciendo una comparación por 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**.

Diremos que este algoritmo 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.

8.4. Buscar sobre una lista ordenada

Por supuesto que si la lista está ordenada podemos hacer lo mismo que antes, con algunas modificaciones que den cuenta de la condición de ordenada de la lista.

Ejercicio 8.1. Modificar la búsqueda lineal para el caso de listas ordenadas. ¿Cuál es nuestra nueva hipótesis sobre comportamiento del algoritmo?

8.5. Búsqueda binaria

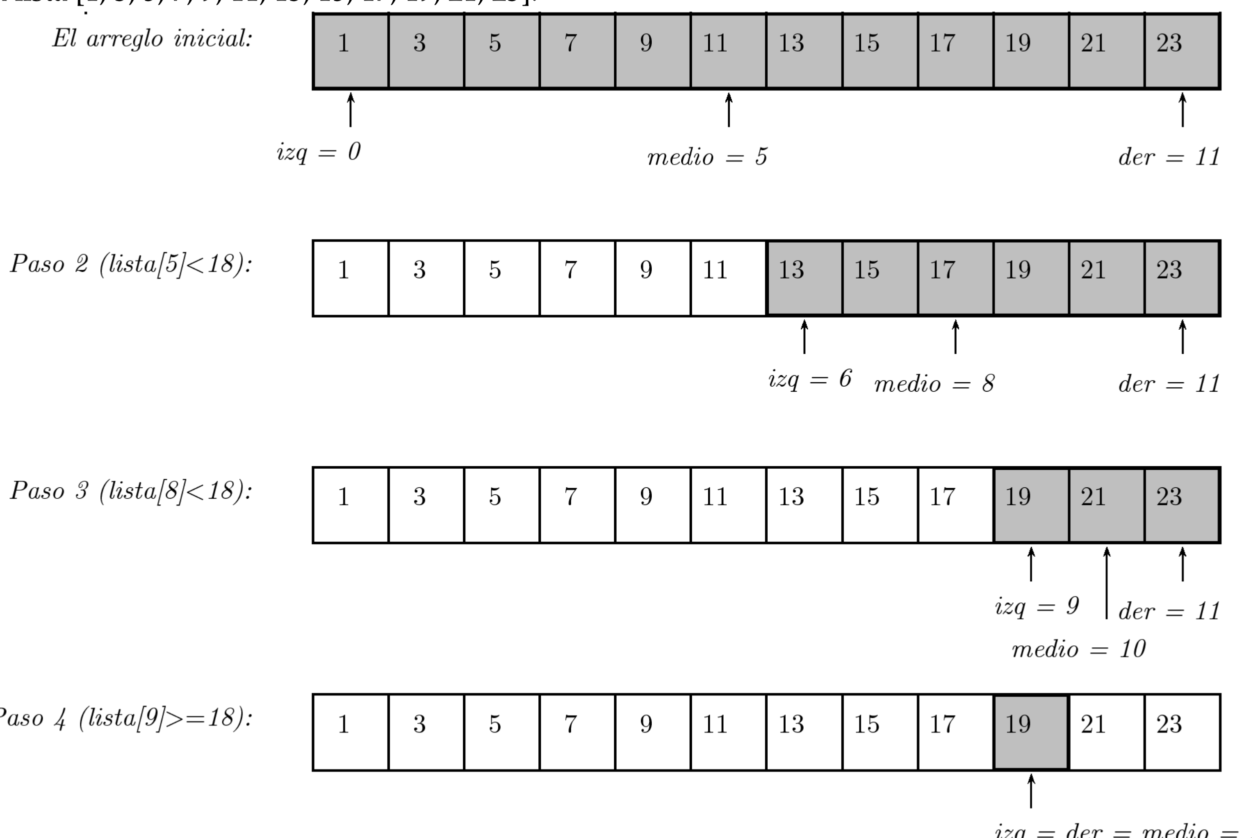
¿Podemos hacer algo mejor? Trataremos de aprovechar el hecho de que la lista está ordenada y vamos a hacer algo distinto: nuestro espacio de búsqueda se irá achicando a segmentos cada vez menores de la lista original. La idea es descartar segmentos de la lista donde el valor seguro que no puede estar:

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 el gráfico que se incluye a continuación, vemos qué pasa cuando se busca el valor 18 en la lista [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23].



Como no se encontró al valor buscado, devuelve -1.
En el Código 8.3 mostramos una posible implementación de este algoritmo.

A continuación varias ejecuciones de prueba:

```
Dame una lista ordenada ([[]] para terminar): [1, 3, 5]
¿Valor buscado?: 0
DEBUG: izq: 0 der: 2 medio: 1
DEBUG: izq: 0 der: 0 medio: 0
Resultado: -1
Dame una lista ordenada ([[]] para terminar): [1, 3, 5]
¿Valor buscado?: 1
DEBUG: izq: 0 der: 2 medio: 1
DEBUG: izq: 0 der: 0 medio: 0
```

Código 8.3 `busqueda_binaria.py`: Función de búsqueda binaria

```
#!/usr/bin/env python
# encoding: latin1

def busqueda_binaria(lista, x):
    """Búsqueda binaria
    Precondición: lista está ordenada
    Devuelve -1 si x no está en lista;
    Devuelve p tal que lista[p] == x, si x está en lista
    """

    # Busca en toda la lista dividiéndola en segmentos y considerando
    # a la lista completa como el segmento que empieza en 0 y termina
    # en len(lista) - 1.

    izq = 0              # izq guarda el índice inicio del segmento
    der = len(lista) - 1 # der guarda el índice fin del segmento

    # un segmento es vacío cuando izq > der:
    while izq <= der:
        # el punto medio del segmento
        medio = (izq+der)/2

        print "DEBUG:", "izq:", izq, "der:", der, "medio:", medio

        # si el medio es igual al valor buscado, lo devuelve
        if lista[medio] == x:
            return medio
        # si el valor del punto medio es mayor que x, sigue buscando
        # en el segmento de la izquierda: [izq, medio-1], descartando la
        # derecha
        elif lista[medio] > x:
            der = medio-1
        # sino, sigue buscando en el segmento de la derecha:
        # [medio+1, der], descartando la izquierda
        else:
            izq = medio+1
        # si no salió del ciclo, vuelve a iterar con el nuevo segmento

    # salió del ciclo de manera no exitosa: el valor no fue encontrado
    return -1

# Código para probar la búsqueda binaria
def main():
    lista = input ("Dame una lista ordenada ([[]] para terminar): ")
    while lista != [[]]:
        x = input("¿Valor buscado?: ")
        resultado = busqueda_binaria(lista, x)
        print "Resultado:", resultado
        lista = input ("Dame una lista ordenada ([[]] para terminar): ")

main()
```

```
Resultado: 0
Dame una lista ordenada ([[]] para terminar): [1, 3, 5]
¿Valor buscado?: 2
DEBUG: izq: 0 der: 2 medio: 1
DEBUG: izq: 0 der: 0 medio: 0
Resultado: -1
Dame una lista ordenada ([[]] para terminar): [1, 3, 5]
¿Valor buscado?: 3
DEBUG: izq: 0 der: 2 medio: 1
Resultado: 1
Dame una lista ordenada ([[]] para terminar): [1, 3, 5]
¿Valor buscado?: 5
DEBUG: izq: 0 der: 2 medio: 1
DEBUG: izq: 2 der: 2 medio: 2
Resultado: 2
Dame una lista ordenada ([[]] para terminar): [1, 3, 5]
¿Valor buscado?: 6
DEBUG: izq: 0 der: 2 medio: 1
DEBUG: izq: 2 der: 2 medio: 2
Resultado: -1
Dame una lista ordenada ([[]] para terminar): []
¿Valor buscado?: 0
Resultado: -1
Dame una lista ordenada ([[]] para terminar): [1]
¿Valor buscado?: 1
DEBUG: izq: 0 der: 0 medio: 0
Resultado: 0
Dame una lista ordenada ([[]] para terminar): [1]
¿Valor buscado?: 3
DEBUG: izq: 0 der: 0 medio: 0
Resultado: -1
Dame una lista ordenada ([[]] para terminar): [[]]
```

¿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 porque 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.

Vemos entonces que si `lista` es una lista ordenada, la búsqueda binaria es muchísimo más eficiente que la búsqueda lineal (por ejemplo, dado que 2^{20} es aproximadamente 1.000.000, si `lista` tiene 1.000.000 de elementos, la búsqueda lineal sobre `lista` será proporcional a 1.000.000, y en promedio hará unas 500.000 comparaciones, mientras que la búsqueda binaria hará como máximo 20 comparaciones).

8.6. 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**.

Unidad 9

Diccionarios

En esta unidad analizaremos otro tipo de dato 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 tipo.

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.

Cualquier variable 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, objetos, etc.

Sabías que ...

En otros lenguajes, a los diccionarios se los llama *arreglos asociativos*, *matrices asociativas*, 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}
```

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.

```
print materias["lunes"]
```

Sin embargo, esto falla si se provee una clave que no está en el diccionario. Es posible, por otro lado, utilizar la función `get`, que devuelve el valor `None` si la clave no está en el diccionario, o un valor por omisión que se establece opcionalmente.

```
>>> print materias["domingo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'domingo'
>>> print materias.get("domingo")
None
>>> print 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 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 dia, ":", codigos
```

Para verificar si una clave se encuentra en el diccionario, es posible utilizar la función has_key o la palabra reservada **in**.

```
d = {'x': 12, 'y': 7}
if d.has_key('x'):
    print d['x']    # Imprime 12
if d.has_key('z'):
    print d['z']    # No se ejecuta
if 'y' in d:
    print d['y']    # Imprime 7
```

Más allá de la creación y el acceso, hay muchas otras operaciones que se pueden realizar sobre los diccionarios, para poder 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 una variable 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 utilizarlo 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 es muy mala.

9.4. Resumen

- Los diccionarios (llamados *arreglos asociativos* o *tablas de hash* en otros lenguajes), son una estructura de datos muy poderosa, que permite asociar un valor a una clave.
- Las claves deben ser de tipo inmutable, los valores pueden ser de cualquier tipo.
- Los diccionarios no están ordenados. Si bien se los puede recorrer, el orden en el que se tomarán los elementos no está determinado.

Referencia del lenguaje 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.

`diccionario.has_key(clave)`
Indica si un diccionario tiene o no una determinada clave. Es posible obtener el mismo resultado utilizando: `if clave in diccionario:`.

`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, no da un error, sino que devuelve el valor predeterminado o `None` en el caso de que no se haya establecido.

`for clave in diccionario:`
Esta estructura permite recorrer una a una todas las claves almacenadas en el diccionario.

`diccionario.keys()`
Devuelve una lista desordenada, con todas las claves que se hayan ingresado al diccionario

`diccionario.values()`
Devuelve una lista desordenada, con todos los valores que se hayan ingresado al diccionario.

`diccionario.items()`
Devuelve una lista desordenada con tuplas de dos elementos, en las que el primer elemento es la clave y el segundo el valor.

`diccionario.pop(clave)`
Devuelve el valor asociado a la clave, y elimina la clave y el valor asociado del diccionario.

`diccionario.popitem()`
Devuelve un elemento al azar del diccionario, representándolo como una tupla `(clave, valor)` y elimina esta pareja del diccionario.

`diccionario.clear()`
Elimina todos los elementos del diccionario

Unidad 10

Contratos y Mutabilidad

En esta unidad se le dará cierta formalización a algunos temas que se habían visto 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 modelizar 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 *assert*). 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`, está instrucción recibe una condición a verificar y, opcionalmente, un mensaje de error que devolverá 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):
    """ Calculo 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 * 1.0 )
```

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. 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.

```
1 def maximo(lista):
2     "Devuelve el elemento máximo de la lista o None si está vacía."
3     if not len(lista):
4         return None
5     max_elem = lista[0]
6     for elemento in lista:
7         if elemento > max_elem:
8             max_elem = elemento
9     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.

```
1 def potencia(b, n):
2     "Devuelve la potencia n del número b, con n entero mayor que 0."
3     p = 1
4     for i in range(n):
5         p *= b
6     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.

```
1 def suma(lista):
2     "Devuelve la suma de todos los elementos de la lista."
3     suma = 0
4     for elemento in lista:
5         suma += elemento
6     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 variables indicamos si son mutables o inmutables.

Cuando una variable 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.

```
>>> a="ejemplo"
>>> a="otro"
>>> a[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 `a` pasa a *apuntar* 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 *apuntan* a un mismo dato, y este dato se modifica, el cambio se ve 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 una variable, 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 **apunte** 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 = range(len(lista))
...     print lista
...
>>> lista = [10, 20, 30, 40]
>>> no_cambia_lista(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 cumplidas 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 del lenguaje Python

`assert condicion[,mensaje]`
Verifica si la condición es verdadera. En caso contrario, levanta una excepción con el mensaje recibido por parámetro.

10.5. Apéndice - 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.

[1] http://en.wikipedia.org/wiki/Invariant_(computer_science)

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.

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 una variable 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.

11.1. Cerrar un archivo

Al terminar de trabajar con un archivo, es recomendable cerrarlo, por diversos motivos: en algunos sistemas los archivos sólo pueden ser abiertos de a un programa por la vez; en otros, lo

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.

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()
```

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 como en el Código 11.1.

Código 11.1 `numera_lineas.py`: Imprime las líneas de un archivo con su número

```
1 archivo = open("archivo.txt")
2 i = 1
3 for linea in archivo:
4     linea = linea.rstrip("\n")
5     print "%4d: %s" % (i, linea)
6     i+=1
7 archivo.close()
```

La llamada a `rstrip` es necesaria ya que cada línea que se lee del archivo contiene un fin de línea y con la llamada a `rstrip("\n")` se remueve.

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 Macintosh 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`'.

Si bien esto es algo que hay que tener en cuenta en una diversidad de casos, en particular en Python por omisión se maneja cualquier tipo de fin de línea como si fuese un '`\n`', salvo que se le pida lo contrario. Para manejar los caracteres de fin de línea *a mano* se puede poner una 'U' en el parámetro modo que le pasamos a `open`.

Otra opción para hacer exactamente lo mismo sería utilizar la función de Python `enumerate(secuencia)`. Esta función devuelve un contador por cada uno de los elementos que se recorren, puede usarse con cualquier tipo de secuencia, incluyendo archivos. La versión equivalente se muestra en el Código 11.2.

Código 11.2 `numera_lineas2.py`: Imprime las líneas de un archivo con su número

```
1 archivo = open("archivo.txt")
2 for i, linea in enumerate(archivo):
3     linea = linea.rstrip("\n")
4     print "%4d: %s" % (i, linea)
5 archivo.close()
```

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 `read` 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.

Código 11.3 `genera_saludo.py`: Genera el archivo saludo.py

```
1 saludo = open("saludo.py", "w")
2 saludo.write("""
3 print "Hola Mundo"
4 """)
5 saludo.close()
```

El ejemplo que se muestra en el Código 11.3 contiene un programa Python que a su vez genera el código de otro programa Python.

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 *log*), 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 11.4 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 `abrir_log` para abrir el archivo de log, luego llamar a `guardar_log` por cada mensaje que se quiera registrar, y finalmente llamar a `cerrar_log` cuando se quiera concluir la registración de mensajes.

Por ejemplo, en el Código 11.5 se muestra un posible programa que utiliza el módulo de log incluido anteriormente.

Código 11.4 `log.py`: Módulo para manipulación de archivos de log

```
1  #! /usr/bin/env python
2  # encoding: latin1
3
4  # El módulo datetime se utiliza para obtener la fecha y hora actual.
5  import datetime
6
7  def abrir_log(nombre_log):
8      """ Abre el archivo de log indicado. Devuelve el archivo abierto.
9      Pre: el nombre corresponde a un nombre de archivo válido.
10     Post: el archivo ha sido abierto posicionándose al final. """
11     archivo_log = open(nombre_log, "a")
12     guardar_log(archivo_log, "Iniciando registro de errores")
13     return archivo_log
14
15 def guardar_log(archivo_log, mensaje):
16     """ Guarda el mensaje en el archivo de log, con la hora actual.
17     Pre: el archivo de log ha sido abierto correctamente.
18     Post: el mensaje ha sido escrito al final del archivo. """
19     # Obtiene la hora actual en formato de texto
20     hora_actual = str(datetime.datetime.now())
21     # Guarda la hora actual y el mensaje de error en el archivo
22     archivo_log.write(hora_actual+" "+mensaje+"\n")
23
24 def cerrar_log(archivo_log):
25     """ Cierra el archivo de log.
26     Pre: el archivo de log ha sido abierto correctamente.
27     Post: el archivo de log se ha cerrado. """
28     guardar_log(archivo_log, "Fin del registro de errores")
29     archivo_log.close()
```

Código 11.5 `usa_log.py`**: Módulo que utiliza el módulo de log**

```
1 #!/usr/bin/env python
2 #encoding latin1
3
4 import log
5
6 # Constante que contiene el nombre del archivo de log a utilizar
7 ARCHIVO_LOG = "mi_log.txt"
8
9 def main():
10     # Al comenzar, abrir el log
11     archivo_log = log.abrir_log(ARCHIVO_LOG)
12     # ...
13     # Hacer cosas que pueden dar error
14     if error:
15         guardar_log(archivo_log, "ERROR: "+error)
16     # ...
17     # Finalmente cerrar el archivo
18     log.cerrar_log(archivo_log)
19
20 main()
```

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:

```
2010-04-10 15:20:32.229556 Iniciando registro de errores
2010-04-10 15:20:50.721415 ERROR: no se pudo acceder al recurso
2010-04-10 15:21:58.625432 ERROR: formato de entrada inválido
2010-04-10 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 parametro de modo.

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.

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.

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(corrimiento, desde)`, que permite desplazarse una cantidad de bytes en el archivo, contando desde el comienzo del archivo, desde la posición actual o desde el final.

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 11.6 se muestra un módulo capaz de guardar y recuperar los puntajes en el formato especificado.

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
```

Código 11.6 `puntajes.py`: Módulo para guardar y recuperar puntajes en un archivo

```
1  #! /usr/bin/env python
2  # encoding: latin1
3
4  def guardar_puntajes(nombre_archivo, puntajes):
5      """ Guarda la lista de puntajes en el archivo.
6      Pre: nombre_archivo corresponde a un archivo válido,
7           puntajes corresponde a una lista de tuplas de 3 elementos.
8      Post: se guardaron los valores en el archivo,
9            separados por comas.
10     """
11
12     archivo = open(nombre_archivo, "w")
13     for nombre, puntaje, tiempo in puntajes:
14         archivo.write(nombre+","+str(puntaje)+","+tiempo+"\n")
15     archivo.close()
16
17 def recuperar_puntajes(nombre_archivo):
18     """ Recupera los puntajes a partir del archivo provisto.
19         Devuelve una lista con los valores de los puntajes.
20     Pre: el archivo contiene los puntajes en el formato esperado,
21          separados por comas
22     Post: la lista devuelta contiene los puntajes en el formato:
23           [(nombre1,puntaje1,tiempo1),(nombre2,puntaje2,tiempo2)].
24     """
25
26     puntajes = []
27     archivo = open(nombre_archivo, "r")
28     for linea in archivo:
29         nombre, puntaje, tiempo = linea.rstrip("\n").split(",")
30         puntajes.append((nombre,int(puntaje),tiempo))
31     archivo.close()
32     return 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 van a desperdiciar un poco más de 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 rápido acceder a los datos. Además, para ciertas aplicaciones como archivos de sonido o video, tendría poco sentido almacenarlos en archivos de texto.

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, se parece 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` que ya se encuentra preparado para eso.

La funciones del ejemplo anterior podría programarse mediante el módulo csv. En el Código 11.7 se muestra una posible implementación que utiliza este módulo.

Si se prueba este código, se obtiene un resultado idéntico al obtenido anteriormente:

```
>>> import puntajes_csv
>>> valores = [("Pepe", 108, "4:16"), ("Juana", 2315, "8:42")]
>>> puntajes_csv.guardar_puntajes("puntajes.txt", valores)
>>> recuperado = puntajes_csv.recuperar_puntajes("puntajes.txt")
>>> print recuperado
[('Pepe', 108, '4:16'), ('Juana', 2315, '8:42')]
```

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.

Código 11.7 `puntajes_csv.py`: Módulo para guardar y recuperar puntajes en un archivo que usa csv

```
#! /usr/bin/env python
# encoding: latin1

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.
    """

    archivo = open(nombre_archivo, "w")
    archivo_csv = csv.writer(archivo)
    archivo_csv.writerows(puntajes)
    archivo.close()

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 = []
    archivo = open(nombre_archivo, "r")
    archivo_csv = csv.reader(archivo)
    for nombre, puntaje, tiempo in archivo_csv:
        puntajes.append((nombre, int(puntaje), tiempo))
    archivo.close()
    return puntajes
```

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.

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 11.8 se muestra el mismo ejemplo de almacenamiento de puntajes, utilizando el módulo `pickle`.

Código 11.8 `puntajes_pickle.py`: Módulo para guardar y recuperar puntajes en un archivo que usa pickle

```
1  #! /usr/bin/env python
2  # encoding: latin1
3
4  import pickle
5
6  def guardar_puntajes(nombre_archivo, puntajes):
7      """ Guarda la lista de puntajes en el archivo.
8      Pre: nombre_archivo corresponde a un archivo válido,
9           puntajes corresponde a los valores a guardar
10     Post: se guardaron los valores en el archivo en formato pickle.
11     """
12
13     archivo = open(nombre_archivo, "w")
14     pickle.dump(puntajes, archivo)
15     archivo.close()
16
17 def recuperar_puntajes(nombre_archivo):
18     """ Recupera los puntajes a partir del archivo provisto.
19         Devuelve una lista con los valores de los puntajes.
20     Pre: el archivo contiene los puntajes en formato pickle
21     Post: la lista devuelta contiene los puntajes en el
22           mismo formato que se los almacenó.
23     """
24
25     archivo = open(nombre_archivo, "r")
26     puntajes = pickle.load(archivo)
27     archivo.close()
28     return puntajes
```

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 la forma:

```
(lp0
(S'Pepe'
p1
I108
S'4:16'
p2
tp3
a(S'Juana'
p4
I2315
S'8:42'
p5
tp6
```

En el apéndice de esta unidad puede verse una aplicación completa de una agenda, que utiliza pickle para almacenar datos en archivos.

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 Macintosh, \ en Windows. La manera de acceder a directorios independientemente del sistema en el que estamos desde Python es usando el modulo `os`.

```
os.path.join("data","archivo.csv")
```

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 intrucciones 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.
- En el caso de los archivos binarios, cada formato tiene sus propias reglas a seguir.
- 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.

- Leer todo el contenido de un archivo, puede consumir memoria innecesariamente.

Referencia del lenguaje Python

`archivo=open(nombre[,modo[,tamaño_buffer]])`
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. `tamaño_buffer` es un entero que especifica el tamaño del buffer deseado, si es negativo (por omisión es `-1`) el sistema operativo decide el tamaño del buffer, si es `0` no se usa buffer, si es `1` se usa buffer por líneas.

`archivo.close()`
Cierra el archivo.

`linea=archivo.readline()`
Lee una línea de texto del archivo

`for linea in archivo:`

```
for linea in archivo:
    # procesar linea
```

Itera sobre las lineas del archivo.

`lineas = archivo.readlines()`
Devuelve una lista con todas las líneas del archivo.

`bytes = archivo.read([n])`
Devuelve la cadena de `n` bytes situada en la posición actual de `archivo`.
Si la cadena devuelta no contiene ningún caracter, es que se ha llegado al final del archivo.
De omitirse el parámetro `n`, devuelve una cadena que contiene 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`, es equivalente a la cantidad de bytes desde el comienzo del archivo.

`archivo.seek(corrimiento, [desde])`
Modifica la posición actual en `archivo`, trasladándose `corrimiento` bytes. El parámetro opcional `desde` especifica desde dónde se mide el valor que le pasemos a `corrimiento`.

- `desde=0`, contará desde el comienzo del archivo. *Valor predeterminado.*
- `desde=1`, contará desde la posición actual.
- `desde=2`, contará desde el final del archivo.

Ejemplos:

```
archivo.seek(0)      # va al principio del archivo
archivo.seek(0,2)    # va al final del archivo
archivo.seek(-16,1) # retrocede 16 bytes de la posición actual
```

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.
Ejemplo de uso:

```
1  import os
2
3  nombre="mi_archivo.txt"
4
5  if not os.path.exists(nombre):
6      archivo = open(nombre,"w+")
7  elif os.path.isfile(nombre):
8      archivo = open(nombre,"r+")
9  else:
10     print "Error, %s no es un archivo" % nombre
```

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. Apéndice

A continuación, el código para un programa de agenda que utiliza archivos csv. Luego, 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 csv

```
1 #!/usr/bin/env python
2 # encoding: latin1
3
4 import csv
5
6 ARCHIVO="agenda.csv"
7 CAMPOS=["Nombre","Apellido","Telefono","Cumpleaños"]
8
9 def leer_csv(datos_csv):
10     """ Devuelve la siguiente línea o None si se terminó el archivo. """
11     try:
12         return datos_csv.next()
13     except:
14         return None
15
16 def leer_datos(archivo):
17     """ Carga todos los datos del archivo en una lista y la devuelve. """
18     abierto = open(archivo)
19     datos_csv = csv.reader(abierto)
20     campos = leer_csv(datos_csv)
21     datos = []
22     elemento = leer_csv(datos_csv)
23     while elemento:
24         datos.append(elemento)
25         elemento = leer_csv(datos_csv)
26     abierto.close()
27     return datos
28
29 def guardar_datos(datos, archivo):
30     """ Guarda los datos recibidos en el archivo. """
31     abierto = open(archivo,"w")
32     datos_csv = csv.writer(abierto)
33     datos_csv.writerow(CAMPOS)
34     datos_csv.writerows(datos)
35     abierto.close()
36
37 def leer_busqueda():
38     """ Solicita al usuario nombre y apellido y los devuelve. """
39     nombre = raw_input("Nombre: ")
40     apellido = raw_input("Apellido: ")
41     return (nombre,apellido)
42
43 def buscar(nombre, apellido, datos):
44     """ Busca el primer elemento que coincida con nombre y con apellido. """
45     for elemento in datos:
46         if nombre in elemento[0] and apellido in elemento[1]:
47             return elemento
48     return None
```

```
49
50 def menu_alta(nombre, apellido, datos):
51     """ Pregunta si se desea ingresar un nombre y apellido y
52         de ser así, pide los datos al usuario. """
53     print "No se encuentra %s %s en la agenda." % (nombre, apellido)
54     confirmacion = raw_input("¿Desea ingresarlo? (s/n) ")
55     if confirmacion.lower() != "s":
56         return
57     telefono = raw_input("Telefono: ")
58     cumple = raw_input("Cumpleaños: ")
59     datos.append([nombre,apellido,telefono,cumple])
60
61 def mostrar_elemento(elemento):
62     """ Muestra por pantalla un elemento en particular. """
63     print
64     print "%s %s" % (elemento[0],elemento[1])
65     print "Telefono: %s" % elemento[2]
66     print "Cumpleaños: %s" % elemento[3]
67     print
68
69 def menu_elemento():
70     """ Muestra por pantalla las opciones disponibles para un elemento
71         existente. """
72     o = raw_input("b: borrar, m: modificar, ENTER para continuar (b/m): ")
73     return o.lower()
74
75 def modificar(viejo, nuevo, datos):
76     """ Reemplaza el elemento viejo con el nuevo, en la lista datos. """
77     indice = datos.index(viejo)
78     datos[indice] = nuevo
79
80 def menu_modificacion(elemento, datos):
81     """ Solicita al usuario los datos para modificar una entrada. """
82     nombre = raw_input("Nuevo nombre: ")
83     apellido = raw_input("Nuevo apellido: ")
84     telefono = raw_input("Nuevo teléfono: ")
85     cumple = raw_input("Nuevo cumpleaños: ")
86     modificar(elemento, [nombre, apellido, telefono, cumple], datos)
87
88 def baja(elemento, datos):
89     """ Elimina un elemento de la lista. """
90     datos.remove(elemento)
91
92 def confirmar_salida():
93     """ Solicita confirmación para salir """
94     confirmacion = raw_input("¿Desea salir? (s/n) ")
95     return confirmacion.lower() == "s"
96
97 def agenda():
98     """ Función principal de la agenda.
99         Carga los datos del archivo, permite hacer búsquedas, modificar
100        borrar, y al salir guarda. """
101    datos = leer_datos(ARCHIVO)
102    fin = False
```

```
103     while not fin:
104         (nombre, apellido) = leer_busqueda()
105         if nombre == "" and apellido == "":
106             fin = confirmar_salida()
107             continue
108         elemento = buscar(nombre, apellido, datos)
109         if not elemento:
110             menu_alta(nombre, apellido, datos)
111             continue
112         mostrar_elemento(elemento)
113         opcion = menu_elemento()
114         if opcion == "m":
115             menu_modificacion(elemento, datos)
116         elif opcion == "b":
117             baja(elemento, datos)
118     guardar_datos(datos, ARCHIVO)
119 
120 agenda()
```

`agenda-pickle.py` Diferencia de agenda con datos en pickle

```
1 import pickle
2 
3 ARCHIVO="agenda.dat"
4 
5 def leer_datos(archivo):
6     """ Carga todos los datos del archivo en una lista y la devuelve. """
7     abierto = open(archivo)
8     datos = pickle.load(archivo)
9     abierto.close()
10     return datos
11 
12 def guardar_datos(datos, archivo):
13     """ Guarda los datos recibidos en el archivo. """
14     abierto = open(archivo,"w")
15     pickle.dump(archivo, datos)
16     abierto.close()
```

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 cuando 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, si dentro de una función surge

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 continua propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa. Veamos entonces como manejar excepciones.

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, se utiliza el término *levantar* para referirse a la acción de generar 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 except 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**, es decir que 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 como 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).

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, siempre 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.

```
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 no está cerrado hay que cerrarlo
    if not(archivo.closed):
        archivo.close()
```

12.2.2. Procesamiento y propagación de excepciones

Hemos visto cómo atrapar excepciones, es necesario ahora que veamos qué se supone que hagamos al atrapar 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. Para hacer esto Python nos brinda la instrucción **`raise`**.

Si se invoca esta instrucción dentro de un bloque **`except`**, sin pasarle parámetros, Python levantará la excepción atrapada por ese bloque.

También podría ocurrir que en lugar de propagar la excepción tal cual fue atrapada, quisiéramos lanzar una excepción distinta, más significativa para quien invocó a la función actual y que posiblemente contenga cierta información de contexto. Para levantar una excepción de cualquier tipo, utilizamos también la sentencia **`raise`**, pero indicándole el tipo de excepción que deseamos lanzar y pasando a la excepción los parámetros con información adicional que queramos brindar.

El siguiente fragmento de código muestra este uso de **`raise`**.

```
def dividir(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        return resultado
    except ZeroDivisionError:
        raise ZeroDivisionError("El divisor no puede ser cero")
```

12.2.3. Acceso a información de contexto

Para acceder a la información de contexto estando dentro de un bloque **except** existen dos alternativas. Se puede utilizar la función `exc_info` del módulo `sys`. Esta función devuelve una tupla con información sobre la última excepción atrapada en un bloque **except**. Dicha tupla contiene tres elementos: el tipo de excepción, el valor de la excepción y las llamadas realizadas.

Otra forma de obtener información sobre la excepción es utilizando la misma sentencia **except**, pasándole un identificador para que almacene una referencia a la excepción atrapada.

```
try:
    # código que puede lanzar una excepción
except Exception, ex:
    # procesamiento de la excepción cuya información
    # es accesible a través del identificador ex
```

Sabías que ...

En otros lenguajes, como el lenguaje 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 que deba ser "comparable", o "iterable".

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 el código 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.

```
1 def factorial(n):
2     """ Calcula el factorial de n.
3     Pre: n debe ser un entero, mayor igual a 0
4     Post: se devuelve el valor del factorial pedido
5     """
6     assert n >= 0, "n debe ser mayor igual a 0"
7     fact=1
8     for i in xrange(2,n+1):
9         fact*=i
10    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, no se debe asumir que vaya a ingresarlo correctamente. Se lo debe guardar en una cadena y luego convertir a un número, es por eso que es recomendable el uso de la función `raw_input` ya que devuelve una cadena que puede ser procesada posteriormente.

```
def lee_entero():
    """ Solicita un valor entero y lo devuelve.
        Si el valor ingresado no es entero, lanza una excepción. """
    valor = raw_input("Ingrese un número entero: ")
    return int(valor)
```

Esta función devuelve un valor entero, o lanza una excepción si la conversión no fue posible. Sin embargo, esto no es suficiente. En el caso en el que el usuario no haya ingresado la información correctamente, es necesario volver a solicitarla.

```
def lee_entero():
    """ Solicita un valor entero y lo devuelve.
        Mientras el valor ingresado no sea entero, vuelve a solicitarlo. """
    while True:
        valor = raw_input("Ingrese un número entero: ")
        try:
```

```
            valor = int(valor)
            return valor
        except ValueError:
            print "ATENCIÓN: Debe ingresar un número entero."
```

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 lee_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 = raw_input("Ingrese un número entero: ")
        try:
            valor = int(valor)
            return valor
        except ValueError:
            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:
        print "Ingrese A (Altas) - B (Bajas) - M (Modificaciones): ",
        opcion = raw_input().upper()
        if opcion 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(variable)`. Por ejemplo, para comprobar que una variable contenga un tipo entero podemos hacer:

```
if type(i) != 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, long, 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`, `long` 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, long, 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, como el ejemplo anterior, en que se requería que se pudiera convertir a un entero.

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. """
    try:
        dividendo = int(x)
        divisor = int(y)
        return dividendo/divisor
    except ValueError:
        raise ValueError, "x e y deben poder convertirse a enteros"
    except ZeroDivisionError:
        raise ZeroDivisionError, "y no puede ser cero"
```

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.

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 interprete 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.

Sabías que ...

Python utiliza la idea de *duck typing*, que viene del concepto de que 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.

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 de Programación Orientada a Objetos.

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.

```
if not hasattr(i,"__add__"):
    raise TypeError, "El elemento no es sumable"
```

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 dentro de un bloque **`try`** y manejar la excepción en caso de error, que saber cuáles son los detalles de la implementación de `__add__` de cada tipo interactuante.

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 deseable encerrarlo en los bloques correspondientes para actuar en consecuencia.

- Si una función no contempla la excepción, ésta es levantada a la función invocante, si ésta no la contempla, la excepción se pasa a la invocante, hasta que se llega a una porción de código que contemple la excepción, o bien se interrumpe la ejecución del programa.

- Cuando una porción de código puede levantar diversos tipos de excepciones, es deseable tratarlas por separado, si bien es posible tratarlas todas juntas.

- Cuando se genera 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, 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 del lenguaje Python

`try: ... except:`

```
try:
    # código
except [tipo_de_excepción [, variable]]:
    # manejo de excepción
```

Puede tener tantos **`except`** como sea necesario, el último puede no tener un tipo de excepción asociado.
Si el código dentro del bloque **`try`** levanta una excepción, se ejecuta el código dentro del bloque **`except`** correspondiente.

`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 [, 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. El primer parámetro corresponde al tipo de excepción a levantar. El mensaje es opcional, se utiliza para dar más información sobre el error acontecido.

12.5. Apéndice

A continuación se muestran dos ejemplos de implementación de un programa que calcula la serie de Fibonacci para números menores a 20.

En primer lugar se muestra una implementación sin uso de excepciones, con las herramientas vistas antes de esta unidad.

```
def calcularFibonacciSinExcepciones(n):
    if (n>=20) or (n<=0):
        print ''' Ha ingresado un valor incorrecto.
El valor debe ser número entero mayor a cero y menor a 20'''
        return
    salida=[]
    a,b = 0,1
    for x in range(n):
        salida.append(b)
        a, b = b, a+b
    return salida

def mainSinExcepciones():
    input = raw_input('Ingrese n para calcular Fibonacci(n):')
    n = int(input)
    print calcularFibonacciSinExcepciones(n)
```

A continuación un código que utiliza excepciones para manejar la entrada de mejor manera.

```
def calcularFibonacciConExcepciones(n):
    try:
        assert(n>0)
        assert(n<20)
    except AssertionError:
        raise ValueError
    a=0
    b=1
    salida = []
    for x in range(n):
        salida.append(b)
        a, b = b, a+b
    return salida

def mainConExcepciones():
    try:
        input = raw_input('Ingrese n para calcular Fibonacci(n):')
        n = int(input)
        print calcularFibonacci2(n)
    except ValueError:
        print '''Ha ingresado un valor incorrecto.
El valor debe ser un número entero mayor a cero y menor a 20'''
```

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 indican algunas formas más complejas de procesar la información leída. En particular, dos algoritmos bastante comunes, llamados *corte de control* y *apareo de archivos*.

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.

ventas.py Recorre un archivo de ventas e imprime totales y subtotales

```
1 # encoding: latin1
2 import csv
3
4 def leer_datos(datos):
5     """ Devuelve el siguiente registro o None si no hay más """
6     try:
7         return datos.next()
8     except:
9         return None
10
11 def ventas_clientes_mes(archivo_ventas):
12     """ Recorre un archivo csv, con la información almacenada en el
13     formato: cliente,año,mes,día,venta """
14
15     # Inicialización
16     ventas = open(archivo_ventas)
17     ventas_csv = csv.reader(ventas)
18
19     item = leer_datos(ventas_csv)
20     total = 0
21
22     while item:
23         # Inicialización para el bucle de cliente
24         cliente = item[0]
25         total_cliente = 0
26         print "Cliente %s" % cliente
27
28         while item and item[0] == cliente:
29             # Inicialización para el bucle de año
30             anyo = item[1]
31             total_anyo = 0
32             print "\tAño: %s" % anyo
33
34             while item and item[0] == cliente and item[1] == anyo:
35                 mes, monto = item[2], float(item[3])
36                 print "\t\tVentas del mes %s: %.2f" % (mes, monto)
37                 total_anyo += monto
38                 # Siguiente registro
39                 item = leer_datos(ventas_csv)
40
41             # Final del bucle de año
42             print "\tTotal para el año %s: %.2f" % (anyo, total_anyo)
43             total_cliente += total_anyo
44
45         # Final del bucle de cliente
46         print "Total para el cliente %s: %.2f\n" % (cliente, total_cliente)
47         total += total_cliente
48
49     # Final del bucle principal
50     print "Total general: %.2f" % total
51
52     # Cierre del archivo
53     ventas.close()
```

```
54
55 ventas_clientes_mes("ventas.csv")
```

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 estar 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.

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 de la siguiente manera.

`notas.py` Recorre un archivo de alumnos y otro de notas e imprime las notas que corresponden a cada alumno

```
1 #!/usr/bin/env python
2 # encoding: latin1
3 import csv
4
5 def leer_datos(datos):
6     """ Obtiene el siguiente registro, o devuelve None si llegó al fin
7         del archivo. """
8     try:
9         return datos.next()
10    except:
11        return None
12
13 def imprimir_notas_alumnos(alumnos, notas):
14    """ Abre los archivos de alumnos y notas, y por cada alumno imprime
15        todas las notas que le corresponden. """
16    notas_a = open(notas)
17    alumnos_a = open(alumnos)
18    notas_csv = csv.reader(notas_a)
19    alumnos_csv = csv.reader(alumnos_a)
20
21    # Saltea los encabezados
22    leer_datos(notas_csv)
```

```
23     leer_datos(alumnos_csv)
24 
25     # Empieza a leer
26     alumno = leer_datos(alumnos_csv)
27     nota = leer_datos(notas_csv)
28     while (alumno):
29         print alumno[2]+", "+alumno[1]+" - "+alumno[0]
30         if (not nota or nota[0] != alumno[0]):
31             print "\tNo se registran notas"
32         while (nota and nota[0] == alumno[0]):
33             print "\t"+nota[1]+": "+nota[2]
34             nota = leer_datos(notas_csv)
35         alumno = leer_datos(alumnos_csv)
36 
37     # Cierro los archivos
38     notas_a.close()
39     alumnos_a.close()
40 
41 imprimir_notas_alumnos("alumnos.csv", "notas.csv")
```

En el ejemplo anterior 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 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 archivos con datos ordenados, y permite generar una salida combinada a partir de estos dos archivos.

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 `miarchivo.readline()`, le estamos diciendo a Python que llame a la función `readline` del tipo `file` para `miarchivo` que es lo mismo que decir que llame al *método* `readline` del objeto `miarchivo`.

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, tienen operaciones propias de cada uno y nos proveen una gran cantidad de funcionalidades que podemos utilizar para nuestros programas.

Como ya se dijo en unidades anteriores, para saber de qué tipo es una variable, utilizamos la función `type`, y para saber qué métodos y atributos tiene esa variable utilizamos la función `dir`.

```
>>> a = open("archivo.txt")
>>> type(a)
```

```
<type 'file'>
>>> dir(a)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__',
'__getattribute__', '__hash__', '__init__', '__iter__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__',
'close', 'closed', 'encoding', 'fileno', 'flush', 'isatty', 'mode', 'name',
'newlines', 'next', 'read', 'readinto', 'readline', 'readlines', 'seek',
'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']
```

En este caso, la función `dir` nos muestra los métodos que tiene un objeto del tipo `file`. 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 (a.name)
<type 'str'>
>>> a.name
'archivo.txt'
>>> type (a.tell)
<type 'builtin_function_or_method'>
>>> a.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.

> Como ya sabemos, 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, 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 distintas variables que son de ese tipo. A las variables que son de una clase en particular, se las 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 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.

En esta primera implementación, optaremos por utilizar la representación de coordenadas cartesianas, e iremos implementando las operaciones a medida que las vayamos necesitando.

En primer lugar, creamos una clase `Punto` que simplemente almacena las coordenadas.

```
1 class Punto(object):
2     """ Representación de un punto en el plano, los atributos son x e y
3         que representan los valores de las coordenadas cartesianas."""
4     def __init__(self, x=0, y=0):
5         "Constructor de Punto, x e y deben ser numéricos"
6         self.x = x
7         self.y = y
```

En la primera línea de código indicamos que vamos a crear una nueva clase, llamada `Punto` La palabra `object` entre paréntesis indica que la clase que estamos creando es un objeto básico, no está basado en ningún objeto más complejo.

> Por convención, en los nombres de las clases definidas por el programador, se escribe cada palabra del nombre con la primera letra en mayúsculas. Ejemplos: `Punto`, `ListaEnlazada`, `Hotel`.

Además definimos uno de los métodos especiales, `__init__`, el **constructor** de la clase. Este método se llama cada vez que se crea una nueva instancia de la clase.

Este método, al igual que todos los métodos de cualquier clase, recibe 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.

Para definir atributos, basta con definir una variable dentro de la instancia, es una buena idea definir todos los atributos de nuestras instancias en el constructor, de modo que se creen con algún valor válido. En nuestro ejemplo `self.x` y `self.y` y se usarán como `punto.x` y `punto.y`.

Para utilizar esta clase que acabamos de definir, lo haremos de la siguiente forma:

```
>>> p = Punto(5,7)
>>> print p
<__main__.Punto object at 0x8e4e24c>
>>> print p.x
5
>>> print p.y
7
```

Al realizar la llamada `Punto(5,7)`, se creó un nuevo punto, y se almacenó una referencia a ese punto en la variable `p`. 5 y 7 son los valores que se asignaron a `x` e `y` respectivamente.

Si bien nosotros no lo invocamos explícitamente, internamente Python realizó la llamada al método `__init__`, asignando así los valores de la forma que se indica en el constructor.

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)
>>> print q.x
A
>>> print 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 `es_numero`, que incluiremos en un módulo llamado `validaciones`:

```
def es_numero(valor):
    """ Indica si un valor es numérico o no. """
    return isinstance(valor, (int, float, long, complex) )
```

Y en el caso de que alguno de los valores no sea numérico, lanzaremos una excepción del tipo `TypeError`. El nuevo constructor quedará así:

```
    def __init__(self, x=0, y=0):
        """ Constructor de Punto, x e y deben ser numéricos,
            de no ser así, se levanta una excepción TypeError """
        if es_numero(x) and es_numero(y):
            self.x=x
            self.y=y
        else:
            raise TypeError("x e y deben ser valores numéricos")
```

Este constructor impide que se creen instancias con valores inválidos para `x` e `y`.

```
>>> p = Punto("A", True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in __init__
TypeError: x e y deben ser valores numéricos
```

Sabías que ...

Python cuenta con un listado de excepciones que se pueden lanzar ante distintas situaciones, en este caso se utilizó `TypeError` que es la excepción que se lanza cuando una operación o función interna se aplica a un objeto de tipo inadecuado. El valor asociado es una cadena con detalles de la incoherencia de tipos.
Otra excepción que podríamos querer utilizar es `ValueError`, que se lanza cuando una operación o función interna recibe un argumento del tipo correcto, pero con un valor inapropiado y no es posible describir la situación con una excepción más precisa.
Si la situación excepcional que queremos indicar no está cubierta por ninguna de las excepciones del lenguaje, podremos crear nuestra propia excepción.
El listado completo de las excepciones provistas por el lenguaje se encuentra en:

- http://docs.python.org/library/exceptions.html
- http://pyspanishdoc.sourceforge.net/lib/module-exceptions.html

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 apreciar la potencia 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)
>>> print 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 un nuevo punto, con 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 un nuevo punto, en lugar de modificar el punto actual.

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 ambos:

```
def distancia(self, otro):
    """ Devuelve la distancia entre ambos puntos. """
    r = self.restar(otro)
    return r.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)
>>> print r.x, r.y
3 4
>>> print r.norma()
5.0
>>> print 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. Un método para mostrar objetos

Para mostrar objetos, Python indica que hay que agregarle a la clase un método especial, llamado `__str__` que debe devolver una cadena de caracteres con lo que queremos mostrar. Ese método se invoca cada vez que se llama 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):
    """ Muestra el punto como un par ordenado. """
    return "(" + str(self.x) + ", " + str(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)
```

Vemos que con `str(p)` se obtiene la cadena construida dentro de `__str__`, y que internamente Python llama a `__str__` cuando se le pide que imprima una variable de la clase `Punto`.

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. Y 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.

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)
>>> print 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)
>>> print p - q
(1, -1)
>>> print p + q
(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#, Perl, Ruby.

14.5. 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 $(puntaje^2) * 10./precio$.

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, ubicación, puntaje, precio).
  - Los valores por omisión para la construcción son: puntaje en 0, precio en `float("inf")` (infinito), nombre y ubicación en '*' (el precio muy alto sirve para que si no se informa el precio de un hotel, se asuma el mayor valor posible.
  - Necesitamos validar que puntaje y precio sean números (utilizaremos la función `es_numero` que ya se usó en el caso de los puntos). Cuando un precio viene en cero se reemplaza su valor por `float("inf")` (de modo de asegurar que el precio nunca quede en cero).
  - Necesitamos validar que nombre y ubicación sean cadenas no vacías (para lo cual tenemos que construir una función `es_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.

Casi todas las tareas, podemos realizarlas con los temas vistos para la creación de la clase `Punto`. Para el último ítem deberemos introducir nuevos métodos especiales.

Ejercicio 14.1. Escribir la función `es_cadena_no_vacia(valor)` que decide si un valor cualquiera es una cadena no vacía o no, e incluirla en el módulo `validaciones`.

El fragmento inicial de la clase programada en Python queda así:

```
1 class Hotel(object):
2     """ Hotel: sus atributos son: nombre, ubicacion, puntaje y precio. """
3
4     def __init__(self, nombre = '*', ubicacion = '*',
5                  puntaje = 0, precio = float("inf")):
6         """ nombre y ubicacion deben ser cadenas no vacías,
7             puntaje y precio son números.
8             Si el precio es 0 se reemplaza por infinito. """
9
10        if es_cadena_no_vacia (nombre):
11            self.nombre = nombre
```

```
12          else:
13              raise TypeError ("El nombre debe ser una cadena no vacía")
14
15          if es_cadena_no_vacia (ubicacion):
16              self.ubicacion = ubicacion
17          else:
18              raise TypeError ("La ubicación debe ser una cadena no vacía")
19
20          if es_numero(puntaje):
21              self.puntaje = puntaje
22          else:
23              raise TypeError ("El puntaje debe ser un número")
24
25          if es_numero(precio):
26              if precio != 0:
27                  self.precio = precio
28              else:
29                  self.precio = float("inf")
30          else:
31              raise TypeError("El precio debe ser un número")
32
33      def __str__(self):
34          """ Muestra el hotel según lo requerido. """
35          return self.nombre + " de "+ self.ubicacion+ \
36                  " - Puntaje: " + str(self.puntaje) + " - Precio: "+ \
37                  str(self.precio)+ " pesos."
```

Con este código tenemos ya la posibilidad de construir hoteles, con los atributos de los tipos correspondientes, y de mostrar los hoteles según nos lo han solicitado.

```
>>> h = Hotel("Hotel City", "Mercedes", 3.25, 78)
>>> print h
Hotel City de Mercedes - Puntaje: 3.25 - Precio: 78 pesos.
```

14.5.1. 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, el método que se debe definir es `__cmp__`, que debe devolver:

- **Un valor entero menor a cero**, si el primer parámetro es menor al segundo.
- **Un valor entero mayor a cero**, si el primer parámetro es mayor que el segundo.
- **Cero**, si ambos parámetros son iguales.

Para crear el método `__cmp__`, definiremos primero un método auxiliar `ratio(self)` que calcula la relación calidad-precio de una instancia de Hotel según la fórmula indicada:

```
    def ratio(self):
        """ Calcula la relación calidad-precio de un hotel de acuerdo
            a la fórmula que nos dio el cliente. """
        return ((self.puntaje**2)*10.)/self.precio
```

A partir de este método es muy fácil crear un método `__cmp__` que cumpla con la especificación previa.

```
    def __cmp__(self, otro):
        diferencia = self.ratio() - otro.ratio()
        if diferencia < 0:
            return -1
        elif diferencia > 0:
            return 1
        else:
            return 0
```

Una vez que está definida esta función podremos realizar todo tipo de comparaciones entre los hoteles:

```
>>> h = Hotel("Hotel City", "Mercedes", 3.25, 78)
>>> i = Hotel("Hotel Mascardi", "Bariloche", 6, 150)
>>> i < h
False
>>> i == h
False
>>> i > h
True
```

14.5.2. Ordenar de menor a mayor listas de hoteles

En una unidad anterior 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 `__cmp__`, 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.

14.5.3. 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 `__cmp__`.

Para situaciones como esta, `sort` puede recibir, opcionalmente, otro parámetro que es la función de comparación a utilizar. Esta función deberá cumplir con el mismo formato que el método `__cmp__`, pero puede ser una función cualquiera, ya sea un método de una clase o una función externa.

Además, para simplificar la escritura de este tipo de funciones, podemos utilizar la función de Python `cmp`, que si le pasamos dos númeroes, devuelve los valores de la forma que necesitamos.

```
    def cmpPrecio(self, otro):
        """ Compara dos hoteles por su precio. """
        return cmp(self.precio, otro.precio)

    def cmpPuntaje(self, otro):
        """ Compara dos hoteles por su puntaje. """
        return cmp(self.puntaje, otro.puntaje)
```

Así, para ordenar según el precio, deberemos hacerlo de la siguiente forma:

```
>>> h1 = Hotel("Hotel Guadalajara", "Pinamar", 2, 55)
>>> h2 = Hotel("Hostería París", "Rosario", 1, 35)
>>> h3 = Hotel("Apart-Hotel Estocolmo", "Esquel", 3, 105)
>>> h4 = Hotel("Posada El Cairo", "Salta", 2.5, 15)
>>> lista = [ h1, h2, h3, h4 ]
>>> lista.sort(cmp=Hotel.cmpPrecio)
>>> for hotel in lista:
...     print hotel
...
Posada El Cairo de Salta - Puntaje: 2.5 - Precio: 15 pesos.
Hostería París de Rosario - Puntaje: 1 - Precio: 35 pesos.
Hotel Guadalajara de Pinamar - Puntaje: 2 - Precio: 55 pesos.
Apart-Hotel Estocolmo de Esquel - Puntaje: 3 - Precio: 105 pesos.
```

14.5.4. 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 `__cmp__`.

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 variables. `p` y `q` son variables distintas, por más que tengan los mismos valores.

Para obtener el comportamiento esperado en estos casos, se redefinen los métodos `__eq__` (correspondiente al operador ==) y `__ne__` (correspondiente a `!=` o <>).

De forma que para poder comparar si dos puntos son o no iguales, deberemos agregar los siguientes dos métodos a la clase Punto:

```
    def __eq__(self, otro):
        """ Devuelve si dos puntos son iguales. """
        return self.x == otro.x and self.y == otro.y

    def __ne__(self, otro):
        """ Devuelve si dos puntos son distintos. """
        return not self == otro
```

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
```

14.6. Ejercicios

Ejercicio 14.2. Modificar el método __cmp__ 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 14.3. 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 5 billetes de 10 pesos, 7 monedas de 25 centavos y 4 monedas de 10 centavos).

Se tiene que poder comparar cajas por la cantidad de dinero que hay en cada una, y además ordenar una lista de cajas de menor a mayor según la cantidad de dinero disponible.

14.7. Resumen

- Los **objetos** son formas ordenadas de agrupar datos (*atributos*) y operaciones sobre estos datos (*métodos*).
- Cada objeto es de una **clase** o tipo, que define cuáles serán sus atributos y métodos. Y cuando se crea una variable de una clase en particular, se crea una **instancia** de esa clase.
- Para nombrar una clase definida por el programador, se suele 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.

Referencia del lenguaje Python

`class una_clase(object):`
Indica que se comienza a definir una clase con el nombre una_clase, que está basada en la clase object.
Dentro de la clase se definen sus métodos y atributos, todos con un nivel de indentación mayor.

`def __init__(self, ...):`
Define el *constructor* de la clase. En general, dentro del constructor se establecen los valores iniciales de todos los atributos.

`variable = una_clase(...)`
Crea una nueva instancia de la clase `una_clase`, los parámetros que se ingresen serán pasados al constructor, luego del parámetro especial `self`.

`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 `variable`.

`variable.atributo`
Permite acceder al valor de un atributo de la instancia. Se lo puede mostrar, guardar o modificar.

```
print variable.atributo # Muestra el valor
a = variable.atributo   # Guarda el valor en otra variable
variable.atributo = b   # Modifica el valor del atributo.
```

`def __str__(self):`
Método especial que debe devolver una cadena de caracteres, con la representación en texto de la instancia.

`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 __cmp__(self, otro):`
Método especial para permitir la comparación de objetos mediante un criterio de orden de menor a mayor. Recibe las dos instancias a comparar.
Debe devolver -1 si el primero es menor, 0 si son iguales y 1 si el segundo es menor.

`def __eq__(self, otro):, def __ne__(self, otro):`
Métodos especiales para comparar objetos sólamente por igualdad o desigualdad. Reciben las dos instancias a comparar. Devuelven `True` o `False` según corresponda.

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.

Se vio, por ejemplo, que 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`).

De la misma forma, hemos utilizado funciones que podían trabajar con los distintos tipos numéricos sin hacer distinción sobre de qué tipo de número se trataba (entero, real, largo o complejo).

Por otro lado, en la unidad anterior se vio también que al construir una clase, es posible incluir el método `__str__` para que cuando se quiera imprimir el objeto se lo haga de la forma deseada; así como una gran variedad de otros métodos especiales, que permiten que operadores comunes sean utilizados sobre distintos tipos de objetos.

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, puede consistir en uno solo o varios métodos o atributos.

Por ejemplo, para que un objeto se pueda comparar con otros, debe cumplir con la interfaz *comparable*, que en Python implica incluir el método `__cmp__` visto en la unidad anterior.

La idea de polimorfismo se basa, entonces, en utilizar distintos tipos de datos a través de una interfaz común.

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, 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 no es lo mismo la respiración de una mojarrita que la de un malvón, no es lo mismo la reproducción de una ameba que la de un elefante.

De la misma forma, al implementar nuestras clases, debemos proveer distintas implementaciones de los métodos que se llaman igual, para que puedan comportarse polimórficamente, como ser las redefiniciones de los métodos `__str__` o `__cmp__` vistas en la unidad anterior.

En particular en Python, la *sobrecarga de operadores*, mencionada anteriormente, 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.

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.

15.1.3. Un ejemplo 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 multiplicar por un escalar (`__mul__(self, escalar)`), podríamos tener la siguiente función polimórfica:

```
def obtener_versor(punto):
    norma = punto.norma()
    return punto * (1.0 / norma)
```

Esta función devolverá un versor de dos dimensiones o de tres dimensiones, según a qué clase pertenezca la variable `punto`.

Atención

A veces puede suceder que una función polimórfica imponga alguna restricción sobre los tipos de los parámetros sobre los que opera. En el ejemplo anterior, el objeto `punto` debe tener el método `norma` y la posibilidad de multiplicarlo por un escalar.

Otro ejemplo que ya hemos visto, utilizando secuencias, es usar un diccionario para contar la frecuencia de aparación de elementos dentro de una secuencia cualquiera.

```
1  def frecuencias(secuencia):
2      """ Calcula las frecuencias de aparición de los elementos de
3          la secuencia recibida.
4          Devuelve un diccionario con elementos: {valor: frecuencia}
5      """
6      # crea un diccionario vacío
7      frec = dict()
8      # recorre la secuencia
9      for elemento in secuencia:
10         frec[elemento] = frec.get(elemento, 0) + 1
11     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}
>>> ran = xrange(3, 10, 2)
>>> frecuencias(ran)
{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 v in seq:
TypeError: 'int' object is not iterable
```

15.2. Herencia

La *herencia* es un mecanismo de la programación orientada a objetos que sirve para crear clases nuevas a partir de clases preexistentes. Se toman (*heredan*) atributos y comportamientos de las clases viejas y se los modifica para modelar una nueva situación.

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` (que contenga como atributos `identificacion`, `nombre`, `apellido`) podemos construir la clase `AlumnoFIUBA` que extiende a `Persona` y agrega como atributo el `padron`.

Para indicar el nombre de la clase base, se la pone entre paréntesis a continuación del nombre de la clase (en lugar de la expresión `object` que poníamos anteriormente –en realidad `object` es el nombre de la clase base genérica–).

Definimos `Persona`:

```
class Persona(object):
    "Clase que representa una persona."
    def __init__(self, identificacion, nombre, apellido):
        "Constructor de Persona"
        self.identificacion = identificacion
        self.nombre = nombre
        self.apellido = apellido
    def __str__(self):
        return " %s: %s, %s" % \
            (str(self.identificacion), self.apellido, self.nombre)
```

A continuación definimos `AlumnoFIUBA` como derivada de `Persona`, de forma tal que inicialice el nuevo atributo, pero a su vez utilice la inicialización de `Persona` para las atributos de la clase base:

```
class AlumnoFIUBA(Persona):
    "Clase que representa a un alumno de FIUBA."
    def __init__(self, identificacion, nombre, apellido, padron):
        "Constructor de AlumnoFIUBA"
        # llamamos al constructor de Persona
        Persona.__init__(self, identificacion, nombre, apellido)
        # agregamos el nuevo atributo
        self.padron = padron
```

Probamos la nueva clase:

```
>>> a = AlumnoFIUBA("DNI 35123456", "Damien", "Thorn", "98765")
>>> print a
DNI 35123456: Thorn, Damien
```

Vemos que se heredó el método `__str__` de la clase base. Si queremos, podemos redefinirlo:

```
    def __str__(self):
        "Devuelve una cadena representativa del alumno"
        return " %d: %s, %s" % \
            (str(self.padron), self.apellido, self.nombre)
```

Volvemos a probar:

```
>>> a = AlumnoFIUBA("DNI 35123456", "Damien", "Thorn", "98765")
>>> print a
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.

Sabías que ...

En el diseño de jerarquias de herencia no siempre es del todo fácil decidir cuándo una clase debe extender a otra. La 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 se puede 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**, pero no la detallaremos por ahora.

La clase de las figuras

Un ejemplo clásico de herencia es el de las figuras cerradas en el plano, con un método para calcular el área. En este caso, la clase base no tiene comportamiento definido ni atributos, dado que cada figura tiene atributos muy distintos (radio en el caso del círculo, base y altura en el caso del triángulo, etc.), y en cuanto al cálculo del área, cada figura tiene una fórmula diferente:

- La clase base:

```
class Figura(object):
    """ Una figura en el plano. """
    def area(self):
        " Este método debe ser redefinido. "
        pass
```

- Los círculos:

```
from math import pi

class Circulo(Figura):
    """ Un círculo en el plano. """
    def __init__(self, radio=0):
        " Constructor de círculo. "
        self.radio = radio

    def area(self):
        " Devuelve el área del círculo. "
        return pi * self.radio * self.radio
```

- Y los triángulos:

```
class Triangulo(Figura):
    """ Un triángulo en el plano. """
    def __init__(self, base=0, altura=0):
        " Constructor de triángulo. "
        self.base = base
        self.altura = altura

    def area(self):
        " Devuelve el área del triángulo. "
        return self.base * self.altura / 2.
```

Y ahora las pruebas:

```
>>> c = Circulo(4)
>>> c.area()
50.26548245743669
>>>
>>> t = Triangulo(3, 5)
>>> t.area()
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 clase `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.

Código 15.1 `Rectangulo.py`: Clase para modelar un Rectángulo

```
1  #! /usr/bin/env python
2  #encoding: latin1
3
4  from Punto import Punto
5
6  class Rectangulo(object):
7      """ Esta clase modela un rectángulo en el plano. """
8
9      def __init__(self, base, altura, origen):
10         """ base (número) es la longitud de su base,
11             altura (número) es la longitud de su base,
12             origen (Punto) es el punto del plano de su esquina
13                           inferior izquierda. """
14         self.base = base
15         self.altura = altura
16         self.origen = origen
17
18     def trasladar(self, dx = 0, dy = 0):
19         self.origen = self.origen + Punto(dx,dy)
20
21     def area(self):
22         return self.base * self.altura
23
24     def __str__(self):
25         """ muestra el rectángulo """
26         return "Base: %s, Altura: %s, Esquina inf. izq.: %s " % \
27                (self.base, self.altura, self.origen)
```

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 del traslado, 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:

```
>>> from Punto import Punto
>>> from Rectangulo import Rectangulo
>>> r = Rectangulo(2, 3, Punto(1, 2))
>>> print r
Base: 2, Altura: 3, Esquina inf. izq.: (1, 2)
>>> print r.area()
6
```

Lo que acabamos de crear es un objeto de acuerdo al siguiente diagrama que se muestra en la Figura 15.3.

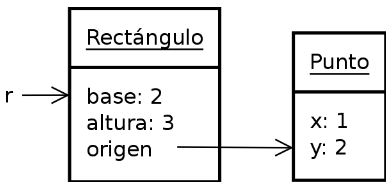


Figura 15.1: Estado de las variables, al momento de crear el rectángulo

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)
>>> print r
Base: 2, Altura: 3, Esquina inf. izq.: (3, 6)
```

También es posible directamente reemplazar el punto contenido, por un nuevo punto.

```
>>> q = Punto(7,2)
>>> r.origen = q
>>> print r
Base: 2, Altura: 3, Esquina inf. izq.: (7, 2)
```

Con lo cual el diagrama pasa a ser el de la Figura 15.3.

> 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 no es tema de esta materia, que se encarga de juntar todos los pedazos de memoria que se descartan durante la ejecución de un programa.

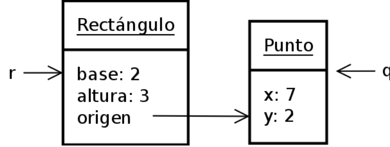


Figura 15.2: Estado de las variables, luego de reemplazar el origen

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 clase contiene como atributo a 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.

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 imprimir el contenido del nodo.

```
class Nodo(object):
    def __init__(self, dato=None, prox = None):
        self.dato = dato
        self.prox = prox
    def __str__(self):
        return str(self.dato)
```

Ejecutamos este código:

```
>>> v3=Nodo("Bananas")
>>> v2=Nodo("Peras", v3)
>>> v1=Nodo("Manzanas", v2)
>>> print v1
Manzanas
>>> print v2
Peras
>>> print v3
Bananas
```

Con esto hemos generado la estructura de la Figura 16.1.

El atributo `prox` de `v3` tiene una referencia nula, lo que indica que `v3` es el último vagón de nuestra estructura.

Hemos creado una lista en forma manual. Si nos interesa recorrerla, podemos hacer lo siguiente:

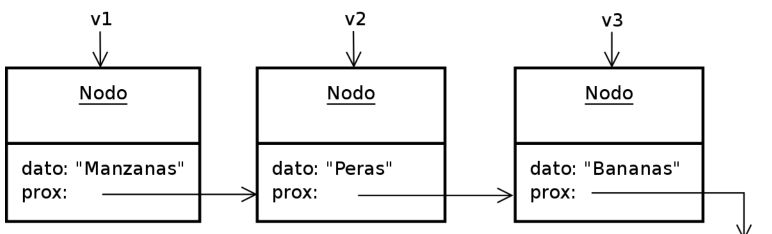


Figura 16.1: Nodos enlazados

```
def verLista(nodo):
    """ Recorre todos los nodos a través de sus enlaces,
        mostrando sus contenidos. """

    # cicla mientras nodo no es None
    while nodo:
        # muestra el dato
        print nodo
        # ahora nodo apunta a nodo.prox
        nodo = nodo.prox
```

```
>>> verLista(v1)
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 las cadena de referencias dentro de los nodos.

Si se desea *desenganchar* un vagón del medio de la lista, alcanza con cambiar el enganche:

```
>>> v1.prox=v3
>>> verLista(v1)
Manzanas
Bananas
>>> v1.prox = None
>>> verLista(v1)
Manzanas
```

De esta manera también se pueden generar estructuras impensables.

¿Qué sucede si escribimos `v1.prox = v1`? La representación es finita y sin embargo en este caso `verLista(v1)` no termina más. Hemos creado una *lista infinita*, también llamada *lista circular*.

16.1.1. Caminos

En una lista cualquiera, como las vistas antes, 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 `v1` a `v1`, como así también una flecha de `v1` a `v2` seguida de una flecha de `v2` a `v1`.

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 por programa, 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 `verLista(v1)`.

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 `v1`. Sin embargo puede suceder que querramos borrar a `v1` y continuar con el resto de la lista como la colección de nodos a tratar (en las listas de Python, **`del`** `lista[0]` no nos hace perder la referencia a `lista`).

Para ello lo que haremos 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:

```
>>> v3=Nodo("Bananas")
>>> v2=Nodo("Peras", v3)
>>> v1=Nodo("Manzanas", v2)
>>> lista=v1
>>> verLista(lista)
Manzanas
Peras
Bananas
```

Ahora sí estamos en condiciones de borrar el primer elemento de la lista sin perder la identidad de la misma:

```
>>> lista=lista.prox
>>> verLista(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 posea.

16.3. La clase `ListaEnlazada`

Basándonos en los nodos implementados anteriormente, pero buscando deslindar 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 mostrar la lista.
- `__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 borrar 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 podrán 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.

- Como vamos a incluir un método `__len__`, consideramos que no tiene sentido recorrer la lista cada vez que se lo llame, para contar cuántos elementos tiene, alcanza con agregar un atributo más (la longitud de la lista), que se inicializa en 0 cuando se crea la lista vacía, se incrementa en 1 cada vez que se agrega un elemento y se decrementa en 1 cada vez que se borra un elemento.

- 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, nombraremos a esa clase como `_Nodo`, y la dejaremos tal como hasta ahora.

> Se trata sólo de una convención, aún 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(object):
    " Modela una lista enlazada, compuesta de Nodos. "

    def __init__(self):
        """ Crea una lista enlazada vacía. """
        # prim: apuntará al primer nodo - None con la lista vacía
        self.prim = None
        # len: longitud de la lista - 0 con la lista vacía
        self.len = 0
```

Nuestra estructura ahora será como la representada por la Figura 16.3.1.

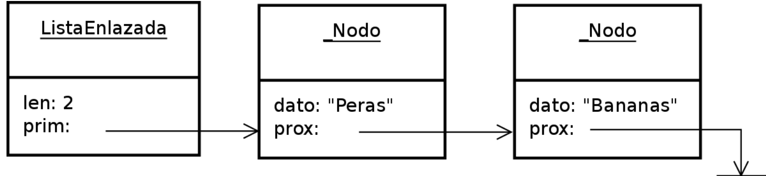


Figura 16.2: Una lista enlazada

Ejercicio 16.1. Escribir los métodos `__str__` y `__len__` para la lista.

Sabías que ...

Una característica importante de la implementación de lista enlazadas es que borrar el primer elemento es una operación de *tiempo constante*, es decir que no depende del largo de la lista, a diferencia de las listas de Python, en las que 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 borra 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 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 ValueError.

  Esto se resuelve con este fragmento de código:

```
# Verificación de los límites
if (i < 0) or (i >= self.len):
    raise IndexError("Índice fuera de rango")
```

- Si no se indica posición, `i` toma la última posición de la lista.

  Esto se resuelve con este fragmento de código:

```
# Si no se recibió i, se devuelve el último.
if i == None:
    i = self.len - 1
```

- Cuando la posición es 0 se trata de un caso particular, ya que en ese caso, además de borrar el nodo, 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).

  Esto se resuelve con este fragmento de código:

```
# Caso particular, si es el primero,
# hay que saltear la cabecera de la lista
if i == 0:
    dato = self.prim.dato
    self.prim = self.prim.prox
```

- Vemos ahora el caso general:

  Mediante un ciclo, se deben ubicar los nodos *npi - 1* y *npi* que están en las posiciones $i - 1$ e i de la lista, respectivamente, de modo de poder ubicar no sólo el nodo que se borrará,

sino también estar en condiciones de saltear el nodo borrado en los enlaces de la lista. La lista debe pasar de contener el camino *npi - 1* $\rightarrow$ *npi* $\rightarrow$ *npi.prox* a contener el camino *npi-1* $\rightarrow$ *npi.prox*.

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:

```
n_ant = self.prim
n_act = n_ant.prox
for pos in xrange(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 según era necesario:

```
# Guarda el dato y elimina el nodo a borrar
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 eliminado y decrementar la longitud en 1:

```
# hay que restar 1 de len
self.len -= 1
# y devolver el valor borrado
return dato
```

Finalmente, en el Código 16.1 se incluye el código completo del método `pop`.

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.

Código 16.1 `pop`: Método pop de la lista enlazada

```
1   def pop(self, i = None):
2       """ Elimina el nodo de la posición i, y devuelve el dato contenido.
3           Si i está fuera de rango, se levanta la excepción IndexError.
4           Si no se recibe la posición, devuelve el último elemento. """
5
6       # Verificación de los límites
7       if (i < 0) or (i >= self.len):
8           raise IndexError("Índice fuera de rango")
9
10      # Si no se recibió i, se devuelve el último.
11      if i == None:
12          i = self.len - 1
13
14      # Caso particular, si es el primero,
15      # hay que saltear la cabecera de la lista
16      if i == 0:
17          dato = self.prim.dato
18          self.prim = self.prim.prox
19
20      # Para todos los demás elementos, busca la posición
21      else:
22          n_ant = self.prim
23          n_act = n_ant.prox
24          for pos in xrange(1, i):
25              n_ant = n_act
26              n_act = n_ant.prox
27
28          # Guarda el dato y elimina el nodo a borrar
29          dato = n_act.dato
30          n_ant.prox = n_act.prox
31
32      # hay que restar 1 de len
33      self.len -= 1
34      # y devolver el valor borrado
35      return dato
```

- Los casos particulares son: la lista vacía, que es un error y hay que levantar una excepción; y el caso en el que x está en el primer nodo, en este caso hay que saltear el primer nodo desde la cabecera de la lista.

  El fragmento de código que resuelve estos casos es:

```
if self.len == 0:
    # Si la lista está vacía, no hay nada que borrar.
    raise ValueError("Lista vacía")

# Caso particular, x esta en el primer nodo
elif self.prim.dato == x:
    # Se descarta la cabecera de la lista
    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.

```
# Obtiene el nodo anterior al que contiene a x (n_ant)
n_ant = self.prim
n_act = n_ant.prox
while n_act != 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.

```
# Si no se encontró a x en la lista, levanta la excepción
if n_act == None:
    raise ValueError("El valor no está en la lista.")

# Si encontró a x, debe pasar de n_ant -> n_x -> n_x.prox
# a n_ant -> n_x.prox
else:
    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`.

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.

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:
            # Si la lista está vacía, no hay nada que borrar.
            raise ValueError("Lista vacía")

        # Caso particular, x esta en el primer nodo
        elif self.prim.dato == x:
            # Se descarta la cabecera de la lista
            self.prim = self.prim.prox

        # En cualquier otro caso, hay que buscar a x
        else:
            # Obtiene el nodo anterior al que contiene a x (n_ant)
            n_ant = self.prim
            n_act = n_ant.prox
            while n_act != None and n_act.dato != x:
                n_ant = n_act
                n_act = n_ant.prox

            # Si no se encontró a x en la lista, levanta la excepción
            if n_act == None:
                raise ValueError("El valor no está en la lista.")

            # Si encontró a x, debe pasar de n_ant -> n_x -> n_x.prox
            # a n_ant -> n_x.prox
            else:
                n_ant.prox = n_act.prox

        # Si no levantó excepción, hay que restar 1 del largo
        self.len -= 1
```

```
if (i > self.len) or (i < 0):
    # error
    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` sea `x`.

```
# Crea nuevo nodo, con x como dato:
nuevo = _Nodo(x)
```

- Si se quiere insertar en la posición 0, hay que cambiar la referencia de `self.prim`.

```
# Insertar al principio (caso particular)
if i == 0:
    # el siguiente del nuevo pasa a ser el que era primero
    nuevo.prox = self.prim
    # el nuevo pasa a ser el primero de la lista
    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.

```
# Insertar en cualquier lugar > 0
else:
    # Recorre la lista hasta llegar a la posición deseada
    n_ant = self.prim
    for pos in xrange(1,i):
        n_ant = n_ant.prox

    # Intercala nuevo y obtiene n_ant -> nuevo -> 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 cualquier caso, incrementar en 1 la longitud
self.len += 1
```

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. Como nuestro cliente no está satisfecho con esto debemos agregar un atributo más

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 > self.len) or (i < 0):
            # error
            raise IndexError("Posición inválida")

        # Crea nuevo nodo, con x como dato:
        nuevo = _Nodo(x)

        # Insertar al principio (caso particular)
        if i == 0:
            # el siguiente del nuevo pasa a ser el que era primero
            nuevo.prox = self.prim
            # el nuevo pasa a ser el primero de la lista
            self.prim = nuevo

        # Insertar en cualquier lugar > 0
        else:
            # Recorre la lista hasta llegar a la posición deseada
            n_ant = self.prim
            for pos in xrange(1,i):
                n_ant = n_ant.prox

            # Intercala nuevo y obtiene n_ant -> nuevo -> n_ant.prox
            nuevo.prox = n_ant.prox
            n_ant.prox = nuevo

        # En cualquier caso, incrementar en 1 la longitud
        self.len += 1
```

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. Hemos visto anteriormente 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 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 como se pidió en el ejercicio 16.4, se modifica la lista para que la inserción al final pueda hacerse en tiempo constante, 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, teniendo únicamente ese nodo, sin necesidad de saber también 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 . . .

Un ejemplo de uso de las listas circulares es dentro del kernel Linux. La mayoría de las listas utilizadas por este kernel son circulares, ya que la mayoría de los datos a los que se quiere acceder son datos que no tienen un orden en particular.
Por ejemplo, la lista de tareas que se están ejecutando es una lista circular. El *scheduler* del kernel permite que cada tarea utilice el procesador durante una porción de tiempo y luego pasa a la siguiente; y al llegar a la *última* vuelve a la *primera*, ya que la ejecución de tareas no se termina.

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.

En particular, en Python, los iteradores tienen que implementar 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`.

Es decir que las siguientes estructuras son equivalentes

```
for elemento in secuencia:
    # hacer algo con elemento

iterador = iter(secuencia)
while True:
    try:
        elemento = iterador.next()
    except StopIteration:
        break
    # hacer algo con elemento
```

En particular, 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(object):
    " Iterador para la clase ListaEnlazada "
    def __init__(self, prim):
        """ Constructor del iterador.
            prim es el primer elemento de la lista. """
        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 == None:
            raise StopIteration("No hay más elementos en la lista")
```

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*.

```
        # Guarda el dato
        dato = self.actual.dato
        # Avanza en la lista
        self.actual = self.actual.prox
        # Devuelve el dato
        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(object):
2     " Iterador para la clase ListaEnlazada "
3     def __init__(self, prim):
4         """ Constructor del iterador.
5             prim es el primer elemento de la lista. """
6         self.actual = prim
7
8     def next(self):
9         """ Devuelve uno a uno los elementos de la lista. """
10        if self.actual == None:
11            raise StopIteration("No hay más elementos en la lista")
12
13        # Guarda el dato
14        dato = self.actual.dato
15        # Avanza en la lista
16        self.actual = self.actual.prox
17        # Devuelve el dato
18        return dato
```

Finalmente, una vez que se tiene el iterador implementado, es necesario modificar la clase `ListaEnlazada` para que devuelva el iterador cuando se llama al método `__iter__`.

```
    def __iter__(self):
        " Devuelve el 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(1)
>>> l.append(3)
>>> l.append(5)
>>> for valor in l:
...     print valor
...
1
3
5
```

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.

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 (se describe también la acción que lleva adelante cada operación):

- `__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.
- `es_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:

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):

Atención

Al usar esa pila dentro de un programa, deberemos ignorar que se está trabajando sobre una lista: solamente podremos usar los métodos de pila.
Si alguien viola este principio, y usa la representación dentro del programa usuario, termina por recibir el peor castigo imaginable para un/a programador/a: sus programas pueden dejar de funcionar el cualquier momento, tan pronto como quien produce del TAD decida cambiar, aunque sea sutilmente, dicha representación.

```
class Pila:
    """ Representa una pila con operaciones de apilar, desapilar y
        verificar si está vacía. """

    def __init__(self):
        """ Crea una pila vacía. """
        # La pila vacía se representa con una lista vacía
        self.items=[]
```

El método `apilar` se implementará agregando el nuevo elemento al final de la lista:

```
    def apilar(self, x):
        """ Agrega el elemento x a la pila. """
        # Apilar es agregar al final de la lista.
        self.items.append(x)
```

Para implementar `desapilar`, se usará 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 levanta una excepción, haremos lo mismo, pero cambiaremos el tipo de excepción, para no revelar la implementación.

```
    def desapilar(self):
        """ Devuelve el elemento tope y lo elimina de la pila.
            Si la pila está vacía levanta una excepción. """
        try:
            return self.items.pop()
        except IndexError:
            raise ValueError("La pila está vacía")
```

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.

Finalmente, el método para indicar si se trata de una pila vacía.

```
    def es_vacia(self):
        """ Devuelve True si la lista está vacía, False si no. """
        return self.items == []
```

Construimos algunas pilas y operamos con ellas:

```
>>> from clasePila import Pila
>>> p = Pila()
```

```
>>> p.es_vacia()
True
>>> p.apilar(1)
>>> p.es_vacia()
False
>>> p.apilar(5)
>>> p.apilar("+")
>>> p.apilar(22)
>>> p.desapilar()
22
>>> p
<clasePila.Pila instance at 0xb7523f4c>
>>> q=Pila()
>>> q.desapilar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "clasePila.py", line 24, in desapilar
    raise ValueError("La pila está vacía")
ValueError: 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 (devolveremos `None`).

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
DEBUG: error pila sobran operandos
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "calculadora_polaca.py", line 64, in main
    print calculadora_polaca(elementos)
  File "calculadora_polaca.py", line 59, 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
DEBUG: error pila faltan operandos
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "calculadora_polaca.py", line 64, in main
    print calculadora_polaca(elementos)
  File "calculadora_polaca.py", line 37, in calculadora_polaca
    raise ValueError("Faltan operandos")
ValueError: Faltan operandos
```

Código 17.1 `calculadora_polaca.py`: Una calculadora polaca inversa

```
#!/usr/bin/env python
#encoding: latin1

from clasePila 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 "+-*/ %" or len(elemento) != 1:
                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 ValueError:
                print "DEBUG: error pila faltan operandos"
                raise ValueError("Faltan operandos")

            if elemento == "+":
                resultado = a2 + a1
            elif 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
    res = p.desapilar()
    if p.esPilaVacia():
        return res
    else:
        print "DEBUG: error pila sobran operandos"
        raise ValueError("Sobran operandos")

def main():
    expresion = raw_input("Ingrese la expresion a evaluar: ")
    elementos = expresion.split()
    print calculadora_polaca(elementos)
```

- 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 64, in main
    print calculadora_polaca(elementos)
  File "calculadora_polaca.py", line 26, 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  4.0
4.0
```

- (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
```

- `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: 6
DEBUG: apila  6.0
DEBUG: +
DEBUG: desapila  6.0
DEBUG: desapila  5.0
DEBUG: apila  11.0
DEBUG: *
DEBUG: desapila  11.0
DEBUG: desapila  4.0
DEBUG: apila  44.0
44.0
```

- `(4 + 5) * (3 + 8)`:

```
>>> calculadora_polaca.main()
Ingrese la expresion a evaluar: 4 5 + 3 8 + *
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: 3
DEBUG: apila  3.0
DEBUG: 8
DEBUG: apila  8.0
DEBUG: +
DEBUG: desapila  8.0
DEBUG: desapila  3.0
DEBUG: apila  11.0
DEBUG: *
DEBUG: desapila  11.0
DEBUG: desapila  9.0
DEBUG: apila  99.0
99.0
```

Ejercicio 17.1. Si se oprime la tecla <BACKSPACE>(o <←>) 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 de <BACKSPACE>, y no por eso hay que cancelar la ejecución de toda la función.

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 hubiera 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 `del self.items[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 *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

Todos sabemos lo que es una cola. Más aún, ¡estamos hartos de hacer colas!

El TAD *cola* modela precisamente ese comportamiento: el primero que llega es el primero en ser atendido, los demás 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.
- `es_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. """
        # La cola vacía se representa por una lista vacía
        self.items=[]
```

El método `encolar` se implementará agregando el nuevo elemento al final de la lista:

```
    def encolar(self, x):
        """ Agrega el elemento x como último de la cola. """
        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. """
        try:
          return self.items.pop(0)
        except:
          raise ValueError("La cola está vacía")
```

Por último, el método `es_vacia`, que indicará si la cola está o no vacía.

```
    def es_vacia(self):
        """ Devuelve True si la cola esta vacía, False si no."""
        return self.items == []
```

Veamos una ejecución de este código:

```
>>> from claseCola import Cola
>>> q = Cola()
>>> q.es_vacia()
True
>>> q.encolar(1)
>>> q.encolar(2)
>>> q.encolar(5)
>>> q.es_vacia()
False
>>> q.desencolar()
```

```
1
>>> q.desencolar()
2
>>> q.encolar(8)
>>> q.desencolar()
5
>>> q.desencolar()
8
>>> q.es_vacia()
True
>>> q.desencolar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "claseCola.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:
    """ Cola implementada sobre lista enlazada"""
    def __init__(self):
        """ Crea una cola vacía. """
        # La cola se representa por una lista enlazada vacía.
        self.items = claseListaEnlazadaConUlt.ListaEnlazada()
```

Sin embargo, una `Cola` es bastante más sencilla que una `ListaEnlazadaConUlt`, 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`.

```
def __init__(self):
    """ Crea una cola vacía. """
    # En el primer momento, tanto el primero como el último son None
    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):
    """ Agrega el elemento x como último de la cola. """
    nuevo = Nodo(x)
    # Si ya hay un último, agrega el nuevo y cambia la referencia.
    if self.ultimo:
        self.ultimo.prox = nuevo
        self.ultimo = nuevo
    # Si la cola estaba vacía, el primero es también el último.
    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):
    """ Elimina el primer elemento de la cola y devuelve su
        valor. Si la cola está vacía, levanta ValueError. """
    # Si hay un nodo para desencolar
    if self.primero:
        valor = self.primero.dato
        self.primero = self.primero.prox
        # Si después de avanzar no quedó nada, también hay que
        # eliminar la referencia del último.
        if not self.primero:
            self.ultimo = None
```

```
            return valor
        else:
            raise ValueError("La cola está vacía")
```

Finalmente, para saber si la cola está vacía, es posible verificar tanto si `self.primero` o `self.ultimo` referencian a `None`.

```
    def es_vacia(self):
        """ Devuelve True si la cola esta vacía, False si no."""
        return self.items == []
```

Una vez implementada toda la interfaz de la cola, podemos probar el TAD resultante

```
>>> from claseColaEnlazada import Cola
>>> q = Cola()
>>> q.es_vacia()
True
>>> q.encolar("Manzanas")
>>> q.encolar("Peras")
>>> q.encolar("Bananas")
>>> q.es_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 "claseColaEnlazada.py", line 42, in desencolar
    raise ValueError("La cola está vacía")
ValueError: La cola está vacía
```

Ejercicio 17.4. Este ejercicio surgió (y lo hicieron ya muchas generaciones de alumnos), haciendo cola:

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. Apéndice

A continuación el código completo de la pila y las colas implementadas en esta unidad.

Código 17.2 `clasePila.py`**:** Implementación básica de una pila

```
1 #! /usr/bin/env python
2 #encoding: latin1
3
4 class Pila:
5     """ Representa una pila con operaciones de apilar, desapilar y
6         verificar si está vacía. """
7
8     def __init__(self):
9         """ Crea una pila vacía. """
10        # La pila vacía se representa con una lista vacía
11        self.items=[]
12
13    def apilar(self, x):
14        """ Agrega el elemento x a la pila. """
15        # Apilar es agregar al final de la lista.
16        self.items.append(x)
17
18    def desapilar(self):
19        """ Devuelve el elemento tope y lo elimina de la pila.
20            Si la pila está vacía levanta una excepción. """
21        try:
22            return self.items.pop()
23        except IndexError:
24            raise ValueError("La pila está vacía")
25
26    def es_vacia(self):
27        """ Devuelve True si la lista está vacía, False si no. """
28        return self.items == []
```

Código 17.3 `claseCola.py`**: Implementación básica de una cola**

```
1  #! /usr/bin/env python
2  #encoding: latin1
3
4  class Cola:
5      """ Representa a una cola, con operaciones de encolar y
6          desencolar.  El primero en ser encolado es también el primero
7          en ser desencolado. """
8
9      def __init__(self):
10         """ Crea una cola vacía. """
11         # La cola vacía se representa por una lista vacía
12         self.items=[]
13
14     def encolar(self, x):
15         """ Agrega el elemento x como último de la cola. """
16         self.items.append(x)
17
18     def desencolar(self):
19         """ Elimina el primer elemento de la cola y devuelve su
20             valor. Si la cola está vacía, levanta ValueError. """
21         try:
22           return self.items.pop(0)
23         except:
24           raise ValueError("La cola está vacía")
25
26     def es_vacia(self):
27         """ Devuelve True si la cola esta vacía, False si no."""
28         return self.items == []
```

Código 17.4 `claseColaEnlazada.py`: Implementación de una cola enlazada

```
#! /usr/bin/env Python
#encoding: latin1

from claseNodo 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. """
        # En el primer momento, tanto el primero como el último son None
        self.primero = None
        self.ultimo = None

    def encolar(self, x):
        """ Agrega el elemento x como último de la cola. """
        nuevo = Nodo(x)
        # Si ya hay un último, agrega el nuevo y cambia la referencia.
        if self.ultimo:
            self.ultimo.prox = nuevo
            self.ultimo = nuevo
        # Si la cola estaba vacía, el primero es también el último.
        else:
            self.primero = nuevo
            self.ultimo = nuevo

    def desencolar(self):
        """ Elimina el primer elemento de la cola y devuelve su
            valor. Si la cola está vacía, levanta ValueError. """
        # Si hay un nodo para desencolar
        if self.primero:
            valor = self.primero.dato
            self.primero = self.primero.prox
            # Si después de avanzar no quedó nada, también hay que
            # eliminar la referencia del último.
            if not self.primero:
                self.ultimo = None
            return valor
        else:
            raise ValueError("La cola está vacía")

    def es_vacia(self):
        """ Devuelve True si la cola esta vacía, False si no."""
        return self.primero == None
```

Unidad 18

Modelo de ejecución de funciones y recursividad

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 hemos ido viendo desde el principio, sin embargo, nunca se explicó porqué 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, en este caso la función a la que pertenecen funciona como una especie de "apellido").

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*:

Paso	Pila	Salida
1. `g()`	g: (vacío)	
2. `x = 10`	g: x→ 10	
3. `b = 45`	g: x→ 10, b→ 45	
4. `print`	g: x→ 10, b→ 45	Imprime: `En g, antes de llamar a f, x vale 10`
5. `f()`	f: (vacío) g: x→ 10, b→ 45	
6. `x = 50`	f: x→ 50 g: x→ 10, b→ 45	
7. `a = 20`	f: x→ 50, a→ 20 g: x→ 10, b→ 45	
8. `print`	f: x→ 50, a→ 20 g: x→ 10, b→ 45	Imprime: `En f, x vale 50`
9. `print`	g: x→ 10, b→ 45	Imprime: `En g, despues de llamar a f, x vale 10`
10.	pila vacía	

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*.
- Cuando se ejecuta dentro de `g` la invocación `f()` (en 5) se *apila* un *marco* vacío que va a alojar las variables asociadas con `f` (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`, todo el tiempo el intérprete busca los valores que necesita usando el marco que está en el tope de la pila.

- 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 *se ve*.

18.2. Pasaje de parámetros

Un parámetro es un nombre 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:

Paso	Pila		Salida
1. `fun2(43)`	fun2	b →43	
2. `fun1(b+5)`	fun1	a →48	
	fun2	b →43	
3. `print`	fun1	a →48	Imprime:
	fun2	b →43	`49` (es decir 48+1)
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 llama():
    ls = [1,2,3,4]
    print ls
    modif(ls)
    print ls
```

Y esta es la ejecución:

```
>>> llama()
[1, 2, 3, 4]
[5, 2, 3, 4]
```

- Cuando se invoca a `modif(ls)` desde `llama`, el esquema de la pila es el siguiente:

  en `modif`: `lista` → | [1,2,3,4] |
  en `llama`: `ls` → |

- Cuando se modifica la lista desde `modif`, el esquema de la pila es el siguiente:

  en `modif`: `lista` → | [5,2,3,4] |
  en `llama`: `ls` → |

- Cuando la ejecución vuelve a `llama`, `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, el llamador no se entera:

```
def cambia_ref(lista):
    lista=[5,1,2,3,4]

def llama2():
    ls=[1,2,3,4]
    print ls
    cambia_ref(ls)
    print ls
```

```
>>> llama2()
[1, 2, 3, 4]
[1, 2, 3, 4]
```

- Cuando se invoca a `cambia_ref(ls)` desde `llama2`, el esquema de la pila es el siguiente:

  en `cambia_ref`: `lista` → | [1,2,3,4] |
  en `llama2`: `ls` → |

- Cuando se cambia referencia a la lista desde `cambia_ref`, el esquema de la pila es el siguiente:

  en `cambia_ref`: `lista` → | [5,1,2,3,4] |
  en `llama2`: `ls` → | [1,2,3,4] |

- Cuando la ejecución vuelve a `llama2`, `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 llamada, se *reciben* en la expresión correspondiente de la función llamadora.

```
def devuelve(valor):
    cuadrado = valor * valor
    return cuadrado

def recibe(valor):
    cuad = devuelve(valor+1)
    print cuad
```

En este caso, si hacemos el seguimiento de la llamada:

```
>>> recibe(5)
36
```

Veremos algo como lo siguiente:

Paso	Instrucción	Pila	
1.	`recibe(5)`	`recibe` | valor →5	
2.	`cuad = devuelve(valor+1)`	`recibe` | valor →5	Se suspende la ejecución. Se llama a `devuelve(6)`.
3.	`devuelve(6)`	`devuelve` | valor →6 `recibe` | valor →5	
4.	`cuadrado = valor * valor`	`devuelve` | valor →6, cuadrado →36 `recibe` | valor →5	

5.	En `devuelve(6)`:
	`return cuadrado`
	En `recibe(5)`:
	`cuad = devuelve(valor+1)`
		`recibe` | valor →5, cuad →36	

6.	`print cuad`	`recibe` | valor →5, cuad →36	Imprime: `36`.
7.		pila vacía	

Según se ve en el paso 5, al momento de devolver un valor, el valor de retorno correspondiente a la función `devuelve` es el que se asigna a la variable `cuad`, a la vez que la llamada a la función se elimina de la pila.

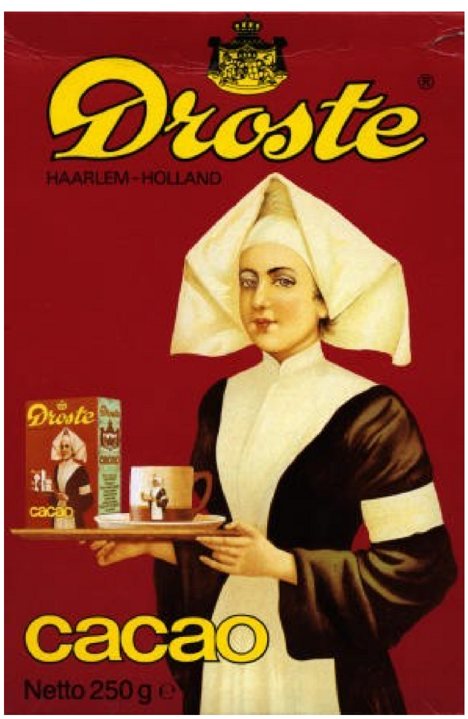


Figura 18.1: Una imagen recursiva: la publicidad de Cacao Droste, bajada de http://en.wikipedia.org/wiki/Image:Droste.jpg

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.

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:
$x! = x * (x-1)!$ si $x > 0$, $0! = 1$
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 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 corta las llamadas recursivas.
- 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 cada paso se asigna a una variable.

```
def factorial(n):
    """ Precondición: n entero >=0
        Devuelve: 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, el `factorial(3)` mediante la pila de evaluación:

Paso	Pila	Comentario
1. `factorial(3)`	`factorial` | n →3	
2. `if n == 0:`	`factorial` | n →3	
3. `f = factorial (n-1)`	`factorial` | n →3	Se suspende el cálculo. Se llama a `factorial(2)`.
4. `factorial(2)`	`factorial` | n →2 `factorial` | n →3	

Paso	Código	Pila	Nota
5.	`if n == 0:`	factorial: n →2 factorial: n →3	
6.	`f = factorial (n-1)`	factorial: n →2 factorial: n →3	Se suspende el cálculo. Se llama a `factorial(1)`.
7.	`factorial(1)`	factorial: n →1 factorial: n →2 factorial: n →3	
8.	`if n == 0:`	factorial: n →1 factorial: n →2 factorial: n →3	
9.	`f = factorial (n-1)`	factorial: n →1 factorial: n →2 factorial: n →3	Se suspende el cálculo. Se llama a `factorial(0)`.
10.	`factorial(0)`	factorial: n →0 factorial: n →1 factorial: n →2 factorial: n →3	
11.	`if n == 0:`	factorial: n →0 factorial: n →1 factorial: n →2 factorial: n →3	
12.	`r = 1`	factorial: n →0, r →1 factorial: n →1 factorial: n →2 factorial: n →3	

13.	En `factorial(0)`:
	    `return r`
	En `factorial(1)`:
	    `f = factorial (n-1)`
		factorial: n →1, f →1
		factorial: n →2
		factorial: n →3	

14.	`r = n * f`	factorial: n →1, f →1, r →1 factorial: n →2 factorial: n →3	

Paso	Código	Pila	

15.	En `factorial(1)`:
	`return r`
	En `factorial(2)`:
	`f = factorial (n-1)`
		`factorial` n →2, f →1
		`factorial` n →3	

16.	`r = n * f`	`factorial` n →2, f →1, r →2 `factorial` n →3	

17.	En `factorial(2)`:
	`return r`
	En `factorial(3)`:
	`f = factorial (n-1)`
		`factorial` n →3, f →2	

18.	`r = n * f`	`factorial` n →3, f →2, r →6	
19.	`return r`	pila vacía	Devuelve el valor 6

Figura 18.2: Otra imagen recursiva: captura de pantalla de RedHat, bajada de http://www.jfedor.org/

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 xrange(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 se trata de una única pila, en la cual se van modificando los valores de `num` y `fact`.

Es por esto que las versiones recursivas de los algoritmos, en general, utilizan más memoria (la pila del estado de las funciones se guarda en memoria) pero suelen ser más elegantes.

18.7. Un ejemplo de recursividad 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:

$b^n = b^{n/2} \times b^{n/2}$ Si n es par.
$b^n = b^{(n-1)/2} \times b^{(n-1)/2} \times b$ Si n 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 debe ser mayor o igual que cero.
        Devuelve: b^n. """

    # Caso base
    if n <= 0:
        return 1

    # n par
    if n % 2 == 0:
        pot = potencia(b, n/2)
        return pot * pot
    # n impar
    else:
        pot = potencia(b, (n-1)/2)
        return pot * pot * b
```

El uso de la variable `pot` 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:

	Llamada			
1.	`potencia(2,10)`			
2.	`pot = potencia(2,5)`	$b \to 2$	$n \to 10$	
3.	`pot = potencia(2,2)`	$b \to 2$	$n \to 5$	
4.	`pot = potencia(2,1)`	$b \to 2$	$n \to 2$	
5.	`pot = potencia(2,0)`	$b \to 2$	$n \to 1$	
6.	`return 1`	$b \to 2$	$n \to 0$	
7.	`return 1 * 1 * 2`	$b \to 2$	$n \to 1$	$pot \to 1$
8.	`return 2 * 2`	$b \to 2$	$n \to 2$	$pot \to 2$
9.	`return 4 * 4 * 2`	$b \to 2$	$n \to 5$	$pot \to 4$
10.	`return 32 * 32`	$b \to 2$	$n \to 10$	$pot \to 32$

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, 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 debe ser mayor o igual que cero.
        Devuelve: b^n. """

    pila = []
    while n > 0:
        if n % 2 == 0:
            pila.append(True)
            n /= 2
        else:
            pila.append(False)
```

```
            n = (n-1)/2

    pot = 1
    while pila:
        es_par = pila.pop()
        if es_par:
            pot = pot * pot
        else:
            pot = pot * pot * b

    return pot
```

Como se puede ver, este código es mucho más complejo que la versión recursiva, esto se debe a que utilizando recursividad 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 recursividad 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 recursividad tiene un resultado muy poco eficiente es el de los números de fibonacci. La sucesión de fibonacci está definida por la siguiente relación:

$$\begin{aligned} \texttt{fib(0)} &= 0 \\ \texttt{fib(1)} &= 1 \\ \texttt{fib(n)} &= \texttt{fib(n-1)} + \texttt{fib(n-2)} \end{aligned}$$

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 debe ser >= 0.
        Devuelve: el número de fibonacci número n. """
    if n == 0 or n == 1:
        return n
    return fib(n-1) + fib(n-2)
```

Sin embargo, si bien es muy sencillo y elegante, este código es extremadamente poco eficiente. Ya que para calcular `fib(n-1)` es necesario calcular `fib(n-2)`, que luego volverá a ser calculado para obtener el valor de `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 número 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.

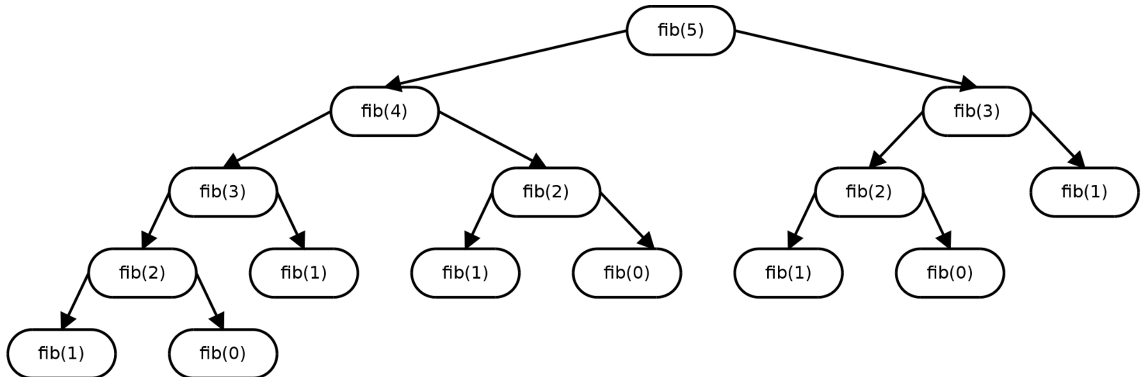


Figura 18.3: Árbol de llamadas para `fib(5)`

```
def fib(n):
    """ Precondición: n debe ser >= 0.
        Devuelve: el número de fibonacci número n. """
    if n == 0 or n == 1:
        return n

    ant2 = 0
    ant1 = 1
    for i in xrange(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.

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. 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
RuntimeError: 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.

Sabías que ...

Existen algunos lenguajes *funcionales*, como Haskell, ML, o Scheme, en los cuales la recursividad es la única forma de realizar un ciclo. Es decir, no existen construcciones `while` ni `for`.
Estos lenguajes cuentan con una optimización especial, llamada *optimización de recursión por cola* (*tail recursion optimization*), que permite que cuando una función realiza su llamada recursiva como **última** acción antes de terminar, no se apile el estado de la función innecesariamente, evitando el consumo adicional de memoria mencionado anteriormente.
La función `factorial` vista en esta unidad es un ejemplo de *recursión por cola* cuya ejecución puede ser optimizada por el compilador o intérprete del lenguaje.

18.10. Resumen

- A medida que se realizan llamadas a funciones, el estado de las funciones anteriores se almacena en una *pila* de llamadas a funciones.
- 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 llama 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 recursividad) 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.

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 (e incluso de acuerdo a una relación de orden que se desee, dada a través del parámetro `cmp`).

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

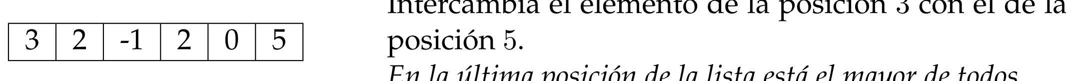
Éste método de ordenamiento se basa en la siguiente idea:

- **Paso 1.1**: Buscar el mayor de todos los elementos de la lista.

3	2	-1	5	0	2

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.

3	2	-1	2	0	5

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).

0	2	-1	2	3	5

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.

-1	0	2	2	3	5

La lista se encuentra ordenada.

La implementación en Python puede verse en el Código 19.1.

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.

Mientras que `buscar_max` es una función que ya se estudió previamente, que busca el mayor elemento 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]
>>> print l
[-1, 0, 2, 2, 3, 5]
>>> l=[]
>>> ord_seleccion(l)
>>> l=[1]
>>> ord_seleccion(l)
>>> print 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 aún 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.

Código 19.1 `seleccion.py`: Ordena una lista por selección

```
#/usr/bin/env python
#encoding: latin1

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. """

    # n = posicion final del segmento a tratar, comienza en len(lista)-1
    n = len(lista)-1

    # cicla mientras haya elementos para ordenar (2 o más elementos)
    while n > 0:

        # p es la posicion del mayor valor del segmento
        p = buscar_max(lista, 0, n)

        # intercambia 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

        # reduce el segmento en 1
        n = n - 1

def buscar_max(lista, ini, fin):
    """ Devuelve la posición del máximo elemento en un segmento de
        lista de elementos comparables.
        Se trabaja sobre lista, que no debe ser vacía.
        ini es la posición inicial del segmento, debe ser válida.
        fin es la posición final del segmento, debe ser válida. """

    pos_max = ini
    for i in xrange(ini+1, fin+1):
        if lista[i] > lista[pos_max]:
            pos_max = i
    return pos_max
```

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

$$T(N) \approx c * (2 + 3 + \ldots + N) \approx c * N * (N+1)/2 \sim N^2 \tag{19.1}$$

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

Éste otro método de ordenamiento se basa en la siguiente idea:

- **Paso 0**: Partimos de la misma lista de ejemplo utilizada para el ordenamiento por selección.

3	2	-1	5	0	2

- **Paso 1**: Considerar el segundo elemento de la lista, y ordenarlo respecto del primero, deplazándolo hasta la posición correcta, si corresponde.

2	3	-1	5	0	2

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.

-1	2	3	5	0	2

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.

-1	2	3	5	0	2

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**:

-1	0	2	3	5	2

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.

-1	0	2	2	3	5

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.

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]
>>> print l
[-1, 0, 2, 2, 3, 5]
>>> l=[]
>>> ord_insercion(l)
>>> l=[1]
>>> ord_insercion(l)
>>> print 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]
>>> print l
[1, 2, 3, 4, 5, 6]
```

Código 19.2 `seleccion.py`: Ordena una lista por selección

```
#!/usr/bin/env python
#encoding: latin1

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. """

    # i va desde la primera hasta la penúltima posición de la lista
    for i in xrange(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 es el valor a reubicar
    v = lista[p]

    # Recorre 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]:
            # Desplaza los elementos hacia la derecha, dejando lugar
            # para insertar el elemento v donde corresponda.
        lista[j] = lista[j-1]
            # Se mueve un lugar a la izquierda
        j -= 1

    # Ubica el valor v en su nueva posición
    lista[j] = v
```

19.2.1. Invariante del ordenamiento por inserción

En el ordenamiento por inserción, a 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...

$$T(N) \approx c * (2 + 3 + \ldots + N) \approx c * N * (N + 1)/2 \sim N^2 \qquad (19.2)$$

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:

$$T(N) \sim N \qquad (19.3)$$

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.

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 éstos 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*

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. 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` que realiza la intercalación ordenada de dos listas ordenadas (dadas dos listas ordenadas se debe obtener una nueva lista que resulte de intercalar a ambas de manera ordenada).

Diseñamos la **función `merge(lista1, lista2)`**:

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`:
        1) Agregar el elemento `i` de `lista1` al final del resultado.
        2) Avanzar el índice `i`.
    b) de lo contrario:
        1) Agregar el elemento `j` de `lista2` al final del resultado.
        2) 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.

20.2. ¿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 $a * N$ a ese tiempo.
- Si llamamos $T(N)$ al tiempo que tarda el algoritmo en ordenar una lista de longitud N, vemos que $T(N) = 2 * T(N/2) + a * N$.

Código 20.1 `mergesort.py`**: Una implementación de** *Merge sort*

```
 1 #!/usr/bin/env python
 2 #encoding: latin1
 3 
 4 def merge_sort(lista):
 5         """ Ordena lista mediante el método merge sort.
 6                 Pre: lista debe contener elementos comparables.
 7                 Devuelve: una nueva lista ordenada. """
 8 
 9         # Una lista de 1 o 0 elementos, ya está ordenada
10     if len(lista) < 2:
11         return lista
12     # Si tiene 2 o más elementos, se divide al medio y ordena cada parte
13     medio = len(lista) / 2
14     izq = merge_sort(lista[:medio])
15     der = merge_sort(lista[medio:])
16     return merge(izq, der)
17 
18 def merge(lista1, lista2):
19         """ Intercala los elementos de lista1 y lista2 de forma ordenada.
20             Pre: lista1 y lista2 deben estar ordenadas.
21                 Devuelve: una lista con los elementos de lista1 y lista2. """
22 
23     i, j = 0, 0
24     resultado = []
25 
26         # Intercalar ordenadamente
27     while(i < len(lista1) and j < len(lista2)):
28         if (lista1[i] < lista2[j]):
29             resultado.append(lista1[i])
30             i += 1
31         else:
32             resultado.append(lista2[j])
33             j += 1
34 
35         # Agregar lo que falta, si i o j ya son len(lista) no agrega nada.
36     resultado += lista1[i:]
37     resultado += lista2[j:]
38 
39     return resultado
```

- 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 $N = 2^k$.

$$
\begin{aligned}
T(N) = T(2^k) &= 2 * T(2^{k-1}) + a * 2^k && (20.1)\\
&= 2 * \left(2 * T(2^{k-2}) + a * 2^{k-1}\right) + a * 2^k && (20.2)\\
&= 2^2 * T(2^{k-2}) + a * 2^k + a * 2^k && (20.3)\\
&\vdots && (20.4)\\
&= 2^i * T(2^{k-i}) + i * a * 2^k && (20.5)\\
&\vdots && (20.6)\\
&= 2^k * T(1) + k * a * 2^k && (20.7)\\
&= b * 2^k + k * a * 2^k && (20.8)
\end{aligned}
$$

Pero si $N = 2^k$ entonces $k = \log_2 N$, y por lo tanto hemos demostrado que:

$$T(N) = bN + aN \log_2 N. \tag{20.9}$$

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.

Si analizamos el espacio que consume, vemos que a cada paso genera una nueva lista, de la suma de los tamaños de las dos listas, es decir que duplica el espacio consumido.

20.3. Ordenamiento rápido o *Quick sort*

Veremos ahora 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.

20.4. ¿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, para el algoritmo presentado), y en ese caso se comporta como *selección*.

En cambio, cuando la lista tiene números ubicados de manera arbitraria dentro de ella, podemos imaginar 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 Código 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*

```
1 #!/usr/bin/env python
2 #encoding: latin1
3
4 def quick_sort(lista):
5     """ Ordena la lista de forma recursiva.
6         Pre: los elementos de la lista deben ser comparables.
7         Devuelve: una nueva lista con los elementos ordenados. """
8
9     # Caso base
10    if len(lista) < 2:
11        return lista
12    # Caso recursivo
13    menores, medio, mayores = _partition(lista)
14    return quick_sort(menores) + medio + quick_sort(mayores)
15
16 def _partition(lista):
17    """ Pre: lista no vacÃa.
18        Devuelve: tres listas: menores, medio y mayores. """
19
20    pivote = lista[0]
21    menores = []
22    mayores = []
23    for x in xrange(1,len(lista)-1):
24        if lista[x] < pivote:
25            menores.append(lista[x])
26        else:
27            mayores.append(lista[x])
28    return menores, [pivote], mayores
```

20.5. 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 más parámetros, 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.

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 $T(N) \sim Nlog_2N$.

20.6. Resumen

- Los ordenamientos de selección e inserción, presentados en la unidad anterior son ordenamientos sencillos pero que 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 $NlogN$. 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 $NlogN$, en el peor caso (según cuál sea el pivote elegido) puede llegar a tardar N^2.

Código 20.3 `quicksort.py`**: Una versión más eficiente de** ***Quicksort***

```
#/usr/bin/env python
#encoding: latin1

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.
        """
    # Caso base
    if inicio >= fin:
        return

    # Caso recursivo
    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

        # Cambia de lugar los elementos
    for i in xrange(inicio+1, fin+1):
        if lista[i] < pivote:
            menores += 1
            if i != menores:
                _swap(lista, i, menores)
        # Pone el pivote al final de los menores
    if inicio != menores:
        _swap(lista, inicio, menores)
        # Devuelve la posición del pivote
        return menores

def _swap(lista, i, j):
        """ Intercambia los elementos i y j de lista. """
    lista[j], lista[i] = lista[i], lista[j]
```

Apéndice A

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 <marcosmedrano0@gmail.com>
Copyright © Nicolás Paez <nicopaez@gmail.com>

Esta obra está licenciada de forma dual, bajo las licencias Creative Commons:

- Atribución-Compartir Obras Derivadas Igual 2.5 Argentina
  http://creativecommons.org/licenses/by-sa/2.5/ar/
- Atribución-Compartir Obras Derivadas Igual 3.0 Unported
  http://creativecommons.org/licenses/by-sa/3.0/deed.es_AR.

A su criterio, puede utilizar una u otra licencia, o las dos. Para ver una copia de las licencias, puede visitar los sitios mencionados, o enviar una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Los íconos utilizados son parte del tema "Human", Copyright © Canonical Ltd, con licencia Licencia Atribución-Compartir Obras Derivadas Igual 2.5 Creative Commons.

El logo de Python es una marca registrada de la Python Software Foundation.