Apuntes
CC1002 Introducción a la Programación

Francisco Gutiérrez Figueroa
Vanessa Peña Araya
Mauricio Quezada Veas
Benjamin Bustos
Romain Robbes

Estos apuntes del curso CC1002 Introducción a la Programación están basados en los libros *How to Design Programs*, MIT Press, de M. Felleisen et al., y *Objects First with Java - A Practical Introduction using BlueJ*, Fifth Edition, Prentice Hall, de David J. Barnes y Michael Kölling.

Índice

Unidad I: Introducción a la Programación

Capítulo 1

Expresiones y Tipos de Datos Básicos

Estas notas pretenden ser un complemento a las cátedras dictadas en el contexto del curso CC1002 Introducción a la Programación, obligatorio para alumnos de Primer Año del Plan Común de Ingeniería y Ciencias, dictado por la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile.

El objetivo principal de este curso no es formar programadores, sino desarrollar en los alumnos una base común en razonamiento algorítmico y lógico, así como una capacidad de modelamiento y abstracción, necesarios para trabajar una habilidad general en la resolución de problemas. Estos problemas no necesariamente estarán acotados en el contexto de las Ciencias de la Computación, sino en el ámbito cotidiano y de la Ingeniería y Ciencias en general.

Los computadores son máquinas con un gran poder de cálculo y procesamiento. De hecho, los computadores fueron diseñados y construidos para poder realizar operaciones matemáticas muchísimo más rápido que un ser humano. Sin embargo, es un ser humano el que le tiene que decir al computador, de alguna forma, qué operaciones debe realizar. A esto se le denomina "programar". En este capítulo se introducirán los conceptos de algoritmo y programa, y se estudiará cómo programar el computador para que evalúe expresiones simples con distintos tipos de datos básicos.

1.1 ¿Qué es un algoritmo?

Un *algoritmo* es una secuencia finita de pasos que permiten ejecutar cualquier tarea (como por ejemplo, hacer un huevo frito). La palabra algoritmo viene de la transcripción latina del nombre de *Abu Abdallah Muhammad ibn Musa al-Khwarizmi*, un famoso matemático, astrónomo y geógrafo persa del siglo IX, padre del álgebra y quien introdujo el concepto matemático de algoritmo.

Podemos considerar que definir un algoritmo es la primera etapa en la resolución de un problema. Generalmente, procedemos como sigue:

- Identificar un problema
- Contextualizar los elementos que definen dicho problema
- Relacionar mediante pasos de ejecución los elementos para resolver el problema

Tal como vimos anteriormente, un algoritmo es la representación natural, paso a paso, de cómo podemos resolver un problema. Esto generalmente se conoce como una técnica de diseño *top-down*

(o de arriba hacia abajo). En otras palabras, partimos de un problema concreto y lo rompemos en unidades elementales que se pueden resolver paso a paso mediante alguna estrategia conocida de antemano.

Veamos a continuación un ejemplo: supongamos que queremos cocinar un huevo frito para acompañar un almuerzo. La definición del algoritmo que resuelve este problema sería:

- Problema: hacer un huevo frito
- Elementos: `huevo`, `aceite`, `cocina`, `fósforo`, `sartén`
- Pasos de ejecución:
  1. Encender un `fósforo`
  2. Con el `fósforo`, prender un quemador en la `cocina`
  3. Colocar la `sartén` sobre el quemador de la `cocina`
  4. Poner unas gotas de `aceite` sobre la `sartén`
  5. Tomar un `huevo` y quebrarlo
  6. Colocar el `huevo` quebrado sobre la `sartén`
  7. Esperar hasta que el `huevo` esté listo

1.2 ¿Qué es un programa?

Un *programa* es una especificación ejecutable de un algoritmo. Para representar un programa en un computador, utilizamos un *lenguaje de programación*. En el contexto de este curso, usaremos el lenguaje Python por su facilidad de aprendizaje.

Para comenzar a trabajar con Python utilizaremos su intérprete. El intérprete de Python es una aplicación que lee expresiones que se escriben, las evalúa y luego imprime en pantalla el resultado obtenido.

Es importante recalcar que utilizaremos en este curso a Python como una herramienta y no un fin. No hay que olvidar que el objetivo es aprender a resolver problemas, utilizando un computador como apoyo para realizar esta tarea.

1.3 Tipos de datos básicos

Un *tipo de datos* es un atributo que indica al computador (y/o al programador) algo sobre la clase de datos sobre los que se va a procesar. Esto incluye imponer restricciones en los datos, tales como qué valores pueden tomar y qué operaciones se pueden realizar. Todos los valores que aparecen en un programa tienen un tipo. A continuación, revisaremos algunos de los tipos de datos básicos con los que vamos a trabajar en este curso:

1.3.1 Enteros (`int`)

El tipo de datos *entero* representa un subconjunto finito de los números enteros. El número mayor que puede representar depende del tamaño del espacio usado por el dato y la posibilidad (o no) de representar números negativos. Las típicas operaciones aritméticas que se pueden realizar con estos datos son: suma, resta, multiplicación y división. Por ejemplo: `...`, `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `...`

1.3.2 Reales (`float`)

El tipo de datos *real* permite representar una aproximación decimal de números reales. La aproximación depende de los recursos disponibles del computador, por lo que todas las operaciones entre valores reales son aproximaciones. Los números reales se escriben separando la parte entera de la parte decimal con un punto. Por ejemplo: `0.303456`

1.3.3 Texto (`str`)

El tipo de datos *texto* (o *string*) permite representar cadenas de caracteres indicadas entre comillas simples (‘ ’) o dobles (“ ”). Por ejemplo: `‘hola’`, `‘103’`, `‘Mi mascota es un ñandú!’`. Al respecto, hay que recalcar que la expresión `‘103’` es de tipo texto porque está entre comillas, aún cuando su contenido se puede entender como un número.

1.4 Programas simples

Como vimos, un programa es una representación ejecutable de un algoritmo. Esta representación está definida por una expresión que al ser evaluada genera un valor. A continuación veremos cómo definir y evaluar programas simples.

1.4.1 Evaluación de expresiones

Una *expresión* es una combinación entre valores y operadores que son evaluados durante la ejecución de un programa. Por ejemplo, la expresión `1 + 1` es una expresión aritmética, tal que al evaluarla produce el valor `2`.

Con los tipos de datos explicados anteriormente, podemos realizar operaciones entre ellos utilizando operadores específicos en cada caso. Así, para datos numéricos (enteros y reales), podemos usar los operadores de suma (`+`), resta (`-`), multiplicación (`*`) y división (`/`). La prioridad de estos operadores es la misma usada en álgebra: primero se evalúan los operadores multiplicativos de izquierda a derecha según orden de aparición (`*` y `/`), y luego los aditivos (`+`, `-`). En el caso de querer imponer una evaluación en particular que no siga el orden preestablecido, podemos indicarlo utilizando paréntesis. Por ejemplo:

```
>>> 3 + 5
 -> 8
>>> 3 + 2 * 5
 -> 13
>>> (3 + 2) * 5
 -> 25
```

Nota: La secuencia de caracteres `>>>` significa que usamos el intérprete de Python. La línea siguiente, con la secuencia de caracteres `->` indica la respuesta del intérprete, tal como Python la evalúa.

En Python, se definen dos operadores adicionales para operaciones matemáticas recurrentes: elevar a potencia (`**`) y calcular el resto de una división entera (`%`). La prioridad del operador `%` es la misma que la de los operadores multiplicativos, mientras que la del operador `**` es mayor. Así:

```
>>> 2 ** 3
-> 8
>>> 4 ** 0.5
-> 2.0
```

```
>>> 10 % 3
-> 1
```

Para operar con valores de tipo texto, en Python utilizamos generalmente dos operadores: si queremos unir (concatenar) dos cadenas de texto, lo indicamos con el operador +; por otro lado, si queremos repetir una cadena de texto, lo indicamos con el operador *. Por ejemplo:

```
>>> 'abra' + 'cadabra'
 -> 'abracadabra'
>>> 'ja' * 3
 -> 'jajaja'
```

Finalmente, es importante notar que en Python los valores a los que se evalúa una expresión dependen del tipo de los operandos. Así, si la expresión está compuesta únicamente de números enteros, el resultado obtenido también será de tipo entero. Por ejemplo:

```
>>> 1 / 2
 -> 0
```

En efecto, dado que 1 y 2 son valores de tipo entero, la división entre ellos también lo será (y, por ende, se transforma el valor obtenido al tipo entero). Si queremos calcular el valor real de dicha operación, debemos forzar a que al menos uno de los dos operandos sea real. Así:

```
>>> 1.0 / 2.0
 -> 0.5
>>> 1 / 2.0
 -> 0.5
>>> 1.0 / 2
 -> 0.5
```

Finalmente, si queremos juntar valores de tipo texto con valores de tipo numérico, debemos convertir estos últimos previamente a valores de tipo texto. Por ejemplo, si queremos transformar un valor n a un valor equivalente de tipo texto, utilizamos la función de Python `str`. De igual manera, si tenemos un valor de tipo texto que se puede entender como número, por ejemplo `'103'`, podemos convertir el valor en tipo numérico usando las funciones `int` y `float`. Así:

```
>>> 'En el curso hay ' + str(100) + ' alumnos'
 ->  'En el curso hay 100 alumnos'
>>> '100' + '1'
 ->  '1001'
>>> int('100') + 1
 -> 101
```

1.4.2 Variables

Una *variable* es el nombre que se le da a un valor o a una expresión. El valor de una variable está dado por la definición más reciente del nombre. Para evaluar una expresión con variables, usamos una semántica por sustitución, esto es, reemplazamos en la expresión los valores asociados al nombre por aquellos que están definidos por la variable.

Por ejemplo, si la variable `x` tiene el valor `8`, al evaluar la expresión `doble = 2 * x`, entonces la variable `doble` contendrá el valor `16` (pues `doble` → `2 * x` → `2 * 8` → `16`).

Para crear variables en un programa podemos utilizar cualquier letra del alfabeto, o bien, una combinación de letras, números y el símbolo _ siempre que el primer carácter no sea un número. Para *asignar* una variable a una expresión (o al resultado de ésta), utilizamos el operador `=`.

Notemos que es importante el orden en que se realiza la definición de variables y expresiones: la sintaxis correcta es `variable = expresión`, y no al revés. En Python se evalúa la expresión y se define la variable con el valor resultante. Por ejemplo:

```
>>> a = 8 # la variable a contiene el valor 8
>>> b = 12 # la variable b contiene el valor 12
>>> a # mostramos el valor de a
 -> 8
>>> a + b # creamos una expresion y mostramos su valor
 -> 20
>>> c = a + 2 * b # creamos una expresion, se evalua y se define c
>>>               # con su valor
>>> c # mostramos el valor de c
 -> 32
>>> a = 10 # redefinimos a
>>> a + b
 -> 22
>>> c # el valor de c no cambia, pues lo calculamos antes de la
>>>   # redefinicion de a
 -> 32
```

En Python, el símbolo `#` sirve para introducir un comentario. Los comentarios son explicaciones que da el programador y que no son procesadas por el intérprete. Es importante utilizar comentarios para introducir aclaraciones relevantes en el código. Por otro lado, no es recomendable abusar de ellos! Es mejor utilizar nombres de variables que tengan un significado propio (relevante a la expresión que están calculando), y utilizar los comentarios sólo en situaciones especiales.

Veamos un mal ejemplo:

```
>>> a = 8
>>> b = 12
>>> c = a * b
```

En este ejemplo notamos que queremos calcular el área de un rectángulo, pero los nombres de las variables no son los indicados. En este caso, las variables `a`, `b` y `c` pueden representar cualquier cosa. Sería mucho más adecuado escribir:

```
>>> ancho = 8
>>> largo = 12
>>> area = ancho * largo
```

1.5 Errores

Hasta ahora hemos visto únicamente expresiones correctas, tal que al evaluarlas obtenemos un resultado. ¿Qué es lo que imprime el intérprete en este caso?

```
>>> dia = 13
>>> mes = 'agosto'
>>> 'Hoy es ' + dia + ' de ' + mes
```

```
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError:  cannot concatenate 'str' and 'int' objects
```

En este caso, el intérprete nos dice que hemos cometido un error en nuestro programa: "`cannot concatenate 'str' and 'int' objects`", y es de tipo `TypeError`. Esto en español significa que estamos intentando unir valores de tipo texto con valores de tipo entero. Tal como lo vimos anteriormente, esto es incompatible para el intérprete por lo que debemos corregir la instrucción.

```
>>> dia = 13
>>> mes = 'agosto'
>>> 'Hoy es ' + str(dia) + ' de ' + mes
 -> 'Hoy es 13 de agosto'
```

Existen distintos tipos de errores. Es útil conocerlos para no cometerlos, y eventualmente para saber cómo corregirlos.

1.5.1 Errores de sintaxis

Un *error de sintaxis* se produce cuando el código no sigue las especificaciones propias del lenguaje. Estos errores se detectan en el intérprete antes de que se ejecute, y se muestran con un mensaje de error indicando el lugar "aproximado" en el que se detectó la falta. Por ejemplo:

```
>>> numero = 15
>>> antecesor = (numero - 1))
```

```
File "<stdin>", line 1
   antecesor = (numero - 1))
                           ^
SyntaxError:  invalid syntax
```

1.5.2 Errores de nombre

Un *error de nombre* se produce cuando utilizamos una variable que no se ha definido anteriormente en el programa. Por ejemplo:

```
>>> lado1 = 15
>>> area = lado1 * lado2
```

```
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
NameError:  name 'lado2' is not defined
```

1.5.3 Errores de tipo

Un *error de tipo* ocurre cuando aplicamos operaciones sobre tipos que no son compatibles. Por ejemplo:

```
>>> dia = 13
>>> mes = 'agosto'
>>> 'Hoy es ' + dia + ' de ' + mes
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

1.5.4 Errores de valor

Un *error de valor* ocurre cuando, a pesar de tener una expresión bien formada, se aplican operaciones sobre valores que no corresponden al tipo en el que fueron definidas. Por ejemplo:

```
>>> nombre = 'Juan Soto'
>>> int(nombre)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Juan␣Soto'
```

1.5.5 Errores de indentación

Un *error de indentación* ocurre cuando los espacios en el código no son consistentes. Esto es porque Python usa los espacios para delimitar *bloques de código*, como veremos en el próximo capítulo.

```
>>> x = 3
>>>     x
```

```
  File "<stdin>", line 1
    x
    ^
IndentationError: unexpected indent
```

1.5.6 Errores lógicos

En un programa no todos los errores pueden ser detectados por el computador. Los *errores lógicos* son aquellos que se producen por descuido del programador al escribir las instrucciones y pueden provocar resultados muy inesperados. Por ejemplo, estos errores se pueden producir al darle un nombre incorrecto o ambiguo a una variable o a otras situaciones más complejas.

Lamentablemente, el intérprete no nos indicará cuándo o en qué línea se producen estos errores, por lo que la mejor manera de enfrentarlos es evitarlos y seguir una metodología limpia y robusta que nos permita asegurar a cabalidad que lo que estamos escribiendo efectivamente es lo que esperamos que el computador ejecute. Por ejemplo:

```
>>> numero = 15
>>> doble = 3 * numero
>>> doble # esperariamos 30, luego debe haber algun error en el codigo
 -> 45
```

Capítulo 2

Funciones[1]

La clase anterior vimos cómo crear expresiones sencillas que operan con números. El objetivo de esta clase es ir un paso más allá y desarrollar pequeños trozos de código que implementen operaciones como un conjunto. Al igual que en matemática, en computación llamamos *función* a una estructura que recibe valores de entrada y genera un valor de salida.

2.1 Variables y funciones

En los cursos de matemática aprendimos a relacionar cantidades mediantes expresiones con variables. Por ejemplo, conocemos bien la relación entre el área de un círculo y su radio. Si el radio de un círculo está dado por la variable "r", entonces su área se calcula mediante la expresión:

$$areaCirculo = 3.14 \cdot r^2$$

Así, si tenemos un círculo cuyo radio tiene valor igual a 5, sabemos que sustituyendo la variable "r" con este valor, obtenemos su área:

$$areaCirculo = 3.14 \cdot 5^2 = 3.14 \cdot 25 = 78.5$$

2.1.1 Definición de funciones

Al igual que en matemática, en computación una función es una regla que cumple con las mismas características, puesto que describe cómo producir información en base a otra que está dada como entrada. Luego, es importante que cada función sea nombrada de forma en que sea clara la relación entre su nombre y el objetivo que cumple. El ejemplo anterior puede transformarse en una función cuyo nombre sería `areaCirculo` y en Python estaría definida como sigue:

```
1 def areaCirculo(radio):
2     return 3.14 * radio ** 2
```

La primera línea define la función, asignándole un nombre y los argumentos que recibe. Notemos que para declarar una función debemos utilizar la palabra clave `def` y terminar la declaración con el símbolo dos puntos (`:`). Los argumentos de una función son la información de entrada que ésta recibe, y que serán utilizados para evaluar las expresiones que la definen. En nuestro ejemplo, la función recibe un sólo argumento, que es el valor del radio del círculo que será utilizado para calcular su área.

[1] Traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

La segunda línea define la salida de la función. La palabra clave `return` indica el fin de la función, y la expresión que la sucede será evaluada y retornada como la salida.

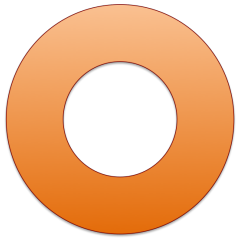
Para utilizar una función, debemos invocarla con el valor de sus argumentos. En nuestro ejemplo, para calcular el área de un círculo con radio igual a 5, invocamos a la función de la siguiente manera:

```
>>> areaCirculo(5)
 -> 78.5
```

Internamente, lo que sucede es similar al reemplazo de variables que se explicó anteriormente. La última línea de la función es evaluada de la siguiente manera:

$$\texttt{areaCirculo(5)} \rightarrow \texttt{3.14 * 5 ** 2} \rightarrow \texttt{3.14 * 25} \rightarrow \texttt{78.5}$$

Una función puede estar compuesta tanto de operadores básicos como también de otras funciones previamente definidas. Es decir, dada la definición de una función, ésta puede ser utilizada a su vez por otras funciones. Por ejemplo, imaginemos que queremos definir una función que calcule el área de un anillo. Dado que el área de un anillo se calcula restando área del círculo externo con el área del círculo interno, podemos utilizar nuestra función `areaCirculo` para implementar esta nueva función.



Para esto, debemos crear una función que reciba dos argumentos, el radio del círculo externo y el radio del círculo interno, calcule el área de ambos círculos y finalmente los reste. Luego, la función que calcula el área de un anillo queda definida de la siguiente manera:

```
def areaAnillo(exterior, interior):
      return areaCirculo(exterior) - areaCirculo(interior)
```

Para utilizar nuestra función, podemos tomar como ejemplo un anillo con radio externo igual a 5 e interno igual a 3:

```
>>> areaAnillo(5, 3)
 -> 50.24
```

En efecto, esta evaluación corresponde a:

$$\begin{aligned}\texttt{areaAnillo(5, 3)} &\rightarrow \texttt{areaCirculo(5) - areaCirculo(3)}\\ &\rightarrow \texttt{3.14 * 5 ** 2 - 3.14 * 3 ** 2}\\ &\rightarrow \texttt{3.14 * 25 - 3.14 * 9}\\ &\rightarrow \texttt{50.24}\end{aligned}$$

2.1.2 Indentación y subordinación de instrucciones

Es importante notar que en Python la indentación (cantidad de tabulaciones hacia la derecha) de cada instrucción determina el alcance al que pertenece. En el caso de nuestra función de ejemplo `areaCirculo`, podemos ver que la primera línea está al margen izquierdo mientras que la segunda está

separado del margen izquierdo por una tabulación. Esto indica que la segunda línea está subordinada a la función, lo que se traduce que esta instrucción es parte de la ella. Así, si una función está compuesta por muchas instrucciones, cada línea debe esta indentada al menos una tabulación hacia la derecha más que la línea que indica el nombre y argumentos de la función. Notemos entonces que la función de nuestro ejemplo puede ser reescrita de la siguiente manera:

```
1 def areaCirculo(radio):
2     pi = 3.14
3     return pi * radio * radio
```

Como se puede observar, las dos líneas que definen esta función tienen una indentación a la derecha, quedando ambas subordinadas a la función misma. En caso de no respetar esta regla, una error de indentación ocurre:

```
1 def areaCirculo(radio):
2     pi = 3.14
3         return pi * radio * radio
```

```
File "<stdin>", line 3
    return pi * radio * radio
    ^
IndentationError: unexpected indent
```

O, al olvidar de indentar:

```
1 def areaCirculo(radio):
2 pi = 3.14
3 return pi * radio * radio
```

```
File "<stdin>", line 2
    pi = 3.14
     ^
IndentationError: expected an indented block
```

2.1.3 Alcance de una variable

La definición de una variable dentro de una función tiene un alcance local. El significado de esta frase se explicará a continuación a través de ejemplos.

Imaginemos que declaramos una variable `a`. Si una función realiza alguna operación que requiere de esta variable, el valor utilizado será aquel que contiene la variable. Por ejemplo:

```
1 >>> a = 100
2 >>> def sumaValorA(x):
3 ...     return x + a
4 >>> sumaValorA(1)
5  -> 101
```

Sin embargo, una variable puede ser redefinida dentro de una función. En este caso, cada vez que se deba evaluar una expresión dentro de la función que necesite de esta variable, el valor a considerar será aquel definido dentro de la función misma. Además, la redefinición de una variable se hace de manera local, por lo que no afectará al valor de la variable definida fuera de la función. Esto se puede observar en el siguiente ejemplo:

```
>>> a = 100
>>> def sumaValorA(x):
...     a = 200
...     return x + a
>>> sumaValorA(1)
 -> 201
>>> a
 -> 100
```

Por otra parte, si el argumento de una función tiene el mismo nombre que una variable definida fuera de ésta, la función evaluará sus instrucciones con el valor del argumento, pues es la variable que está dentro de su alcance:

```
>>> a = 100
>>> def sumaValorA(a):
...     return 1 + a
>>> sumaValorA(5)
 -> 6
```

Finalmente, cualquier variable definida dentro de la función no existe fuera de ésta, ya que queda fuera de su alcance (recuerde que éste es local a la función). Por ejemplo:

```
>>> def sumaValorA(a):
        b = 100
        return a + b

>>> sumaValorA(50)
150
>>> b
```

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

2.2 Problemas

Rara vez los problemas vienen formulados de tal manera que basta con traducir una fórmula matemática para desarrollar una función. En efecto, típicamente se tiene una descripción informal sobre una situación, la que puede incluir información ambigua o simplemente poco relevante. Así, la primera etapa de todo programador es extraer la información relevante de un problema y luego traducirlo en expresiones apropiadas para poder desarrollar un bloque de código. Consideremos el siguiente ejemplo:

"Genera S.A. le paga $4.500 por hora a todos sus ingenieros de procesos recién egresados. Un empleado típicamente trabaja entre 20 y 65 horas por semana. La gerencia de informática le pide desarrollar un programa que calcule el sueldo de un empleado a partir del número de horas trabajadas."

En la situación anterior, la última frase es la que indica cuál es el problema que queremos resolver: escribir un programa que determine un valor en función de otro. Más específicamente, el programa recibe como entrada un valor, la cantidad de horas trabajadas, y produce otro, el sueldo de un

empleado en pesos. La primera oración implica cómo se debe calcular el resultado, pero no lo especifica explícitamente. Ahora bien, en este ejemplo particular, esta operación no requiere mayor esfuerzo: si un empleado trabaja una cantidad h de horas, su sueldo será: $\$4.500 \cdot h$.

Ahora que tenemos una expresión para modelar el problema, simplemente creamos una función en Python para calcular los valores:

```
1 def sueldo(h):
2     return 4500 * h
```

En este caso, la función se llama `sueldo`, recibe un parámetro `h` representando a la cantidad de horas trabajadas, y devuelve `4500 * h`, que corresponde al dinero que gana un empleado de la empresa al haber trabajado `h` horas.

2.3 Un poco más sobre errores

En el capítulo anterior se habló acerca de los distintos tipos de error que se pueden producir al generar programas de computación. En particular, se discutió sobre los errores de sintaxis, de nombre y lógicos. Ahora se discutirá sobre dos nuevos tipos de error: los errores en tiempo de ejecución y los errores de indentación.

2.3.1 Errores de ejecución

Existen maneras de cometer errores que el intérprete de Python no notará hasta que la expresión escrita sea evaluada. Un claro ejemplo es la división por cero. Para el intérprete de Python la expresión `1 / 0` representa la división entre dos números cualquiera, pero que al evaluarla se generará el error descrito a continuación:

```
1 >>> 1 / 0
```

```
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

En Python este tipo de error se llama `ZeroDivisionError`, la que indica claramente la fuente de la falla.

Otra manera de obtener este tipo de error es invocando una función con un número equivocado de argumentos. Por ejemplo, si utilizamos la función `areaCirculo` con dos argumentos en vez de uno, recibiremos un mensaje de error que lo indica:

```
1 >>> areaCirculo(5,3)
```

```
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError:  areaCirculo() takes exactly 1 argument (2 given)
```

2.3.2 Errores de indentación

Como vimos anteriormente, la indentación de un conjunto de instrucciones en Python tiene una connotación semántica, no sólo sintáctica, puesto que indica la subordinación de una instrucción. Así, es posible generar errores cuando una instrucción no está indentada de manera correcta.

Imagine que se desea definir una función que calcule el área de un cuadrado dado el largo de uno de sus lados. Una implementación de esta función podría ser de la siguiente manera:

```
1 >>> def areaCuadrado(lado):
2 ... return lado * lado
```

Sin embargo, al intentar evaluar la expresión anterior, se obtiene un mensaje de error que indica que se espera una indentación de la instrucción para poder definir la función de manera correcta:

```
File "<stdin>", line 2
   return lado * lado
          ^
IndentationError:  expected an indented block
```

Se nota que este error de indentación tiene un mensage diferente que el error que vimos anteriormente (“`IndentationError:  unexpected indent`”), dado que el error de indentación es distinto.

Capítulo 3

Receta de Diseño[1]

En el capítulo anterior vimos que el desarrollo de una función requiere varios pasos. Necesitamos saber qué es lo relevante en el enunciado del problema y qué podemos ignorar. Además, necesitamos saber qué es lo que la función recibirá como parámetros, y cómo relaciona estos parámetros con la salida esperada. Además, debemos saber, o averiguar, si Python provee operaciones básicas para manejar la información que necesitamos trabajar en la función. Si no, deberíamos desarrollar funciones auxiliares que implementen dichas operaciones. Finalmente, una vez que tengamos desarrollada la función, necesitamos verificar si efectivamente realiza el cálculo esperado (para el cual efectivamente implementamos la función). Esto puede evidenciar errores de sintaxis, errores de ejecución, o incluso errores de diseño.

Para trabajar apropiadamente, lo mejor es seguir una *receta de diseño*, esto es, una descripción paso a paso de qué es lo que tenemos que hacer y en qué orden. Basándonos en lo que hemos visto hasta ahora, el desarrollo de un programa requiere al menos las siguientes cuatro actividades:

1. Entender el propósito de la función.
2. Dar ejemplos de uso de la función.
3. Probar la función.
4. Especificar el cuerpo de la función.

En las siguientes secciones estudiaremos en detalle cada una de estas cuatro actividades.

3.1 Entender el propósito de la función

El objetivo de diseñar una función es el crear un mecanismo que consume y produce información. Luego, deberíamos empezar cada función dándole un nombre significativo y especificando qué tipo de información consume y qué tipo de información produce. A esto lo llamamos *contrato*. Por ejemplo, supongamos que nos piden diseñar una función que calcule el área de un rectángulo. Supongamos que la función se llama `areaRectangulo`. Su contrato se define como:

```
1 # areaRectangulo: num num -> num
```

[1] Traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

El contrato consiste en dos partes: la primera, a la izquierda de los dos puntos especifica el nombre de la función; la segunda, a la derecha de los dos puntos, especifica qué tipo de datos consume y qué es lo que produce. Los tipos de valores de entrada se separan de los de salida por una flecha. En el caso de nuestro ejemplo el tipo de datos que consume es de tipo numérico, es decir, puede ser de tipo entero (`int`) o real (`float`), por lo que lo representamos con la palabra `num`. El valor que se producetambién es de tipo numérico, dado que es de tipo entero si es que ambos datos de entrada son enteros, o es de tipo real si es que al menos uno de los datos de entrada es un número real. En general, representaremos los tipos de datos que conocemos hasta el momento como sigue (se irán agregando otros a lo largo del curso):

- Tipo entero: se representa con la palabra `int`.
- Tipo real: se representa con la palabra `float`.
- Tipo numérico (real o entero): se representa con la palabra `num`.
- Tipo texto: se representa con la palabra `str`.

Por ejemplo, para la función `areaAnillo` del capítulo anterior, su contrato es:

```
1 # areaAnillo: num num -> float
```

dado que los datos de entrada son numéricos (enteros o reales), pero el dato de salida siempre será de tipo `float`.

Una vez que tenemos especificado el contrato, podemos agregar el *encabezado* de la función. Éste reformula el nombre de la función y le da a cada argumento un nombre distintivo. Estos nombres son variables y se denominan los *parámetros* de la función. Miremos con más detalle el contrato y el encabezado de la función `areaRectangulo`:

```
1 # areaRectnagulo:  num num -> num
2 def areaRectangulo(largo, ancho):
3     ...
```

Aquí especificamos que los parámetros (en orden) que recibe la función se llaman `largo` y `ancho`.

Finalmente, usando el contrato y los parámetros, debemos formular un *propósito* para la función, esto es, un comentario breve sobre qué es lo que la función calcula. Para la mayoría de nuestras funciones, basta con escribir una o dos líneas; en la medida que vayamos desarrollando funciones y programas cada vez más grandes, podremos llegar a necesitar agregar más información para explicar el propósito de una función. Así, hasta el momento llevamos lo siguiente en la especificación de nuestra función:

```
1 # areaRectangulo:  num num -> num
2 # calcula el area de un rectangulo de medidas
3 # largo y ancho
4 def areaRectangulo(largo, ancho):
5     ...
```

3.2 Dar ejemplos de uso de la función

Para tener un mejor entendimiento sobre qué es lo que debe calcular la función, evaluamos ejemplos para valores de entrada significativos y determinamos manualmente cuál debe ser la salida esperada. Por ejemplo, la función `areaRectangulo` debe generar el valor `15` para las entradas `5` y `3`. Así, la especificación de nuestra función queda de la forma:

```
# areaRectangulo:  num num -> num
# calcula el area de un rectangulo de medidas
# largo y ancho
# ejemplo: areaRectangulo(5, 3) debe producir 15
def areaRectangulo(largo, ancho):
    ...
```

El hacer ejemplos ANTES DE ESCRIBIR EL CUERPO DE LA FUNCIÓN ayuda de muchas maneras. Primero, es la única manera segura de descubrir errores lógicos. Si usáramos la función una vez implementada para generar estos ejemplos, estaríamos tentados a confiar en la función porque es mucho más fácil evaluar la función que predecir qué es lo que efectivamente hace. Segundo, los ejemplos nos fuerzan a pensar a través del proceso computacional, el que, para casos más complejos que veremos más adelante, es crítico para el desarrollo del cuerpo de la función. Finalmente, los ejemplos ilustran la prosa informal del propósito de la función. Así, futuros lectores del código, tales como profesores, colegas, o incluso clientes, podrán entender cuál es el concepto que está detrás del código.

3.3 Probar la función

Antes de completar la definición de la función, debemos definir como probarla. El proceso de probar una función se llama *testeo* o *testing*, y cada prueba se conoce como *test*.

En cualquier función que desarrollemos, nos debemos asegurar que al menos calcula efectivamente el valor esperado para los ejemplos definidos en el encabezado. Para facilitar el testeo, podemos hacer uso del comando `assert` de Python para definir un caso de uso y compararlo con el valor esperado.

Así, por ejemplo, si queremos probar que un valor calculado de la función es igual a uno que calculamos manualmente, podemos proceder como sigue:

```
assert areaRectangulo(5, 3) == 15
```

En este caso, le indicamos a Python que evalúe la aplicación de la función `areaRectangulo` con los parámetros `5` y `3`, y verifique si el resultado obtenido es efectivamente `15`. Si ese es el caso, la función se dice que *pasa el test*. En caso contrario, Python lanza un error y es entonces indicio que debemos verificar con detalle nuestra función. Por ejemplo:

```
>>> assert areaRectangulo(5, 3) == 15
>>> assert areaRectangulo(5, 3) == 0

Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AssertionError
```

Es importante recalcar que las pruebas que realizamos no pueden probar que una función produce salidas correctas para TODAS las entradas posibles. Esto se debe a que hay un número infinito de

combinaciones de valores de entrada para pasar como parámetros. Sin embargo, el testeo es una estrategia muy potente para verificar errores de sintaxis o de diseño en la función.

Para los casos en los que la función no pasa un test, debemos poner especial atención a los ejemplos que especificamos en el encabezado. En efecto, es posible que los ejemplos estén incorrectos, que la función tenga algún tipo de error, o incluso que tanto los ejemplos como la función tengan errores. En cualquier caso, deberíamos volver a revisar la definición de la función siguiendo los cuatro pasos anteriores.

3.4 Especificar el cuerpo de la función

Finalmente, debemos definir cuál es el cuerpo de la función. Esto es, debemos reemplazar los puntos suspensivos (`...`) de nuestra definición anterior por un conjunto de instrucciones que conformarán el cómo se procesarán los parámetros de entrada para producir la salida.

Notemos que podemos formular el cuerpo de la función únicamente si entendemos cómo la función calcula el valor de salida a partir del conjunto de valores de entrada. Así, si la relación entre las entradas y la salida están dadas por una fórmula matemática, basta con traducir esta expresión a Python. Si por el contrario nos enfrentamos a un problema verbal, debemos construir previamente la secuencia de pasos necesaria para formular la expresión.

En nuestro ejemplo, para resolver el problema basta con evaluar la expresión `largo * ancho` para obtener el área del rectángulo. Así, la traducción en Python de este proceso sería:

```
1 def areaRectangulo(largo, ancho):
2     return largo * ancho
```

Finalmente, la definición completa de nuestra función siguiendo la receta de diseño es como sigue:

```
1 # areaRectangulo:  num num -> num
2 # calcula el area de un rectangulo de medidas
3 # largo y ancho
4 # ejemplo: areaRectangulo(5, 3) debe producir 15
5 def areaRectangulo(largo, ancho):
6     return largo * ancho
7 # Tests
8 assert areaRectangulo(5, 3) == 15
```

Notemos que todas las funciones que se definan deben seguir la receta de diseño. Por ejemplo, si queremos definir la función `areaCuadrado` podemos seguir la receta de diseño y reutilizar la función `areaRectangulo`, obteniendo lo siguiente:

```
1 # areaCuadrado:  num -> num
2 # calcula el area de un cuadrado de medida lado
3 # ejemplo: areaCuadrado(5) debe producir 25
4 def areaCuadrado(lado):
5     return areaRectangulo(lado, lado)
6 # Tests
7 assert areaCuadrado(5) == 25
```

Capítulo 4

Módulos y Programas[1]

En general, un programa consta no sólo de una, sino de muchas definiciones de funciones. Por ejemplo, si retomamos el ejemplo del anillo que vimos en el capítulo 2, tenemos dos funciones: una para calcular el área de un círculo (`areaCirculo`) y una para calcular el área del anillo propiamente tal (`areaAnillo`). En otras palabras, dado que la función `areaAnillo` retorna el valor que queremos en nuestro programa, decimos que es la *función principal*. De igual manera, dado que la función `areaCirculo` apoya a la función principal, decimos que es una *función auxiliar*.

El uso de funciones auxiliares hace que el diseño de programas sea más manejable, y deja finalmente al código más limpio y entendible de leer. Por ejemplo, consideremos las siguientes dos versiones para la función `areaAnillo`:

```
1 def areaAnillo(interior, exterior): # Buena practica
2     return areaCirculo(exterior) - areaCirculo(interior)
3
4 def areaAnillo(interior, exterior): # Mala practica
5     return 3.14 * exterior ** 2 - 3.14 * interior ** 2
```

La primera definición está basada en una *composición* de funciones auxiliares. El diseñarla de esta manera nos ayudó a descomponer el problema en subproblemas más pequeños, pero más abordables. De hecho, el sólo leer la definición de la función (sin siquiera saber cómo están definidas las funciones auxiliares) nos da a entender que para calcular el área del anillo basta con restar el área de un círculo externo con el área del círculo en su interior. Por el contrario, la definición de la segunda versión de nuestra función obliga al lector el reconstruir la idea de que las subexpresiones en efecto calculan el área de un círculo. Peor aún, ¡estamos escribiendo dos veces la misma expresión!

Para un programa pequeño como el que hemos visto en el ejemplo, las diferencias entre ambos estilos de diseño de funciones son menores, aun cuando bastante significativas. Sin embargo, para programas o funciones más grandes, el usar funciones auxiliares no se vuelve una opción, sino una necesidad. Esto es, cada vez que se nos pida escribir un programa, debemos considerar el descomponerlo en funciones, y éstas a su vez descomponerlas en funciones auxiliares hasta que cada una de ellas resuelva UNO Y SÓLO UN SUBPROBLEMA particular.

[1] Parte de este texto fue traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

4.1 Descomponer un programa en funciones

Consideremos el siguiente problema:

"Una importante cadena de cines de Santiago tiene completa libertad en fijar los precios de las entradas. Claramente, mientras más cara sea la entrada, menos personas estarán dispuestas a pagar por ellas. En un reciente estudio de mercado, se determinó que hay una relación entre el precio al que se venden las entradas y la cantidad de espectadores promedio: a un precio de $5.000 por entrada, 120 personas van a ver la película; al reducir $500 en el precio de la entrada, los espectadores aumentan en 15. Desafortunadamente, mientras más personas ocupan la sala para ver la película, más se debe gastar en limpieza y mantenimiento general. Para reproducir una película, el cine gasta $180.000. Asimismo, se gastan en promedio $40 por espectador por conceptos de limpieza y mantenimiento. El gerente del cine le encarga determinar cuál es la relación exacta entre las ganancias y el precio de las entradas para poder decidir a qué precio se debe vender cada entrada para maximizar las ganancias totales."

Si leemos el problema, está clara cuál es la tarea que nos piden. Sin embargo, no resulta del todo evidente el cómo hacerlo. Lo único de lo que podemos estar seguros es que varias cantidades dependen entre sí.

Cuando nos vemos enfrentados a estas situaciones, lo mejor es identificar las dependencias y ver las relaciones una por una:

- Las *ganancias* corresponden a la diferencia entre los ingresos y los gastos.
- Los *ingresos* se generan exclusivamente a través de la venta de entradas. Corresponde al producto del valor de la entrada por el número de espectadores.
- Los *gastos* están formados por dos ítemes: un gasto fijo ($180.000) y un gasto variable que depende del número de espectadores.
- Finalmente, el enunciado del problema también especifica cómo el número de espectadores depende del precio de las entradas.

Definamos, pues, una función por cada una de estas dependencias; después de todo, las funciones precisamente calculan cómo distintos valores dependen de otros. Siguiendo la receta de diseño que presentamos en el capítulo anterior, comenzaremos definiendo los contratos, encabezados y propósitos para cada una de las funciones:

```
# ganancias:  int -> int
# calcular las ganancias como la diferencia entre los ingresos y
# los gastos dado precioEntrada
def ganancias(precioEntrada):
    ...
```

Notemos que las ganancias totales dependen del precio de las entradas, dado que tanto los ingresos como los gastos dependen a su vez del precio de las entradas.

```
# ingresos:  int -> int
# calcular el ingreso total, dado precioEntrada
def ingresos(precioEntrada):
    ...

```

```
6  # gastos:  int -> int
7  # calcular los gastos totales, dado precioEntrada
8  def gastos(precioEntrada):
9      ...
10
11 # espectadores: int -> int
12 # calcular el numero de espectadores, dado precioEntrada
13 def espectadores(precioEntrada):
14     ...
```

Esto nos permite enunciar la primera regla en el diseño de programas:

Antes de escribir cualquier línea de código siga la receta de diseño para cada función: formule el contrato, encabezado y propósito de la función, plantee ejemplos de uso relevantes y formule casos de prueba para verificar que su función se comportará correctamente.

Una vez escritas las formulaciones básicas de las funciones y al haber calculado a mano una serie de ejemplos de cálculo, podemos reemplazar los puntos suspensivos `...` por expresiones de Python. En efecto, la función `ganancias` calcula su resultado como la diferencia entre los resultados arrojados por las funciones `ingresos` y `gastos`, tal como lo sugiere el enunciado del problema y el análisis de las dependencias que hicimos anteriormente. El cálculo de cada una de estas funciones depende del precio de las entradas (`precioEntrada`), que es lo que indicamos como parámetro de las funciones. Para calcular los ingresos, primero calculamos el número de espectadores para `precioEntrada` y lo multiplicamos por `precioEntrada`. De igual manera, para calcular los gastos sumamos el costo fijo al costo variable, que corresponde al producto entre el número de espectadores y 40. Finalmente, el cálculo del número de espectadores también se sigue del enunciado del problema: podemos suponer una relación lineal entre el número de espectadores y el valor de la entrada. Así, 120 espectadores están dispuestos a pagar $5.000, mientras que cada $500 que se rebajen del precio, vendrán 15 espectadores más.

La definición de las funciones es como sigue:

```
1  def ganancias(precioEntrada):
2      return ingresos(precioEntrada) - gastos(precioEntrada)
3
4  def ingresos(precioEntrada):
5      return espectadores(precioEntrada) * precioEntrada
6
7  def gastos(precioEntrada):
8      return 180000 + espectadores(precioEntrada) * 40
9
10 def espectadores(precioEntrada):
11     return 120 + (5000 - precioEntrada) * 15 / 500
```

Si bien es cierto que podríamos haber escrito directamente la expresión para calcular el número de espectadores en todas las funciones, esto es altamente desventajoso en el caso de querer modificar una parte en la definición de la función. De igual manera, el código resultante sería completamente ilegible. Así, formulamos la siguiente regla que debemos seguir junto con la receta de diseño:

Diseñe funciones auxiliares para cada dependencia entre cantidades mencionadas en la especificación de un problema y por cada dependencia descubierta al elaborar ejemplos de casos de uso. Siga la receta de diseño para cada una de ellas.

De igual manera, en ocasiones podemos encontrarnos con valores que se repiten varias veces en una misma función o programa. Claramente, si queremos modificar su valor, no nos gustaría tener que modificarlo en cada una de las líneas en que aparece. Luego, lo recomendable es que sigamos una *definición de variable*, en la que asociamos un identificador con un valor (de la misma manera que a una variable le asociamos el resultado de una expresión). Por ejemplo, podemos asociarle el valor `3.14` a una variable de nombre `PI` para referirnos al valor de π en todas las líneas que necesitemos en nuestro programa. Así:

```
PI = 3.14
```

Luego, cada vez que nos refiramos a `PI`, el intérprete reemplazará el valor por `3.14`.

El usar nombres para las constantes hace más entendible el código para identificar dónde se reemplazan distintos valores. De igual manera, el programa se vuelve más mantenible en el caso de necesitar modificar el valor de la constante: sólo lo cambiamos en la línea en que hacemos la definición, y este cambio se propaga hacia abajo cada vez que se llama al identificador. En caso contrario, deberíamos modificar a mano cada una de las líneas en que escribimos directamente el valor.

Formulemos esta tercera regla:

Dé nombres relevantes a las constantes que utilizará frecuentemente en su programa, y utilice estos nombres en lugar de hacer referencia directa a su valor.

4.2 Módulos

La programación modular es una técnica de diseño que separa las funciones de un programa en *módulos*, los cuales definen una finalidad única y contienen todo lo necesario, código fuente y variables, para cumplirla. Conceptualmente, un módulo representa una separación de intereses, mejorando la mantenibilidad de un software ya que se fuerzan límites lógicos entre sus componentes. Así, dada una segmentación clara de las funcionalidades de un programa, es más fácil la búsqueda e identificación de errores.

Hasta el momento, solo hemos escrito programas en el intérprete de Python, por lo que no podemos reutilizar el código que hemos generado hasta el momento. Para guardar código en Python, lo debemos hacer en archivos con extensión `.py`. Así, basta con abrir un editor de texto (como por ejemplo el bloc de notas), copiar las funciones que deseamos almacenar y guardar el archivo con un nombre adecuado y extensión `.py`. Es importante destacar que existen muchas herramientas que destacan las palabras claves de Python con diferentes colores, haciendo más claro el proceso de escribir código.

Imaginemos ahora que queremos calcular el perímetro y el área de un triángulo dado el largo de sus lados. Primero debemos definir la función `perimetro` que recibe tres parámetros:

```
1 # perimetro:  num num num -> num
2 # calcula el perimetro de un triangulo de lados a, b, y c
3 # ejemplo:  perimetro(2, 3, 2) devuelve 7
```

```
4  def perimetro(a,b,c):
5      return a + b + c
6  # Test
7  assert perimetro(2, 3, 2) == 7
```

Dado que esta función pertenece a lo que se esperaría fueran las funcionalidades disponibles de un triángulo, crearemos un módulo que la almacene, cuyo nombre será `triangulo`. Así, abriremos un archivo con nombre `triangulo.py` y copiaremos nuestra función dentro de él.

Luego, solo nos queda definir la función de área. Sabemos que el área de un triángulo puede calcularse en función de su semiperímetro, representado por p, que no es más que la mitad del perímetro de un triángulo. La relación entre área y semiperímetro de un triángulo de lados a, b y c está dada por la siguiente fórmula:

$$A = \sqrt{p * (p - a) * (p - b) * (p - c)}$$

Para traducir esta fórmula en una función ejecutable necesitamos la función raíz cuadrada, que está incluida en el módulo `math` de Python. Para importar un módulo externo debemos incluir la siguiente línea en nuestro módulo triángulo: `import math`, que literalmente significa importar un módulo externo para ser usado en un programa. Para utilizar una función de un módulo, la notación a usar es `modulo.funcion(...)`. Luego, si la función raíz cuadrada del módulo `math` de Python se llama `sqrt` y toma un parámetro, podemos definir la función `area` de un triángulo de la siguiente manera:

```
1  # area: num num num -> float
2  # calcula el area de un triangulo de lados a,b, y c
3  # ejemplo: area(2, 3, 2) devuelve 1.98...
4  def area(a, b, c):
5      semi = perimetro(a, b, c) / 2.0
6      area = math.sqrt(semi * (semi - a) * (semi - b) * (semi - c))
7      return area
8
9  # Tests
10 assert area(3,4,5) == 6
```

Finalmente, nuestro módulo `triangulo` quedaría de la siguiente manera:

Contenido del archivo triangulo.py

```
1  import math
2
3  # perimetro:  num num num -> num
4  # calcula el perimetro de un triangulo de lados a, b, y c
5  # ejemplo:  perimetro(2, 3, 2) devuelve 7
6  def perimetro(a,b,c):
7      return a + b + c
8
9  # Test
10 assert perimetro(2, 3, 2) == 7
11
12
13 # area:  num num num -> float
14 # calcula el area de un triangulo de lados a,b, y c
```

Contenido del archivo triangulo.py (cont)

```
15 # ejemplo:  area(3,4,5) devuelve 6
16 def area(a, b, c):
17     semi = perimetro(a, b, c) / 2.0
18     area = math.sqrt(semi * (semi - a) * (semi - b) * (semi - c))
19     return area
20
21 # Test
22 assert area(3,4,5) == 6
```

4.3 Programas interactivos

Muchas veces se requiere crear programas que poseen algún tipo de interacción con el usuario. Por ejemplo, el usuario podría entregar el valor de los lados de un triángulo para calcular su perímetro o su área. En esta sección cubriremos dos conceptos básicos de cómo interactuar con un programa de software: pedir datos al usuario e imprimir mensajes en pantalla.

Para pedir datos al usuario, Python provee dos funciones: `input` y `raw_input`. La primera de ellas, `input`, recibe como parámetro un mensaje de tipo texto para el usuario y recupera el dato ingresado. Esto lo podemos ver en el siguiente ejemplo, en el cual se le pide al usuario ingresar un número:

```
1 >>> input('Ingrese un numero ')
2  -> Ingrese un numero 4
3  -> 4
```

Los valores ingresados por el usuario pueden ser guardados en variables. Con la función `input`, el tipo de la variable será el más adecuado a lo que ingrese el usuario. Es decir, si el usuario entrega un número, la variable será de tipo numérico, y si el usuario entrega una palabra o frase, la variable será de tipo texto. Veamos el ingreso de número:

```
1 >>> numero = input('Ingrese un numero: ')
2  -> Ingrese un numero: 10
3 >>> numero
4  -> 10
5 >>> doble = numero * 2
6 >>> doble
7  -> 20
```

El ingreso de texto es similar:

```
1 >>> nombre = input('Cual es su nombre?')
2  -> Cual es su nombre?'Enrique'
3 >>> nombre
4  -> 'Enrique'
```

Es importante notar que al ingresar valores de tipo texto, estos deben estar entre comillas para ser indentificados como tal por el intérprete. Cuando el usuario intenta ingresar texto sin comillas, el intérprete mostrará un error en pantalla.

La otra función para ingresar datos disponible en Python, `raw_input`, tiene un comportamiento similar, con la excepción de que todo valor ingresado se almacenará con tipo texto. Esto se ve en el siguiente código:

```
>>> numero = raw_input('Ingrese un numero: ')
 -> Ingrese un numero: 10
>>> numero
 -> '10'
>>> doble = numero * 2
>>> doble
 -> '1010'
```

Para que la interacción entre el computador y el humano no sea solamente en una dirección, también es posible que el programa entregue información al usuario. Así, un programa puede desplegar información en la consola de Python usando la función `print`. Esta función se utiliza escribiendo su palabra clave, seguida del texto o número a imprimir, como se ve en el siguiente ejemplo:

```
>>> print 'Hola, mundo!'
 -> Hola, mundo!
```

Cuando queremos mostrar más de un texto o número en una misma línea, por ejemplo dos frases seguidas, podemos unirlas por comas. Notemos que esto es equivalente a crear un elemento de tipo texto generado con el operador `+`. Para ver como funciona, preguntemos el nombre y el apellido al usuario, y luego mostrémoslo en pantalla.

```
>>> nombre = input('Cual es su nombre? ')
 -> Cual es su nombre? 'Enrique'
>>> apellido = input('Cual es su apellido? ')
 -> Cual es su apellido? 'Jorquera'
>>> print 'Su nombre es', nombre, apellido
 -> Su nombre es Enrique Jorquera
```

Volvamos al ejemplo del inicio, en donde calculamos el perímetro y área de un triángulo. Ya que sabemos cómo preguntar información al usuario, sería interesante construir un programa que pregunte los lados de un triángulo al usuario y utilice nuestro módulo para calcular los valores de su perímetro y área.

Para realizar este programa, debemos realizar tres pasos:

1. Importar el módulo creado;
2. preguntar por los valores necesarios, en este caso los lados del triángulo;
3. utilizar el módulo `triangulo` para calcular el área y perímetro.

Así, primero que nada, debemos importar el módulo que creamos con la palabra clave `import`:

```
import triangulo
```

Luego, debemos preguntar por el largo de cada lado del triángulo y almacenarlos en variables cuyos nombres sean representativos, como se muestra a continuación:

```
print 'Calcular el area y perimetro de un triangulo'
l1 = input('Ingrese el largo del primer lado')
l2 = input('Ingrese el largo del segundo lado')
l3 = input('Ingrese el largo del tercer lado')
```

Y finalmente utilizamos nuestro módulo para calcular al área y perímetro del triángulo dado:

```
print 'El perimetro del triangulo es', triangulo.perimetro(l1, l2, l3)
print 'El area del triangulo es', triangulo.area(l1, l2, l3)
```

El programa resultante se puede ver a continuación:

```
import triangulo

print 'Calcular el area y perimetro de un triangulo'
l1 = input('Ingrese el largo del primer lado')
l2 = input('Ingrese el largo del segundo lado')
l3 = input('Ingrese el largo del tercer lado')

print 'El perimetro del triangulo es', triangulo.perimetro(l1, l2, l3)
print 'El area del triangulo es', triangulo.area(l1, l2, l3)
```

Ahora que tenemos listo nuestro programa, podemos guardarlo en un archivo .py y ejecutarlo cada vez que necesitemos calcular el área y perímetro de un triángulo cualquiera (suponiendo que los valores entregados corresponden a un triángulo válido).

Para terminar, una manera alternativa de importar una función de un módulo es ocupar la instrucción:

```
from nombreModulo import nombreFuncion
```

Note que con esta instrucción sólo se está importando la función `nombreFuncion` del módulo `nombreModulo`, y ninguna otra que pueda haber en dicho módulo. Además, para invocar esta función ya no se debe escribir `nombreModulo.nombreFuncion(...)`, sino que debe escribirse `nombreFuncion(...)`. Finalmente, si se desea importar de esta forma todas las funciones del módulo, se ocupa la instrucción:

```
from nombreModulo import *
```

Modificando nuestro programa interactivo para ocupar esta forma alternativa de importar funciones de un módulo, queda como sigue:

Contenido del archivo pruebaTriangulo.py

```
from triangulo import *

print 'Calcular el area y perimetro de un triangulo '
```

Contenido del archivo pruebaTriangulo.py (cont)

```
4 l1 = input ('Ingrese el largo del primer lado ')
5 l2 = input ('Ingrese el largo del segundo lado ')
6 l3 = input ('Ingrese el largo del tercer lado ')
7
8 print 'El perimetro del triangulo es ', perimetro(l1, l2, l3)
9 print 'El area del triangulo es ', area(l1, l2, l3)
```

Capítulo 5

Expresiones y Funciones Condicionales[1]

En general, los programas deben trabajar con distintos datos en distintas situaciones. Por ejemplo, un videojuego puede tener que determinar cuál es la velocidad de un objeto en un rango determinado, o bien cuál es su posición en pantalla. Para un programa de control de maquinaria, una condición puede describir en qué casos una válvula se debe abrir. Para manejar condiciones en nuestros programas, necesitamos una manera de saber si esta condición será *verdadera* o *falsa*. Así, necesitamos una nueva clase de valores, los que, por convención, llamamos valores *booleanos* (o valores de verdad). En este capítulo veremos los valores de tipo booleano, expresiones que se evalúan a valores booleanos, y expresiones que calculan valores dependiendo del valor de verdad de una evaluación.

5.1 Valores booleanos

Consideremos el siguiente problema:

"Genera S.A. le paga $4.500 por hora a todos sus ingenieros de procesos recién egresados. Un empleado típicamente trabaja entre 20 y 65 horas por semana. La gerencia de informática le pide desarrollar un programa que calcule el sueldo de un empleado a partir del número de horas trabajadas *si este valor está dentro del rango apropiado*."

Las palabras en *cursiva* resaltan qué es lo nuevo respecto al problema que presentamos en el capítulo de Funciones. Esta nueva restricción implica que el programa debe manipular al valor de entrada de una manera si tiene una forma específica, y de otra manera si no. En otras palabras, de la misma manera que las personas toman decisiones a partir de ciertas condiciones, los programas deben ser capaces de operar de manera condicional.

Las condiciones no deberían ser nada nuevo para nosotros. En matemática, hablamos de proposiciones *verdaderas* y *falsas*, las que efectivamente describen condiciones. Por ejemplo, un número puede ser igual a, menor que, o mayor que otro número. Así, si x e y son números, podemos plantear las siguientes tres proposiciones acerca de x e y:

1. $x = y$: "x es igual a y";
2. $x < y$: "x es estrictamente menor que y";

[1] Parte de este texto fue traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

3. $x > y$: "x es estrictamente mayor que y".

Para cualquier par de números (reales), una y sólo una de estas tres proposiciones es verdadera. Por ejemplo, si $x = 4$ y $y = 5$, entonces la segunda proposición es verdadera y las otras son falsas. Si $x = 5$ y $y = 4$, entonces la tercera es verdadera y las otras son falsas. En general, una proposición es verdadera para ciertos valores de variables y falsa para otros.

Además de determinar si una proposición atómica es verdadera o falsa en algún caso, a veces resulta importante determinar si la combinación de distintas proposiciones resulta verdadera o falsa. Consideremos las tres proposiciones anteriores, las que podemos combinar, por ejemplo, de distintas maneras:

1. $x = y$ y $x < y$ y $x > y$;
2. $x = y$ o $x < y$ o $x > y$;
3. $x = y$ o $x < y$.

La primera proposición compuesta es siempre falsa, pues dado cualquier par de números (reales) para x e y, dos de las tres proposiciones atómicas son falsas. La segunda proposición compuesta es, sin embargo, siempre verdadera para cualquier par de números (reales) x e y. Finalmente, la tercera proposición compuesta es verdadera para ciertos valores y falsa para otros. Por ejemplo, es verdadera para $x = 4$ y $y = 4$, y para $x = 4$ y $y = 5$, mientras que es falsa si $x = 5$ y $y = 3$.

Al igual que en matemática, Python provee comandos específicos para representar el valor de verdad de proposiciones atómicas, para representar estas proposiciones, para combinarlas y para evaluarlas. Así, el valor lógico *verdadero* es `True`, y el valor *falso* se representa por `False`. Si una proposición relaciona dos números, esto lo podemos representar usando *operadores relacionales*, tales como: `==` (igualdad), `>` (mayor que), y `<` (menor que).

Traduciendo en Python las tres proposiciones matemáticas que definimos inicialmente, tendríamos lo siguiente:

1. `x == y`: "x es igual a y";
2. `x < y`: "x es estrictamente menor que y";
3. `x > y`: "x es estrictamente mayor que y".

Además de los operadores anteriores, Python provee como operadores relacionales: `<=` (menor o igual que), `>=` (mayor o igual que), y `!=` (distinto de).

Una expresión de Python que compara números tiene un resultado, al igual que cualquier otra expresión de Python. El resultado, sin embargo, es `True` o `False`, y no un número. Así, cuando una proposición atómica entre dos números es verdadera, en Python se evalúa a `True`. Por ejemplo:

```
1 >>> 4 < 5
2  -> True
```

De igual manera, una proposición falsa se evalúa a `False`:

```
1 >>> 4 == 5
2  -> False
```

Para expresar condiciones compuestas en Python usaremos tres conectores lógicos: `and` (conjunción lógica: "y"), `or` (disyunción lógica: "o") y `not` (negación: "no"). Por ejemplo, supongamos que queremos combinar las proposiciones atómicas `x == y` y `y < z`, de tal manera que la proposición compuesta sea verdadera cuando ambas condiciones sean verdaderas. En Python escribiríamos:

```
x == y and y < z
```

para expresar esta relación. De igual manera, si queremos formular una proposición compuesta que sea verdadera cuando (al menos) una de las proposiciones sea verdadera, escribimos:

```
x == y or y < z
```

Finalmente, si escribimos algo como:

```
not x == y
```

lo que estamos indicando es que deseamos que la negación de la proposición sea verdadera.

Las condiciones compuestas, al igual que las condiciones atómicas, se evalúan a `True` o `False`. Consideremos por ejemplo la siguiente condición compuesta: `5 == 5 and 5 < 6`. Note que está formada por dos proposiciones atómicas: `5 == 5` y `5 < 6`. Ambas se evalúan a `True`, y luego, la evaluación de la compuesta se evalúa a: `True and True`, que da como resultado `True` de acuerdo a las reglas de la lógica proposicional. Las reglas de evaluación para `or` y `not` siguen el mismo patrón.

En las siguientes secciones veremos por qué es necesario formular condiciones para programar y explicaremos cómo hacerlo.

5.2 Funciones sobre booleanos

Consideremos la siguiente función sencilla para verificar una condición sobre un número:

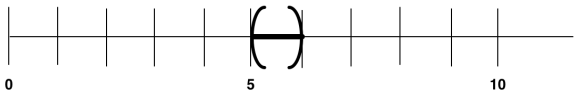
```
# esIgualA5: num -> bool
# determinar si n es igual a 5
def esIgualA5(n):
    return n == 5
```

Esta función produce `True` si y sólo si su argumento es igual a `5`. Su contrato contiene un nuevo elemento: la palabra `bool`. Al igual que `int`, `float` y `str`, la palabra `bool` representa una clase de valores *booleanos* que está definida en Python. Sin embargo, a diferencia de los valores de tipo numérico y texto, los booleanos sólo pueden ser `True` o `False`.

Consideremos este ejemplo un poco más sofisticado:

```
# estaEntre5y6: num -> bool
# determinar si n esta entre 5 y 6 (sin incluirlos)
def estaEntre5y6(n):
   return 5 < n and n < 6
```

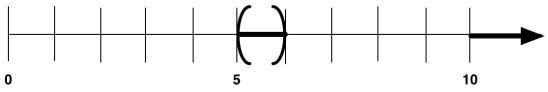
Es claro ver que esta función consume un número (entero o real) y produce `True` si el número está entre `5` y `6`, sin incluirlos. Una forma de entender el funcionamiento de esta función es describiendo el intervalo que definen las condiciones en la recta numérica, tal como lo muestra la siguiente figura.



De igual manera, si queremos operar con condiciones más complejas sobre números, un primer paso puede ser determinar los rangos de definición en la recta numérica, y luego definir una función sobre los valores que se pueden tomar en el (los) intervalo(s). Por ejemplo, la siguiente función determina si un número está entre 5 o 6, o bien es mayor que 10:

```
# estaEntre5y6MayorQue10: num -> bool
# determinar si n esta entre 5 y 6 (sin incluirlos),
# o bien es mayor o igual que 10
def estaEntre5y6MayorQue10(n):
    return estaEntre5y6(n) or n >= 10
```

Y la figura que representa las porciones de la recta numérica donde la función se evalúa a `True` es:



Esto es, cualquier número entre `5` y `6` sin incluirlos, o bien, cualquier número mayor o igual que 10. En este caso, desarrollamos una condición compuesta *componiendo* los distintos trozos que definen al intervalo donde la función se debe evaluar a *verdadero*.

5.3 Condiciones

Imaginemos que queremos crear un programa que juegue al cachipún con el usuario, pero que siempre le gane, independiente de lo que éste le entregue. Esto significa que debemos diseñar e implementar un programa que, dada una jugada del usuario, entregue la jugada que le gana según las reglas del cachipún. Para esto, en las dos secciones siguientes veremos las *expresiones condicionales* y las instrucciones que provee Python para crear funciones con este tipo de expresiones.

Las *expresiones condicionales* se caracterizan por ser del tipo:

si pregunta **entonces** respuesta

En particular para Python, la traducción de estas expresiones está dada de la siguiente manera:

```
if pregunta:
    respuesta
```

Al ser ejecutada, se verifica si el resultado de la evaluación de la pregunta es verdadero o falso. En Python, esto quiere decir si el valor evaluado es igual a `True` o `False`. Por ejemplo, imaginemos que tenemos dos variables de tipo numérico y queremos saber si son iguales. Si las variables se llaman `x` e `y`, podemos mostrar en pantalla al usuario si es que esta condición es verdadera:

```
if x == y:
   print 'Son iguales!'
```

Una expresión condicional puede estar compuesta de más de una pregunta asociada a una respuesta. En el ejemplo anterior, podríamos además decir cuál de las dos variables representa al número mayor. Así, las expresiones condicionales también pueden ser de la forma:

si pregunta **entonces** respuesta,
sino pregunta **entonces** respuesta
...
sino pregunta **entonces** respuesta

En Python, para modelar este tipo de expresiones podemos utilizar las instrucciones `elif` y `else`, como se muestra a continuación:

```
if pregunta:
    respuesta
elif pregunta:
    respuesta
...
elif pregunta:
    respuesta
```

O:

```
if pregunta:
    respuesta
elif pregunta:
    respuesta
...
else:
    respuesta
```

Al igual que en las expresiones condicionales, los tres puntos indican que las expresiones `if` pueden tener más de una condición. Las expresiones condicionales, como ya hemos visto, se componen de dos expresiones **pregunta** y una **respuesta**. La pregunta es una expresión condicional que al ser evaluada siempre debe entregar un valor booleano, y la respuesta es una expresión que sólo será evaluada si es que la condición asociada a esta se cumple.

Tomemos el ejemplo anterior, en donde comparamos las variables de tipo numérico `x` e `y`, y mostremos al usuario cuál de los dos es mayor o, en su defecto, si son iguales. Así, tenemos tres casos posibles: (i) ambas variables representan a números del mismo valor; (ii) `x` tiene un valor mayor a `y`; (iii) `y` tiene un valor mayor a `x`. La traducción de estas tres condiciones en el lenguaje Python está dado como sigue:

```
if x == y:
   print 'Son iguales!'
elif x > y:
   print 'x es mayor que y'
elif y > x:
   print 'y es mayor que x'
```

O:

```
if x == y:
   print 'Son iguales!'
```

```
3 elif x > y:
4     print 'x es mayor que y'
5 else:
6     print 'y es mayor que x'
```

Cuando se evalúa una expresión condicional completa, esto se hace en orden, evaluando cada pregunta, o condición, una por una. Si una pregunta se evalúa como verdadero, entonces la respuesta asociada a esa pregunta se evaluará y será el resultado de la expresión condicional completa. Si no es así, entonces se continuará con la evaluación de la siguiente pregunta y así sucesivamente hasta que alguna de las condiciones se cumpla. Esto quiere decir que para el ejemplo anterior, primero se evaluará la primera pregunta (`x == y`) y si esta se cumple, se mostrará en consola el mensaje `'Son iguales!'`. Si es que no se cumple, entonces seguirá con la siguiente instrucción `elif` y evaluará su pregunta asociada, `x > y`, imprimiendo en pantalla si es que esta condición se cumple. Si no, evaluará la última pregunta e imprimirá el mensaje.

Aunque las expresiones del ejemplo anterior tienen una sintaxis algo diferente, ambas son equivalentes. Podemos notar que la expresión de la derecha está formada solamente con instrucciones `if` y `elif`, lo que significa que se evalúan las tres condiciones posibles de nuestro ejemplo de manera explícita. Mientras que la expresión de la derecha utiliza la instrucción `else`, la cual indica que su respuesta será evaluada sólo si ninguna de las preguntas anteriores se evalúa como verdadera.

Volvamos a nuestro ejemplo del cachipún en el cual el usuario siempre pierde. Para diseñar un programa que determine la jugada ganadora dada una entrada del usuario, debemos identificar las tres situaciones posibles, resumidas a continuación:

- Si el usuario entrega piedra, el programa debe entregar papel
- Si el usuario entrega papel, el programa debe entregar tijera
- Si el usuario entrega tijera, el programa debe entregar piedra

Luego, el programa completo consta de tres partes principales: (i) pedir al usuario la jugada a ingresar; (ii) identificar la jugada que le ganará a la ingresada por el jugador humano; y por último (iii) mostrarla en pantalla. La segunda parte estará definida en una función que, dada una entrada, entregue como resultado la jugada ganadora. Así, siguiendo la receta de diseño, debemos, primero que todo, escribir su contrato y formular su propósito:

```
1 # jaliscoCachipun:  str -> str
2 # entrega la jugada ganadora del cachipun dada una entrada valida
3 def jaliscoCachipun(jugada):
4     ...
```

Luego, debemos agregar un ejemplo de la función:

```
1 # jaliscoCachipun:  str -> str
2 # entrega la jugada ganadora del cachipun dada una entrada valida
3 # ejemplo:  jaliscoCachipun('tijera') debe producir 'piedra'
4 def jaliscoCachipun(jugada):
5     ...
```

A continuación, escribimos un test para probar que nuestar función se comporta de manera adecuada.

```
13 assert jaliscoCachipun('tijera') == 'piedra'
```

Finalmente, debemos especificar el cuerpo de la función:

```
4  def jaliscoCachipun(jugada):
5      if jugada == 'piedra':
6          return 'papel'
7      elif jugada == 'papel':
8          return 'tijera'
9      elif jugada == 'tijera':
10         return 'piedra'
```

La definición complesta de nuestra función esta dada como sigue:

Contenido del archivo cachipun.py

```
1  # jaliscoCachipun:  str -> str
2  # entrega la jugada ganadora del cachipun dada una entrada valida
3  # ejemplo:  jaliscoCachipun('tijera') debe producir 'piedra'
4  def jaliscoCachipun(jugada):
5      if jugada == 'piedra':
6          return 'papel'
7      elif jugada == 'papel':
8          return 'tijera'
9      elif jugada == 'tijera':
10         return 'piedra'
11
12 #test
13 assert jaliscoCachipun('tijera') == 'piedra'
```

Ahora que nuestra función está completa, podemos usarla para jugar con el usuario:

Contenido del archivo juegoCachipun.py

```
1  from cachipun import jaliscoCachipun
2
3  print 'Juego del Jalisco cachipun'
4  jugada = input('Ingrese una jugada (piedra, papel o tijera)')
5  jugadaGanadora = jaliscoCachipun(jugada)
6  print ('Yo te gano con ' + jugadaGanadora)
```

5.4 Bloques de código condicionales en Python

Dado que la respuesta de una expresión condicional puede estar compuesta por más de una línea de código, es necesario indicar a qué pregunta pertenecen. Para esto, todas las líneas de código que pertenezcan a la respuesta de una expresión condicional deben estar indentadas un nivel más que la instrucción condicional a la que pertenecen. En otras palabras, todas las líneas que tengan una

indentación más que la cláusula `if` están subordinadas a esta, y se dice que forman un *bloque de código*. Un bloque de código sólo será evaluado si es que condición asociada se cumple.

En el ejemplo que vimos anteriormente, si no agregamos la indentación correspondiente se producirá un error al intentar ejecutar el programa:

```
>>> if x == y:
... print 'son iguales!'
```

```
  File "<stdin>", line 2
    print 'son iguales!'
        ^
IndentationError: expected an indented block
```

Esto es similar al comportamiento de Python cuando uno define funciones.

5.5 Diseñar funciones condicionales

Tal como vimos anteriormente, la clave para diseñar funciones que requieran expresiones condicionales es reconocer que el enunciado del problema genera casos e identificar cuáles son. Para enfatizar la importancia de esta idea, introduciremos y discutiremos la receta de diseño para las funciones condicionales. La nueva receta introduce un nuevo paso, llamado *análisis de los datos*, la cual requiere que un programador entienda las diferentes situaciones que se discuten en el enunciado del problema. También modifica los pasos de Ejemplo y Cuerpo de la receta de diseño explicada en los capítulos anteriores.

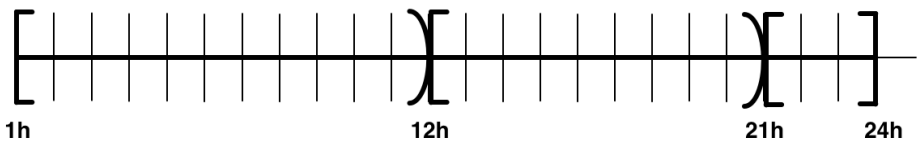
5.5.1 Análisis de los datos y definición

Luego de determinar que en el enunciado de un problema debemos lidiar con diferentes situaciones, es necesario identificar cada una de ellas.

Para funciones numéricas, una buena estrategia es dibujar una recta numérica e identificar los intervalos correspondientes a la situación particular a estudiar. Imaginemos que queremos implementar un programa que retorne el saludo correspondiente a la hora del día. Así, si son más de las 1 de la mañana y menos de las 12 de la tarde, el programa responderá 'Buenos días!'; si menos de las 21 horas, el mensaje será 'Buenas tardes!'; y si es más de las 21, entonces el programa deseará las buenas noches. Consideremos el contrato de esta función:

```
# saludo: int -> int
# Determinar el saludo adecuado a la hora del dia 1 <= h <= 24
def saludo(hora):
    ...
```

Esta función recibe como entrada números enteros que están dentro del rago descrito por el contrato, y genera respuestas para tres diferentes situaciones, indicados por los intervalos de la siguiente figura:



5.5.2 Dar ejemplos de uso la función

Los ejemplos a escoger para este paso en la receta de diseño deben considerar las diferentes situaciones posibles. Como mínimo, debemos desarrollar un ejemplo por cada situación. Si describimos cada situación como un intervalo numérico, los ejemplos también deben incluir todos los casos de borde.

Para nuestra función `saludo`, deberíamos usar `1`, `12`, y `21` como casos de borde. Además, deberíamos escoger números como `8`, `16`, y `22` para probar el comportamiento al interior de cada uno de los tres intervalos.

5.5.3 El cuerpo de la función: diseñar condiciones

El cuerpo de la función debe estar compuesta de una instrucción `if` que tiene tantas clásulas como situaciones diferentes. Este requerimiento sugiere de inmediato el siguiente cuerpo para nuestra función:

```
if (...):
    ...
elif (...):
    ...
elif (...):
    ...
```

Luego formulamos las condiciones para describir cada una de las situaciones. Las condiciones son proposiciones sobre los parámetros de la función, expresados con operadores relacionales o con funciones hechas por nosotros mismos.

Las líneas de nuestro ejemplo se completa para traducirse en las siguientes tres condiciones:

1. $(1 \leq \text{hora})$ y $(\text{hora} < 12)$
2. $(12 \leq \text{hora})$ y $(\text{hora} < 21)$
3. $(21 \leq \text{hora})$

Agregando estas condiciones a la función, tenemos una mejor aproximación de la definición final:

```
def saludo(hora):
    if (1 <= hora) and (hora < 12):
        ...
    elif (12 <= hora) and (hora < 21):
        ...
    elif (21 <= hora):
        ...
```

En este punto, el programador debe asegurarse que las condiciones escogidas distinguen las diferentes entradas posibles de manera correcta. En particular, que cada dato posible esté dentro de una posible situación o intervalo. Esto quiere decir que cuando una pregunta o condición son evaluadas como `True`, todas las condiciones precedentes deben ser evaluadas como `False`.

5.5.4 El cuerpo de la función: responder a cada condición

Finalmente, debemos determinar qué debe producir la función por cada una de las cláusulas `if`. Más concretamente, debemos considerar cada expresión `if` por separado, asumiendo que la condición se

cumple.

En nuestro ejemplo, los resultados son especificados directamente del enunciado del problema. Estos son `'Buenos dias!'`, `'Buenas tardes!'`, y `'Buenas noches!'`. En ejemplos más complejos, debe ser el programador quien determina la expresión la respuesta de cada condición, puesto que no siempre están descritas de manera tan explícita. Estas se pueden contruir siguiendo los pasos de la receta de diseño que hemos aprendido hasta ahora.

```
def saludo(hora):
    if (1 <= hora) and (hora < 12):
        return 'Buenos dias!'
    elif (12 <= hora) and (hora < 21):
        return 'Buenas tardes!'
    elif (21 <= hora):
        return 'Buenas noches!'
```

5.5.5 Simplificar condiciones

Cuando la definición de una función está completa y probada, un programador querrá verificar si es que las condiciones pueden ser simplificadas. En nuestro ejemplo, sabemos que la `hora` es siempre mayor o igual a uno, por lo que la primera condición podría ser formulada como:

```
hora <= 12
```

Más aún, sabemos que las expresiones `if` son evaluadas secuencialmente. Esto es, cuando la segunda condición es evaluada, la primera ya debe haber producido `False`. Por lo tanto sabemos que en la segunda condicional la cantidad *no es* menor o igual a 12, lo que implica que su componente izquierda es innecesaria. La definición completa y simplificada de la función `saludo` se describe como sigue:

Funcion saludo completa, y simplificada

```
# saludo: int -> str
# Determinar el saludo adecuado a la hora del dia 1 <= h <= 24
# ejemplos:
#    saludo(11) debe devolver 'Buenos dias!'
#    saludo(15) debe devolver 'Buenas tardes!'
def saludo_simple(hora):
    if (hora <= 12):
        return 'Buenos dias!'
    elif (hora <= 21):
        return 'Buenas tardes!'
    elif (21 < hora):
        return 'Buenas noches!'

# test:
assert saludo_simple(11) == 'Buenos dias!'
assert saludo_simple(15) == 'Buenas tardes!'
```

¿Sería correcto el ejemplo? En realidad, no lo es. Si revisan los intervalos en la figura, se van a dar cuenta que el programa devuelve un saludo equivocado para la 12, entre otros problemas. ¡Esto

muestra la importancia de agregar tests ANTES de escribir el cuerpo de la función!

La versión correcta (y con condiciones simplificadas) de la funcion de saludo es la siguiente:

Funcion saludo testeada, depurada, y simplificada

```
# saludo: int -> str
# Determinar el saludo adecuado a la hora del dia 1 <= h <= 24
# ejemplos:
#    saludo(11) debe devolver 'Buenos dias!'
#    saludo(15) debe devolver 'Buenas tardes!'
def saludo_depurado(hora):
    if (hora < 12):
        return 'Buenos dias!'
    elif (hora < 21):
        return 'Buenas tardes!'
    else:
        return 'Buenas noches!'

# test:
assert saludo_depurado(1) == 'Buenos dias!'
assert saludo_depurado(11) == 'Buenos dias!'
assert saludo_depurado(12) == 'Buenas tardes!'
assert saludo_depurado(15) == 'Buenas tardes!'
assert saludo_depurado(21) == 'Buenas noches!'
assert saludo_depurado(23) == 'Buenas noches!'
assert saludo_depurado(24) == 'Buenas noches!'
```

Capítulo 6

Recursión

Muchas veces nos tocará enfrentarnos con definiciones que dependen de sí mismas. En particular, en programación se habla de funciones y estructuras recursivas cuando su definición depende de la misma definición de éstas. En este capítulo veremos un par de ejemplos de funciones recursivas.

6.1 Potencias, factoriales y sucesiones

Para calcular la potencia de un número con exponente entero, a^b, podemos usar la siguiente definición:

$$a^b = \begin{cases} 1 & \text{si } b = 0 \\ a \cdot a^{b-1} & \text{si } b > 0 \end{cases}$$

Como vemos, cuando el exponente es mayor que 0, para calcular la potencia necesitamos la misma definición con un exponente menor. La evaluación una potencia termina en el caso en que el exponente es 0. Por ejemplo, si queremos calcular 2^4, basta con aplicar la definición:

$$\begin{aligned} 2^4 &= 2 \cdot 2^{4-1} \\ &= 2 \cdot 2^3 \\ &= 2 \cdot (2 \cdot 2^2) \\ &= 2 \cdot (2 \cdot (2 \cdot 2^1)) \\ &= 2 \cdot (2 \cdot (2 \cdot (2 \cdot 2^0))) \\ &= 2 \cdot (2 \cdot (2 \cdot (2 \cdot 1))) \\ &= 16 \end{aligned}$$

Una función que se define en términos de sí misma es llamada *función recursiva*.

Observe que si quitamos la primera parte de la definición de potencia, al calcular 2^4, su evaluación nunca llegará a término. Esta parte es necesaria para dar término a la evaluación de una función recursiva, a la cual llamaremos *caso base*.

La segunda parte de la definición, a la cual llamaremos *caso recursivo*, es la que hace uso de su propia definición para continuar la evaluación hasta llegar al caso base.

Veamos cómo queda la función `potencia` escrita en Python:

```python
# potencia:  num int -> num
# calcula la potencia de base elevado a exp
# ejemplo:  potencia(2, 4) devuelve 16
def potencia(base, exp):
        if exp == 0:
                # caso base
               return 1
        else:
                # caso recursivo
              return base * potencia(base, exp - 1)

# Test
assert potencia(2, 4) == 16
assert potencia(-1, 5) == -1
assert potencia(3, 0) == 1
```

Para entender cómo definir una función recursiva será clave la etapa de entender el propósito de la función. Como vimos, una función cumple el objetivo de consumir y producir información. Si en el proceso de consumir información se llega al mismo problema inicial (usualmente con una entrada o un parámetro más pequeño), entonces una solución recursiva puede ser correcta. En el ejemplo de la potencia, vimos que, por definición, debemos multiplicar la base por otro valor, que resulta ser la misma potencia con un exponente más pequeño, en cuyo caso conviene utilizar una solución recursiva.

De manera similar podemos definir el factorial de un número entero n:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

```python
# factorial:  int -> int
# calcula el factorial de n
# ejemplo:  factorial(10) devuelve 3628800
def factorial(n):
        if n == 0:
                # caso base
               return 1
        else:
                # caso recursivo
                return n * factorial(n - 1)

# Test
assert factorial(0) == 1
assert factorial(5) == 120
assert factorial(10) == 3628800
```

Otro ejemplo clásico de recursión es la generación de los números de Fibonacci. Los números de Fibonacci forman una sucesión de números que parten de la siguiente forma:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Se puede apreciar que cada número de la sucesión es igual a la suma de los dos números anteriores. Por supuesto, los dos primeros números de Fibonacci (0 y 1) son parte de la definición (caso base) de la sucesión, dado que no hay dos números anteriores para formarlos. El n-ésimo número de Fibonacci se calcula sumando los dos números anteriores de la sucesión, por lo que la recursión es clara en este caso. Formalmente, cada número de Fibonacci se puede calcular siguiendo esta definición:

$$F_n = \begin{cases} n & \text{si } 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & \text{si } n > 1 \end{cases}$$

La implementación en Python de una función que calcula el enésimo número de Fibonacci es la siguiente:

```
1  # fibonacci:  int -> int
2  # calcula el n-esimo numero de la sucesion de fibonacci
3  # ejemplo:  fibonacci(7) devuelve 13
4  def fibonacci(n):
5          if n < 2:
6                  # caso base
7                  return n
8          else:
9                  # caso recursivo
10                 return fibonacci(n - 1) + fibonacci(n - 2)
11
12 # Test
13 assert fibonacci(0) == 0
14 assert fibonacci(1) == 1
15 assert fibonacci(7) == 13
```

6.2 Torres de Hanoi

Un ejemplo de recursión más complicado es el problema de las Torres de Hanoi.

Las Torres de Hanoi es el nombre de un puzzle matemático que consiste en mover todos los discos de una vara a otra, bajo ciertas restricciones. El juego consta de una plataforma con tres varas y n discos puestos en orden decreciente de tamaño en una de ellas. El objetivo del juego es mover todos los discos de una vara a la otra, de forma que al final se mantenga el mismo orden.

Las reglas del juego son las siguientes:

1. Sólo 1 disco puede ser movido a la vez.
2. No puede haber un disco más grande encima de uno más pequeño.
3. Un movimiento consiste en mover un disco en la cima de una pila de discos hacia otra pila de discos puestos en otra vara.

Nos interesa saber cuántos movimientos son necesarios para resolver el juego.

Solución

La clave para resolver el puzzle no está en determinar cuáles son los movimientos a realizar, sino en que el juego puede ser descompuesto en *instancias* más pequeñas. En el caso de las Torres de Hanoi, el problema está en mover n discos. Por lo tanto, veamos una forma de resolver el problema de forma de tener que resolver el juego con $n - 1$ discos, y volvamos a aplicar el procedimiento hasta mover todos los discos.

El objetivo del juego es mover la pila completa de una vara a la otra. Por lo que, inicialmente, lo único a lo que podemos apuntar a lograr es a trasladar el disco más grande de su vara a otra, y no nos

queda otra opción que mover todos los discos restantes de su vara a otra.

Supongamos que ya tenemos una función `hanoi(n)` que nos dice cuántos movimientos hay que realizar para mover n discos de una vara a otra. Esa función es la que queremos definir, ¡pero al mismo tiempo la necesitamos para resolver el problema! En el ejemplo de la figura necesitamos 15 movimientos para resolver el puzzle.

En resumen, debemos considerar los siguientes movimientos:

- Para mover el disco más grande de una vara a otra, necesitamos mover los $n-1$ discos anteriores a otra vara, lo cual nos toma `hanoi(n-1)` movimientos.
- Luego, debemos mover el disco más grande de su vara a la desocupada, esto nos toma 1 movimiento.
- A continuación, debemos volver a mover los $n-1$ discos restantes para que queden encima del disco grande que acabamos de mover. Esto nuevamente nos toma `hanoi(n-1)` movimientos.
- En total, necesitamos $2\times$ `hanoi(n-1)` $+1$ movimientos para n discos.

¿Cuál es el caso base? Si tenemos 1 disco, sólo debemos moverlo de su vara a la otra para completar el juego.

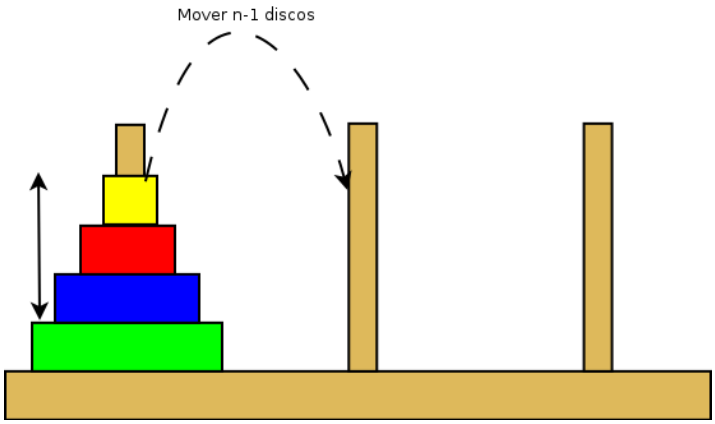


Figura 6.1: Mover los $n-1$ primeros discos, recursivamente hacia la segunda vara.

Ahora que entendimos el propósito y la solución del juego, podemos escribirla en Python:

```
# hanoi:  int -> int
# calcula la cantidad de movimientos necesarios para resolver
# las Torres de Hanoi con n discos y 3 varas
# ejemplo:  hanoi(4) devuelve 15
def hanoi(n):
        if n == 1:
                # caso base
                return 1
        else:
                # caso recursivo
                return 2 * hanoi(n - 1) + 1
```

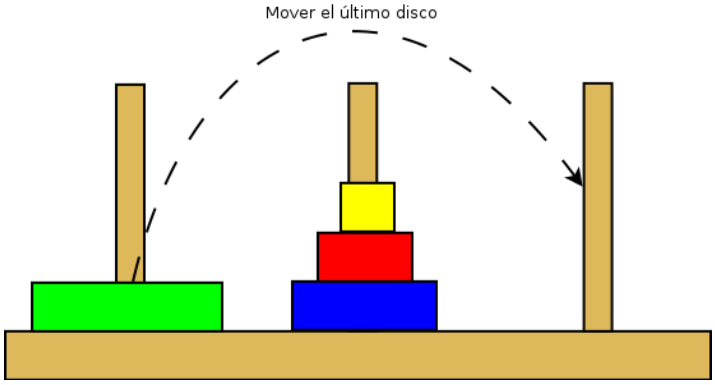


Figura 6.2: Mover el último disco hacia la tercera vara.

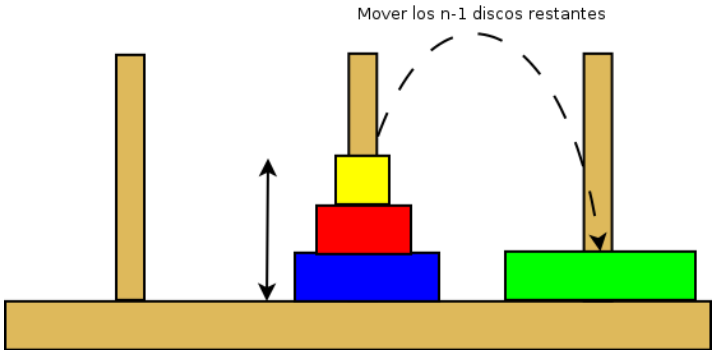


Figura 6.3: Volver a mover los primeros $n-1$ discos, recursivamente hacia la tercera vara.

```
12
13 # Test
14 assert hanoi(1) == 1
15 assert hanoi(4) == 15
16 assert hanoi(5) == 31
```

Se puede apreciar que la solución de Hanoi sigue un patrón especial. De hecho, la solución a la ecuación de recurrencia $h(n) = 2 \cdot h(n-1) + 1$ es $h(n) = 2^n - 1$, por lo que si hubiéramos llegado a ese resultado, podríamos utilizar la función `potencia` para resolver el problema.

6.3 Copo de nieve de Koch

Otro ejemplo interesante de recursión es el copo de nieve de Koch. El copo de nieve de Koch es un fractal cuya forma es similar a un copo de nieve. En la Figura 6.4 se puede apreciar un ejemplo.

El objetivo es describir el contorno de la figura hasta cierto nivel (puesto que el perímetro de la figura final es infinito). Para esto, es necesario describir un poco más en detalle la figura.

El fractal se genera a partir de un triángulo equilátero de lado s. A $s/3$ de distancia de un vértice se genera otro tríangulo equilátero, de lado $s/3$. A distancia $(s/3)/3$ del vértice del último tríangulo, se vuelve a generar otro más, y así sucesivamente. En la Figura 6.5 se pueden apreciar cuatro iteraciones del proceso.

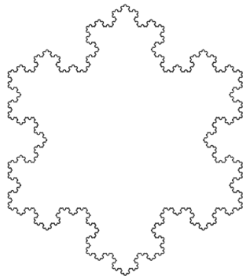


Figura 6.4: Copo de nieve de Koch.

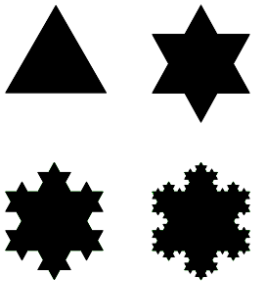


Figura 6.5: 4 primeras iteraciones de la generación del fractal. Fuente: Wikipedia.

¿Cómo podemos describir su contorno de manera recursiva? No es difícil observar que al generar la figura, al estar parados en algún triángulo de alguna iteración, a 1/3 del lado de distancia de un vértice comenzamos a generar ¡la misma figura! El caso base lo debemos definir nosotros, puesto que sin él, la figura se irá generando indefinidamente.

Para programar nuestro copo de nieve en Python, utilizaremos el módulo `Turtle` que viene incluido en el lenguaje. El módulo `Turtle` provee varias funciones que dibujan en la pantalla. Conceptualmente, se trata de una tortuga robótica que se mueve en la pantalla marcando una línea a su paso. Algunas de las funciones provistas son:

- `turtle.forward(size)`: Se mueve `size` pixeles en su dirección.
- `turtle.left(angle)`, `turtle.right(angle)`: La tortuga gira a la izquierda o a la derecha, respectivamente, dependiendo de su sentido, en `angle` grados.
- `turtle.speed(speed)`: Se establece la velocidad de la torturga. El parámetro `speed = 0` indica que se mueve a la máxima velocidad.
- `turtle.done()`: Se le indica que se han terminado las instrucciones para la tortuga.

Con estas funciones podemos indicarle cómo dibujar un fractal. Sin importar dónde comencemos, debemos dibujar un triángulo equilátero y al avanzar 1/3 de su lado, se dibuja un fractal nuevamente. La Figura 6.6 muestra cómo debería quedar nuestra implementación.

Analicemos el proceso de dibujar el copo de nieve. Observe que el copo de nieve se trata de dibujar un triángulo equilátero, por lo que podemos dividir el problema en dibujar sólo un lado (puesto que los otros dos son iguales, salvo por el ángulo de donde viene). Supongamos que tenemos una función `snowflake`que dibuja los tres lados. Cada lado debe ser dibujado usando la curva de Koch.

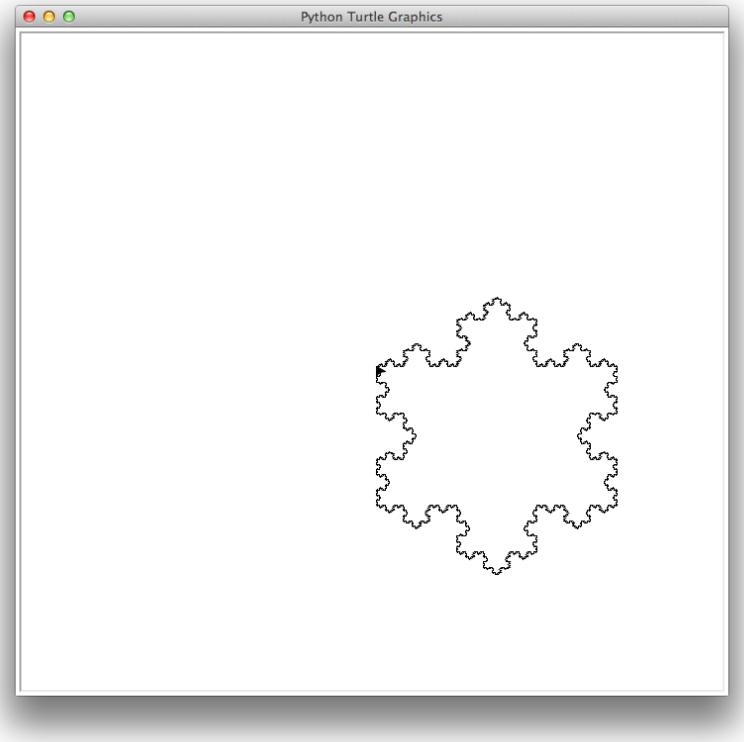


Figura 6.6: El resultado de dibujar el copo de nieve con Turtle

La función que dibuja cada lado la llamaremos `koch`, que representará la curva de Koch. Esta función debe dibujar cada lado del tríangulo actual. Esta función debería recibir el lado del triángulo inicial y el caso base, es decir, el tamaño mínimo a partir del cual no debe continuar la recursión. Llamemos a estos parámetros `size` y `min_size`, respectivamente.

Una implementación posible es la siguiente:

1. Avanzar $1/3 \times$ `size` en la dirección actual.
2. Girar a la izquierda 60 grados.
3. Dibujar la misma curva de Koch de lado $1/3 \times$ `size`.
4. Girar a la derecha 120 grados.
5. Dibujar la misma curva de Koch de lado $1/3 \times$ `size`.
6. Girar a la izquierda 60 grados.
7. Avanzar $1/3 \times$ `size` en la dirección actual.

Sin embargo, tiene un error. Si dibujamos esta curva tres veces y creamos el triángulo con `snowflake`, resultará en lo que se puede apreciar en la Figura 6.7. Al separar nuestra función en dos, una que dibuja un lado y la otra que usa la primera para dibujar los tres lados, hemos perdido información. En particular, los vértices de los triángulos que se forman indirectamente al juntar las tres curvas (que son iguales al primero, al ser equiláteros del mismo tamaño) no generan sub-triángulos y no se forma la figura del copo de nieve. Para esto debemos generar más sub-triángulos incluso en

esos vértices.

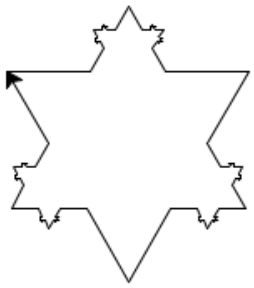


Figura 6.7: Primer intento del copo de nieve, juntando las tres curvas de Koch de la primera implementación.

Para solucionar este problema, modifiquemos nuestro algoritmo para que sólo dibuje una línea recta cuando llegamos al límite del tamaño:

1. Si `min_size` $\geq$ `size`/ 3, avanzar `size` en la dirección actual.
2. Si no:
   - Dibujar la misma curva de Koch de lado 1/3 $\times$ `size`.
   - Girar a la izquierda 60 grados.
   - Dibujar la misma curva de Koch de lado 1/3 $\times$ `size`.
   - Girar a la derecha 120 grados.
   - Dibujar la misma curva de Koch de lado 1/3 $\times$ `size`.
   - Girar a la izquierda 60 grados.
   - Dibujar la misma curva de Koch de lado 1/3 $\times$ `size`.

Al graficar esta implementación, resultará en lo que se puede ver en la Figura 6.8.

Figura 6.8: La curva de Koch para un lado del triángulo.

La implementación en Python final:

Contenido del archivo koch.py

```
1 import turtle
2
3 # koch: int int -> None
```

Contenido del archivo koch.py (cont)

```
4  # dibuja la curva de koch de largo size
5  # y largo minimo min_size
6  # ejemplo: koch(320, 1)
7  def koch(size, min_size):
8          if (size / 3 < min_size):
9                  # caso base
10                 turtle.forward(size)
11         else:
12                 # caso recursivo
13                 koch(size / 3, min_size)
14                 turtle.left(60)
15                 koch(size / 3, min_size)
16                 turtle.right(120)
17                 koch(size / 3, min_size)
18                 turtle.left(60)
19                 koch(size / 3, min_size)
20
21 # snowflake: int int ->
22 # dibuja el copo de nieve de Koch
23 # de un triangulo de lado size
24 # y lado minimo min_size
25 # ejemplo: snowflake(320, 1)
26 def snowflake(size, min_size):
27         koch(size, min_size)
28         turtle.right(120)
29         koch(size, min_size)
30         turtle.right(120)
31         koch(size, min_size)
32
33 # ejemplo de uso
34 turtle.speed(0)
35 snowflake(320, 1)
36 turtle.done()
```

6.4 Receta de diseño para la recursión

Cuando escribimos funciones recursivas, es necesario seguir una versión distinta de la receta de diseño. En efecto, antes de escribir el codigo, tenemos que seguir varios pasos bien definidos.

1. Escribir varios ejemplos de uso de la funcion (incluyendo parametros y resultado).
   2 potencia 4 = 16
   2 potencia 0 = 1
   2 potencia 3 = 8

2. Decidir cual de los argumentos va a tener una descomposición recursiva.
   El argumento con el procedimiento recursivo es la potencia

3. Entender cual es el caso de base para este argumento.
   Cuando la potencia es igual a 0, el resultado es uno

4. Entender cual es el paso recursivo, como se descompone el problema
   Uno resta una a la potencia, y multiplica
5. Nombrar cada pieza del problema
   Tenemos una "base", y una "potencia"
6. Aplicar el problema a uno de los ejemplos
   (a) Tomar un ejemplo: **2 potencia 3 = 8**
   (b) Determinar los parametros y el resultado
   Los parametros son 2 y 3, el resultado 8
   (c) Determinar cuales de los parametros son piezas del problema
   2 es la base, y 3 es la potencia
   (d) Cual es la repuesta para la llamada recursiva?
   Para reducir el problema, tenemos que restar uno a la potencia, entonces la llamada recursiva es: 2 potencia 2 = 4
   (e) Determinar como se combinan los elementos para determinar la respuesta final
   dado que 2 potencia 2 = 4, multiplicamos este resultado por la base (2), y tenemos el resultado final, 8
7. Ocupar la receta de diseño normal, tomando en cuenta que el patrón de base es una función condicional, una rama siendo el caso de base, la otra siendo el caso recursivo.

Capítulo 7

Testing y Depuración de Programas

En este capítulo veremos formas y técnicas de testing y depuración de programas, basándonos en el diseño por contrato visto en el capítulo de Receta de Diseño.

Tal como vimos en aquel capítulo, es necesario probar que la función que definamos cumpla con el contrato estipulado. Estas pruebas deben asegurar con suficiente certeza de que la función cumple su objetivo de acuerdo a los parámetros ingresados. En este punto el contrato es muy importante, ya que especifica los tipos de datos y sus dominios que serán considerados dentro de la función. No es factible probar todos los posibles parámetros de una función, pero el contrato disminuye considerablemente estas opciones. De los casos restantes, debemos escoger sólo los casos más representativos.

Por ejemplo, consideremos la función `maximo`, que tiene el siguiente contrato:

```
# maximo: num num -> num
# devuelve el mayor de ambos numeros, a y b
# ejemplo: maximo(2, 4) devuelve 4
def maximo(a, b):
        ...
```

El contrato establece que los parámetros de la función deben ser numéricos, por lo que el siguiente no sería un buen test:

```
assert maximo('hola', 5) == 'hola'
```

En cambio, este sería un buen test para la función:

```
assert maximo(10, 20) == 20
```

Consideremos la función `potencia` definida en el capítulo de Recursión. Esta función acepta como parámetros un número como base, y un entero como exponente. Podríamos probar distintos casos de esta forma:

```
assert potencia(2, 4) == 16
assert potencia(1.5, 3) == 3.375
assert potencia(10, 3) == 1000
```

Sin embargo, tal como definimos potencia, tanto en código como matemáticamente, hay casos llamados *de borde*, es decir, extremos dentro del dominio de datos que acepta, que serían relevantes

de usar en los tests. En el caso de la potencia, la definición cambia de acuerdo al valor del exponente, por lo que el caso de borde sería un buen test para nuestra función:

```
assert potencia(10000000, 0) == 1
```

Si tenemos una función, sea o no recursiva, con casos de borde o extremos dentro del dominio de datos, debemos testearla en esos casos. En general, debemos probar nuestra función en **casos representativos**. Por ejemplo, en `potencia`, el primer y tercer test son redundantes. El segundo es relevante ya que usamos otro tipo de datos para la base. El último test definido también es relevante, ya que prueba un caso de borde.

7.1 Afirmaciones (*assertions*)

Hasta el momento hemos visto usar la sentencia `assert` (afirmación) de Python para probar nuestras funciones. La sintaxis específica es la siguiente:

```
assert <condicion>
```

Como observación, `assert` no es una expresión. No puede ser evaluada y asignada a una variable, sino que puede ser vista como una sentencia o una palabra clave de Python que realiza una acción. Por otra parte, `<condicion>` sí corresponde a una expresión, por lo que se puede usar cualquier expresión que se evalúe a un valor de tipo `Boolean` en el lugar de `<condicion>`. Por ejemplo:

```
assert True
assert 10 < 12
assert a == b and (c < d or a < b)
```

Cuando la condición se evalúa a `True`, la afirmación no hace nada y el programa puede continuar. Cuando la condición evalúa a `False` (es decir, si no se cumple), la afirmación arroja un error y el programa termina:

```
>>> assert False
```

```
Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
     AssertionError
```

Esto es útil para probar el buen funcionamiento de nuestras funciones. Claramente, el comportamiento de las afirmaciones sugiere que Ud. debe escribirlas **antes** de escribir el código de su función, con valores que Ud. haya calculado antes. Recuerde que el código es una representación de su solución, no la solución.

El uso de distintos operadores lógicos para `assert` nos ayuda a escribir mejores tests. Observe que el uso de más operadores no es una característica especial de la afirmación, sino que corresponde a que la condición que se pasa es una expresión, y en ella se puede usar cualquier operador booleano.

En particular, podemos utilizar operadores no sólo de igualdad, sino también de comparación (`<` o `>`, etc). Por ejemplo, suponga que tenemos una función que calcula aleatoriamente un número entre 1 y 10. ¿Cómo hacemos un test para eso, si cada evaluación de la función resultará en un valor distinto? En este caso, la observación clave está en que no nos importa qué valor tome la función, sino el *rango* de valores. Por ejemplo:

```
import random
# escogeAlAzar: -> int
# Devuelve un numero al azar entre 1 y 10
def escogeAlAzar():
        # random.random() retorna un numero al azar entre 0 y 1
        return int(10 * random.random()) + 1

# Test
assert escogeAlAzar() <= 10
assert escogeAlAzar() >= 1
```

Si nuestra función retorna de un tipo booleano, no es necesario indicar la igualdad. Si tenemos una función `esPar` que retorna `True` si su argumento es un entero par, y `False` si no, puede ser probada de la siguiente forma:

```
assert esPar(4)
assert esPar(5) == False
```

Dado que `esPar` evalúa a un booleano, no es necesario indicar la igualdad. Si retorna `False`, es necesario indicarla, puesto que en caso contrario la afirmación arrojará error al no cumplirse.

7.2 Testear con números reales

En Python, y en la gran mayoría de los lenguajes de programación, los números reales son representados en el computador utilizando *aritmética de punto flotante*. Los números reales son infinitos, pero una máquina es finita y por lo tanto es necesario representar sólo un subconjunto finito de los números reales en un computador.

Estos números están sujetos a *errores de precisión*. Es decir, algunas operaciones no serán exactas debido a que no existe una representación para *cada* número posible. Puede intentar este ejemplo en el intérprete:

```
>>> 0.1 + 0.2
 -> 0.30000000000000004
```

Los errores de precisión también pueden propagarse. Por ejemplo, considere dos cantidades, a_0 y a_1, que en su representación en un computador poseen errores e_0 y e_1 respectivamente (como `0.1+0.2` en Python). Si multiplicamos estos valores, el error se amplifica:

$$(a_0 + e_0) \cdot (a_1 + e_1) \approx a_0 a_1 + a_0 e_1 + a_1 e_0$$

Con esto en mente, ¿cómo podemos estar seguros de que una función que manipule números de punto flotante está correcta con respecto a nuestras pruebas? Para esto utilizamos una tolerancia que denotaremos ε, que nos servirá para comparar dentro de un rango de valores.

Implementemos la función *distancia euclidiana* que calcula la distancia entre dos puntos dados por sus coordenadas, x_0, y_0, x_1, y_1.

```
# distancia:  num num num num -> num
# calcula la distancia euclidiana entre (x0, y0) y (x1, y1)
# ejemplo:  distancia(1, 0, 4, 0) devuelve 3.0
def distancia(x0, y0, x1, y1):
```

```
5           dx = (x1 - x0)
6           dy = (y1 - y0)
7           return (dx ** 2 + dy ** 2) ** 0.5
```

Consideremos las siguientes expresiones:

```
>>> d1 = distancia(0.1, 0.2, 0.2, 0.1)
 -> 0.14142135623730953
>>> d2 = distancia(1, 2, 2, 1)
 -> 1.4142135623730951
>>> 10 * d1 == d2
 -> False
```

Conceptualmente, la última expresión debía ser verdadera, pero fue evaluada a `False` por errores de precisión. Para esto usamos el valor de tolerancia ε para evitar estos problemas:

```
# cerca: num num num -> bool
# retorna True si x es igual a y con
# precision epsilon
def cerca(x, y, epsilon):
        diff = x - y
        return abs(diff) < epsilon
```

El valor de `epsilon` dependerá de cuánta precisión necesitemos para nuestro programa. La función `abs` devuelve el valor absoluto de su argumento: `abs(1) == 1`, `abs(-1) == 1`, etc. Se necesita esta función porque sino un valor muy grande de `d2` podría hacer verdadera la afirmación, aun cuando lógicamente sea falsa. Ahora podemos definir nuestros tests:

```
# Tests
tolerancia = 0.0001
assert cerca(distancia(0, 0, 4, 0), 4.0, tolerancia)
assert cerca(distancia(0, 1, 1, 0), 1.4142, tolerancia)
```

7.3 Ejemplo: cálculo de la raíz cuadrada

Suponga que requiere evaluar dentro de una expresión la raíz cuadrada de un número. El módulo `math` de Python provee la función `math.sqrt`, pero para efectos de este ejercicio supondremos que no disponemos de `math`.

Herón de Alejandría, un matemático griego del siglo I, propuso un método para calcular una aproximación de la raíz cuadrada de un número. El algoritmo está basado en la siguiente idea general: si z es una estimación de $\sqrt{x}$, pero superior a $\sqrt{x}$, entonces el valor de x/z sería una estimación menor que $\sqrt{x}$. Si uno calcula el promedio de estas dos estimaciones, se logra una estimación más cercana del valor de $\sqrt{x}$. Por otra parte, si z es inferior a $\sqrt{x}$, entonces x/z sería superior a $\sqrt{x}$, y el promedio entre x y z estaría más cerca del valor de $\sqrt{x}$. El método de Herón consiste en repetir este proceso hasta alcanzar una precisión suficiente, lo que en el siglo I, sin computadores, era un proceso muy largo de realizar a mano. Por suerte, ¡nosotros tenemos hoy en día computadores para ayudarnos con los cálculos!

Vamos a diseñar una función recursiva para calcular la raíz cuadrada de un número:

- Los argumentos de la función recursiva son: el número positivo x, una estimación de $\sqrt{x}$, y un nivel de precisión `epsilon`. Para verificar que x es un número positivo utilizaremos una *precondición*, que implica agregar un test dentro de la función. En Python esto se puede implementar con `assert`.
- Caso base: si el cuadrado de la estimación está a distancia `epsilon` de x, se retorna el valor de la estimación. Para esto, utilizaremos la función `cerca`.
- Caso recursivo: Se calcula una mejor estimación de acuerdo al método de Heron, y se realiza el llamado recursivo.
- La función recursiva se llamará `heron_r` y será una función auxiliar a la función `heron`, que hará el primer llamado a la función recursiva con una estimación inicial de $\sqrt{x}$. Nota: el elegir como estimación un valor cercano a $\sqrt{x}$ hace que el programa termine más rápido, pero usar un valor genérico como 1 también funciona.

Siguiendo la receta de diseño, la función para calcular la raíz cuadrada usando el método de Heron queda como sigue:

```
# heron_r: num num num -> num
# helper funcion de heron, calcula la raiz cuadrada de x
# con precision epsilon y valor inicial de estimacion
# ejemplo: heron(2,0.00001,1) devuelve 1.414215...
def heron_r(x, epsilon, estimacion):
        # pre-condicion
        assert x > 0
        if cerca(estimacion * estimacion, x, epsilon):
                #caso base
                return estimacion
        else:
                # caso recursivo
                mejor_estimacion = (estimacion + x / estimacion) / 2
                return heron_r(x, epsilon, mejor_estimacion)

# heron: num num -> num
# calcula la raiz cuadrada de x, con precision epsilon
# ejemplo:
# heron(2,0.1) devuelve 1.416...
# heron(2,0.00001) devuelve 1.414215...
def heron(x, eps):
        return heron_r(x, eps, x / 2.0)

import math
assert cerca(heron(2, 0.1), math.sqrt(2), 0.1)
assert cerca(heron(3, 0.01), math.sqrt(3), 0.01)
assert cerca(heron(4, 0.001), math.sqrt(4), 0.001)
```

Unidad II: Programación Funcional

Capítulo 8

Datos Compuestos[1]

Hasta el momento hemos visto únicamente cómo operar con valores simples (esto es, con números, con strings, y con valores lógicos). Sin embargo, en computación es recurrente el tener que manipular valores *compuestos* que corresponden a alguna combinación sobre estos valores simples. Por ejemplo, supongamos que queremos calcular la suma entre dos fracciones dadas.

Es claro que una primera alternativa para resolver este problema es definir una función sobre *cuatro* valores enteros, y operar sobre ellos como es usual en matemática:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Así pues, la función que permite resolver este problema es:

```
# sumaFracciones:  int int int int -> float
# calcula la suma entre dos fracciones a/b y c/d
# ejemplo:  sumaFracciones(1, 2, 3, 4) devuelve 1.25
def sumaFracciones(a,b,c,d):
    return (a * d + b * c) * 1.0 / b * d

# Test (usa la funcion cerca)
epsilon = 0.000001
assert cerca(sumaFracciones(1, 2, 3, 4), 1.25, epsilon)
```

Sin embargo, notemos que en el ejemplo anterior, el valor que devuelve la función NO es una fracción, sino un número real que corresponde a la *representación decimal* del resultado. ¿Cómo podemos entonces indicarle al computador que queremos manipular fracciones?

8.1 Estructuras (*structs*)

Una *estructura* (struct) es un tipo de datos que permite encapsular un conjunto fijo de valores (de uno o más tipos), representados por atributos, para conformar un único valor compuesto. En efecto, podemos representar una *fracción* como una estructura formada por dos atributos: un *numerador* y un *denominador*.

[1]Parte de este capítulo fue traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

Para trabajar con estructuras en este curso, disponemos del módulo `estructura.py` que contiene las definiciones básicas para poder crear estructuras.

```
1 import estructura
2 estructura.crear("nombre", "atributo1 atributo2 ...  atributoN")
```

En el ejemplo anterior, importamos el módulo `estructura` y utilizamos la instrucción `crear` para crear una nueva estructura. Notemos que este comando recibe dos parámetros: el *nombre* de la estructura, y un texto con los distintos *atributos* que la representan, cada uno separado por un espacio simple.

Así, por ejemplo, para crear una estructura que represente a una fracción, proseguimos de la siguiente manera:

```
1 import estructura
2
3 estructura.crear("fraccion", "numerador denominador")
```

Finalmente, notemos que podemos acceder directamente a los distintos atributos que definen a la estructura:

```
1 >>> a = fraccion(1, 2)
2  -> fraccion(numerador=1, denominador=2)
3 >>> a.numerador
4  -> 1
5 >>> a.denominador
6  -> 2
```

En la siguiente sección veremos en detalle cómo diseñar apropiadamente una estructura. Luego, completaremos nuestro ejemplo inicial para ver cómo crear funciones que manipulen estructuras.

8.2 Receta de diseño para estructuras

Recordemos que en el Capítulo 3 vimos cómo escribir la receta de diseño para funciones simples. En el Capítulo 5 la extendimos para el caso en que utilizamos condiciones en la definición de una función, y en el Capítulo 6 vimos el caso de funciones con recursión. Ahora veremos qué es lo que debemos hacer cuando nos vemos enfrentados a definir una función que opera sobre estructuras.

En primer lugar, debemos reconocer *desde el planteamiento del problema* cuáles son las estructuras que vamos a necesitar. En efecto, la regla a seguir es que cada vez que necesitemos operar sobre un conjunto de valores relacionados entre sí, debemos escribir una estructura que los encapsule. Si no utilizamos estructuras, lo más probable es que perdamos rápidamente la pista de qué valor pertenece a qué elemento del programa, especialmente cuando tenemos funciones grandes que procesan mucha información.

En segundo lugar, se deben usar estructuras para organizar las funciones. Para ello, utilizaremos una *plantilla* que la acoplaremos a la receta de diseño que ya conocemos. De esta manera, debemos asegurarnos en todo momento que la definición de la estructura efectivamente se corresponde con la plantilla. Los pasos a incorporar en la nueva receta de diseño son:

8.2.1 Diseñar estructuras

Antes de poder desarrollar una función, debemos entender cómo representar la información dada en el problema. Para lograr esto, buscamos descripciones de conjuntos de valores interrelacionados relevantes, y por cada una de ellas, diseñamos una estructura.

8.2.2 Plantilla

Una función que opera sobre datos compuestos, por lo general opera sobre las componentes de las estructuras que recibe como entrada. Para poder recordar claramente cuáles son estas componentes, debemos diseñar una plantilla. Así, una plantilla es un encabezado y un cuerpo de función que lista todas las posibles combinaciones de expresiones que se pueden generar con las entradas de la función. En otras palabras, una plantilla expresa lo que sabemos sobre las entradas, pero aún no nos dice nada sobre cómo va a ser la salida de la función. Luego, utilizamos la plantilla para cualquier función que consuma los mismos tipos de parámetros.

8.2.3 Cuerpo de la función

La plantilla nos debería dar todas las pistas que necesitamos para escribir apropiadamente la función. Al igual que las funciones que hemos estado desarrollando hasta ahora, el objetivo de esta etapa es formular una expresión que compute la respuesta pedida a partir de la información disponible utilizando otras funciones, o bien, las operaciones primitivas del lenguaje de programación. La plantilla nos recuerda cuáles son los atributos disponibles de los parámetros compuestos, y cómo los podemos relacionar entre ellos. Finalmente, escribamos siguiendo la receta de diseño la función que dadas dos fracciones retorne la suma de ellas:

Contenido del archivo fraccion.py

```
1  import estructura
2
3
4  # Diseno de la estructura
5  # fraccion: numerador (int) denominador (int)
6  estructura.crear("fraccion", "numerador denominador")
7
8  # Contrato
9  # sumaFracciones:  fraccion fraccion -> fraccion
10
11 # Proposito
12 # crear una nueva fraccion que corresponda a la suma
13 # de dos fracciones f1 y f2
14
15 # Ejemplo:
16 # sumaFracciones(fraccion(1, 2), fraccion(3, 4))
17 # devuelve fraccion(10, 8)
18
19
20 # Plantilla
21 # def funcionConFracciones(fraccion1, fraccion2):
22 #       ...  fraccion1.numerador ...  fraccion2.numerador ...
23 #       ...  fraccion1.numerador ...  fraccion2.denominador ...
```

Contenido del archivo fraccion.py (cont)

```
24 #         ...  fraccion1.denominador ...  fraccion2.numerador ...
25 #         ...  fraccion1.denominador ...  fraccion2.denominador ...
26
27
28 # Cuerpo de la funcion
29 def sumaFracciones(f1, f2):
30         num = f1.numerador * f2.denominador \
31                 + f1.denominador * f2.numerador
32         den = f1.denominador * f2.denominador
33         return fraccion(num, den)
34 # Tests
35 f12 = fraccion(1, 2)
36 f34 = fraccion(3, 4)
37 assert sumaFracciones(f12, f34) == fraccion(10, 8)
```

Capítulo 9

Estructuras de Datos Recursivas[1]

Tal como lo vimos en el capítulo anterior, una de las maneras para representar información compuesta es usando estructuras. En efecto, las estructuras son útiles cuando sabemos cuántos datos queremos combinar. Sin embargo, en muchos otros casos, no sabemos cuántas cosas queremos enumerar, y entonces formamos una *lista*. Una lista puede tener un largo arbitrario, esto es, contiene una cantidad finita pero indeterminada de elementos.

9.1 Listas

Para el manejo de listas, ocuparemos el módulo `lista.py`, implementado para efectos de este curso. Para poder ocuparlo, primero hay que importar todas sus funciones:

```
1 from lista import *
```

La definición de una lista es recursiva. Por una parte, una lista puede ser *vacía*, es decir no contiene ningún elemento. Esto se obtiene mediante el identificador `listaVacia`, provisto en el módulo `lista`. Por otra parte, una lista puede estar compuesta por elementos que contienen un *valor* y un enlace al resto de la lista. Con esto, es posible crear una lista más larga concatenando otras listas. En efecto, el construir una lista lo podemos entender como ir armando una cadena, uniendo sus eslabones uno por uno.

Así, en una lista distinguimos dos campos en su estructura: el `valor` y la lista `siguiente`. El campo *valor* puede ser de cualquier tipo (básico o compuesto), mientras que el campo *siguiente* es precisamente una lista, tal como los eslabones de una cadena. La definición de la estructura para listas está incluida en el módulo `lista.py`, note en particular en el contrato de la estructura su definición recursiva:

```
1 # Diseno de la estructura
2 # lista : valor (cualquier tipo) siguiente (lista)
3 estructura.crear("lista", "valor siguiente")
```

Para crear una lista nueva, el módulo provee la función `crearLista` que recibe dos parámetros: el `valor` del primer elemento de la lista y el `resto` de la lista. Veamos un ejemplo: supongamos que queremos formar una lista con los planetas del sistema solar. Primero comenzamos con un eslabón de la lista que sólo contiene a Mercurio:

[1] Traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

```
crearLista("Mercurio", listaVacia)
```

Luego viene el planeta Venus:

```
crearLista("Venus", crearLista("Mercurio", listaVacia))
```

A continuación la Tierra, y así sucesivamente:

```
crearLista("Tierra", crearLista("Venus", crearLista("Mercurio", listaVacia)))
```

En toda lista distinguimos dos elementos: la *cabeza* y la *cola*. La cabeza de una lista es el valor que está al frente de la lista (es decir, el primer valor disponible). La cola de una lista es todo lo que va encadenado a la cabeza. Así, en nuestro último ejemplo, la *cabeza* de la lista es el string `"Tierra"`, mientras que la *cola* es la lista formada por el eslabón `(Venus, (Mercurio, (listaVacia)))`. En efecto, estos elementos son proporcionados por el módulo:

```
>>> L = crearLista("Tierra", crearLista("Venus",
crearLista("Mercurio", listaVacia)))
>>> cabeza(L)
 -> "Tierra"
>>> cola(L)
 -> lista(valor='Venus', siguiente=
lista(valor='Mercurio', siguiente=None))
```

En general, una lista no tiene por qué contener valores de un único tipo. Por ejemplo, la siguiente lista es completamente válida (aunque no tiene mucho sentido sin conocer el contexto en el que fue creada).

```
crearLista("Pepito", crearLista(41, crearLista(True, listaVacia)))
```

Así, en esta lista el primer elemento es un string, el segundo es un número, y el último es un booleano. Podríamos pensar que esta lista corresponde a la representación de un registro de personal que tiene el nombre de un empleado, la edad, y si este empleado tiene aún días disponibles para tomarse vacaciones.

Supongamos ahora que nos dan una lista de números. Una de las primeras cosas que nos gustaría hacer es sumar los números de esta lista. Más concretamente, supongamos que estamos únicamente interesados en listas de tres números. Así, una `listaDeTresNumeros` es una lista que se puede crear con la instrucción `crearLista(x, crearLista(y, crearLista(z, listaVacia)))`, donde `x`, `y`, `z` son tres números.

Escribamos el contrato, el propósito, el encabezado y ejemplos para una función que sume estos números siguiendo la receta de diseño:

```
# sumaTres: listaDeTresNumeros -> num
# suma los tres numeros en unaLista
# sumaTres(crearLista(2, crearLista(1, crearLista(0, listaVacia))))
# devuelve 3
# sumaTres(crearLista(0, crearLista(1, crearLista(0, listaVacia))))
# devuelve 1
def sumaTres(unaLista):
        ...
```

Sin embargo, al definir el cuerpo de la función nos topamos con un problema. Una lista de la manera en que la hemos construido es una estructura. Así, deberíampos proveer una plantilla con las distintas alternativas que se pueden elegir para construir las expresiones. Por desgracia, aún no sabemos cómo seleccionar los elementos de una lista. Por otro lado, recordemos que a través de la función `cabeza` podemos acceder al primer elemento de una lista, mientras que la función `cola` nos entrega el resto. Veamos más en detalle qué nos devuelven las combinaciones de estas operaciones:

```
>>> cabeza(crearLista(10, listaVacia))
 -> 10
>>> cola(crearLista(10, listaVacia)) #listaVacia
 ->
>>> cabeza(cola(crearLista(10, crearLista(22, listaVacia))))
 -> 22
```

En efecto, la expresión presentada en el último ejemplo se evalúa paso a paso a través de la siguiente secuencia:

```
>>> cabeza(cola(crearLista(10, crearLista(22, listaVacia))))
 -> cabeza(crearLista(22, listaVacia))
 -> 22
```

La clave está en considerar que `crearLista(unValor, unaLista)` es un valor *como si fuera de tipo lista*. Así, como siempre, comenzamos con las evaluaciones desde adentro hacia afuera, tal como lo hemos hecho hasta ahora.

Luego, utilizando las funciones `cabeza` y `cola` disponibles en el módulo para manejar listas, podemos escribir la plantilla para `sumaTres`:

```
# sumaTres: listaDeTresNumeros -> num
# suma los tres numeros en unaLista
# def sumaTres(unaLista):
    ...  cabeza(unaLista) ...
    ...  cabeza(cola(unaLista)) ...
    ...  cabeza(cola(cola(unaLista))) ...
```

Notemos que las tres expresiones presentadas en la plantilla nos recuerdan que el argumento de entrada de la función, `unaLista`, efectivamente contiene tres componentes y se pueden extraer con alguna de esas expresiones.

9.2 Definición de datos para listas de largo arbitrario

Supongamos que queremos representar el inventario de una juguetería que vende muñecas, sets de maquillaje, payasos, arcos, flechas, y pelotas. Para construir tal inventario, el dueño de la tienda debería comenzar con una hoja de papel en blanco, y luego ir llenándola con los nombres de los juguetes que tiene en los distintos estantes.

De la sección anterior vimos que representar una lista de juguetes es simple: basta con crear una lista de strings, encadenando los eslabones uno por uno. La secuencia para formar la lista podría ser, por ejemplo:

```
listaVacia
crearLista(pelota, listaVacia)
```

```
crearLista(flecha, crearLista(pelota, listaVacia))
```

Sin embargo, para una tienda real, esta lista de seguro contendrá muchos más elementos, y la lista crecerá y disminuirá a lo largo del tiempo. Lo cierto es que en ningún caso podremos decir por adelantado cuántos elementos distintos contendrá la lista. Luego, si quisiéramos desarrollar una función que consuma tales listas, no podremos simplemente decir que la lista de entrada tendrá uno, dos o tres elementos.

En otras palabras, necesitamos una definición de datos que precisamente describa la clase de listas que contenga una cantidad arbitraria de elementos (por ejemplo, strings). Desafortunadamente, hasta ahora sólo hemos visto definiciones de datos que tienen un tamaño fijo, que tienen un número determinado de componentes, o listas con una cantidad delimitada de elementos.

Note que todos los ejemplos que hemos desarrollado hasta ahora siguen un patrón: *comenzamos con una lista vacía, y empezamos a encadenar elementos uno a continuación del otro.* Notemos que podemos abstraer esta idea y plantear la siguiente definición de datos:

Una *lista de strings* es:

1. una lista vacía, `listaVacia`, o bien
2. `crearLista(X, Y)`, donde `X` es un string, y `Y` es una *lista de string.*

Como ya mencionamos anteriormente, nos debería llamar la atención que esta estructura se define *en términos* de sí misma en el segundo ítem. Esto es, para definir una lista, usamos como elemento constituyente otra lista (tal como una cadena se forma al encadenar eslabones entre sí). Estas definiciones se llaman *definiciones recursivas.*

9.3 Procesar listas de largo arbitrario

Tal como vimos anteriormente, en el caso de una tienda nos gustaría poder contar con un inventario de todo el stock de elementos a la venta. En particular, nos interesa mantener este registro en un computador para que un empleado pueda, por ejemplo, verificar si un juguete está o no disponible. Supongamos, pues, que existe la función `hayPelotas` que devuelve `True` si en la tienda quedan pelotas, y `False` en caso contrario. ¿Cómo podríamos implementar esta función?

Dado que ya tenemos una descripción rigurosa de qué es lo que se espera de esta función, sigamos la receta de diseño para escribir su contrato, encabezado y propósito:

```
# hayPelotas: lista(str) -> bool
# determinar si el string "pelota" esta en la lista unaLista
def hayPelotas(unaLista):
        ...
```

Siguiendo la receta de diseño, debemos ilustrar el funcionamiento de la función a través de unos ejemplos. Primero, debemos determinar cuál es la entrada más simple que puede recibir la función: la lista vacía (`listaVacia`). En este caso, dado que la lista no contiene ningún elemento, es natural que el string `"pelota"` no esté, y por tanto, la salida debe ser `False`:

```
>>> hayPelotas(listaVacia)
 -> False
```

Luego, consideremos listas de un solo elemento:

```
>>> hayPelotas(crearLista("pelota", listaVacia))
 -> True
>>> hayPelotas(crearLista("muneca", listaVacia))
 -> False
```

Y finalmente, veamos el funcionamiento de la función en listas más generales (con más de un elemento):

```
>>> hayPelotas(crearLista("arco", crearLista("flecha", \
        crearLista("muneca", listaVacia))))
 -> False
>>> hayPelotas(crearLista("soldadito", crearLista("pelota", \
        crearLista("oso", listaVacia))))
 -> True
```

El paso siguiente es diseñar la plantilla de la función que se corresponda con la definición del tipo de datos. En este caso, dado que la definición de una lista de strings tiene dos cláusulas (lista vacía o una lista de strings), la plantilla se debe escribir usando un bloque condicional `if`:

```
# hayPelotas: lista(str) -> bool
# determinar si el string "pelota" esta en la lista unaLista
# def hayPelotas(unaLista):
#         if vacia(unaLista):
#                 ...
#         else:
#                 ...  cabeza(unaLista)
#                 ...  cola(unaLista) ...
```

En este caso, primero preguntamos si la lista está vacía (usando la función `vacia` provista por el módulo `lista.py`). Si no, podemos acceder a la `cabeza` y `cola` de la misma.

Ahora que disponemos de la plantilla para la función, podemos escribir el cuerpo de la misma. Para ello, abordaremos separadamente cada una de las dos ramas de la condición que se definen en la plantilla:

1. Si la lista está vacía, debemos retornar el valor `False` (pues el string no puede estar en una lista vacía).
2. Si no, debemos preguntar si la cabeza corresponde al string buscado. En este caso, surgen dos alternativas:
   (a) El string efectivamente corresponde a la cabeza: entonces retornamos el valor `True`.
   (b) Si no, debemos buscar recursivamente dentro de la lista que corresponde a la cola de la lista inicial.

Luego, la definición completa del cuerpo de la función es:

```
def hayPelotas(unaLista):
        if vacia(unaLista):
                return False
```

```
4         else:
5                 if cabeza(unaLista) == "pelota":
6                         return True
7                 else:
8                         return hayPelotas(cola(unaLista))
```

9.3.1 Receta de diseño para funciones con definiciones de datos recursivas

Las definiciones de datos recursivas pueden parecer mucho más complejas que aquellas para datos compuestos, pero como hemos visto en los ejemplos anteriores la receta de diseño que conocemos sigue funcionando. Repasemos la nueva receta de diseño para definiciones de datos recursivas, que generaliza aquella para datos compuestos. La partes nuevas de la receta se preocupan de descubrir cuándo se necesita una definición de datos recursiva, cómo generar una plantilla y como definir el cuerpo de la función:

- Diseño y análisis de datos: Si el enunciado de un problema involucra información compuesta de tamaño arbitrario, se necesita una definición de datos recursiva. Hasta ahora, sólo hemos visto listas, pero veremos otras definiciones más adelante. Para que una definición de datos recursiva sea válida, debe satisfacer dos condiciones. Primero, debe contener al menos dos cláusulas. Segundo, al menos una de ellas *no debe referirse* de vuelta a la definición.

  Ejemplo: una lista de símbolos contiene dos cláusulas, ya que puede ser

  1. una lista vacía, o
  2. una lista `crearLista(s, lista(simbolos))`, donde `s` es un simbolo y `lista(simbolos)` es una lista de simbolos.

- Plantilla: Se formula con expresiones condicionales. Debe haber una condición por cada cláusula en la definición de datos recursiva, escribiendo expresiones adecuadas en todas las condiciones que procesen datos compuestos.

  Ejemplo: la plantilla para una función que procesa una lista de simbolos seria

```
1 # def procesarLista(unaLista):
2 #         if vacia(unaLista):
3 #                 ...
4 #         else:
5 #                 ...   cabeza(unaLista)
6 #                 ...   procesarLista(cola(unaLista)) ...
```

- Cuerpo de la función: Se empieza por los casos base (aquellos que no tienen llamados recursivos). Luego, se continúa con los casos recursivos. Debemos recordar cuáles de las expresiones en la plantilla se calculan, y suponemos que para los llamados recursivos la función retorna el valor esperado (hipótesis de inducción). Finalmente, se combinan los valores obtenidos de los llamados recursivos dependiendo del problema (sumarlos, calcular el mínimo, calcular el máximo, etc.).

  Ejemplo: supongamos que queremos implementar la función `cuantos`, que calcula cuántos símbolos contiene la lista de simbolos. Siguiendo la receta, tenemos que para el caso base (lista vacia) la respuesta es 0. Para el caso recursivo, tenemos que la lista contiene un símbolo (la cabeza) más los simbolos que contenga el resto de la lista, es decir, basta con sumar 1 al resultado del llamado recursivo para obtener el valor final.

```
def cuantos(unaLista):
          if vacia(unaLista):
                  return 0
         else:
                  return 1 + cuantos(cola(unaLista))
```

- Combinar valores: El paso de combinar valores puede consistir en una expresión a evaluar (como en el ejemplo anterior), o puede requerir preguntar algo sobre el primer objeto en la lista (en cuyo caso, puede ser necesaria una condición `if` anidada), o puede requerir definir funciones auxiliares (por ejemplo, si se quiere calcular el mínimo de la lista).

9.4 Funciones que producen listas

Recordemos la función `sueldo` que definimos en la Sección 2.2:

```
# sueldo: int -> int
# calcular el sueldo de un trabajador
# (a $4.500 la hora) que ha trabajado h horas
def sueldo(h):
        return 4500 * h
```

Esta función recibe como parámetro el número de horas trabajadas por un empleado, y produce su sueldo semanal. Por simplicidad, supondremos que todos los empleados ganan lo mismo por hora, es decir, $4.500. Sin embargo, una empresa no está necesariamente interesada en una función como `sueldo`, que calcula el sueldo de un solo empleado, sino más bien en una función que calcule el sueldo total de todos sus empleados (sobre todo si hay muchos).

Llamemos a esta función `listaSueldos`, tal que recibe una lista de cuántas horas los empleados de la compañía han trabajado, y devuelva una lista de los sueldos semanales por cada uno de ellos. Es claro que estas dos listas se pueden representar por *listas de enteros*. Dado que ya disponemos de una definición de datos para la entrada y la salida (modificando levemente la definición que vimos en la sección anterior), podemos comenzar inmediatamente a desarrollar la función.

```
# listaSueldos:  lista(int) -> lista(int)
# crear una lista de sueldos semanales desde
# una lista de horas trabajadas (listaHoras)
def listaSueldos(listaHoras):
        ...
```

Luego, como es usual, necesitamos proponer algunos ejemplos de entrada y salida:

```
>>> listaSueldos(listaVacia)
 -> listaVacia
>>> listaSueldos(crearLista(10, listaVacia))
 -> (45000, listaVacia)
>>> listaSueldos(crearLista(44, crearLista(10, listaVacia)))
 -> (198000, (45000, listaVacia))
```

Y, similar al ejemplo anterior, la plantilla a utilizar en la definición de la función es:

```
# def listaSueldos(listaHoras):
# if (vacia(listaHoras) == True):
#         ...
# else:
#         ...   cabeza(listaHoras)
#         ...   listaSueldos(cola(listaHoras)) ...
```

Ahora que tenemos definida la plantilla podemos proceder a completar el cuerpo de la función. De los ejemplos, tenemos la siguiente primera aproximación:

1. Si `listaHoras` está vacía, entonces tenemos que devolver una lista vacía.
2. Si no, creamos una lista donde la cabeza corresponde a aplicar la función `sueldo` a la cabeza de `listaHoras`, y la encadenamos a la lista que se forma al aplicar recursivamente la función con la cola de `listaHoras`.

Con lo anterior, podemos escribir finalmente el cuerpo de la función:

```
# listaSueldos:  lista(int) -> lista(int)
# crear una lista de sueldos semanales desde una
# lista de horas trabajadas (listaHoras)
def listaSueldos(listaHoras):
        if vacia(listaHoras):
                return listaVacia
        else:
                return crearLista(sueldo(cabeza(listaHoras)), \
                        listaSueldos(cola(listaHoras)))
```

9.5 Listas que contienen estructuras

Volvamos al ejemplo del inventario en la juguetería. En un principio, hicimos la suposición algo ingenua que un inventario consistía únicamente de una lista de elementos. Sin embargo, en registros más formales, se requiere no sólo contar con el listado de los artículos en stock, sino además otras características como el precio, proveedores, código de barras, imágenes, entre otros. De la misma manera, para representar la planilla de sueldos de los empleados de una empresa, por lo general necesitamos más información que la que usamos para modelar una primera versión del problema.

Afortunadamente, recordemos que los elementos de una lista no tienen por qué ser únicamente atómicos (es decir, números, strings o valores lógicos). En efecto, podemos diseñar estructuras y agregarlas sin problemas a cualquier lista que definamos. Así pues, intentemos ir un paso más allá y hagamos el inventario de la juguetería un poco más realista. Comenzaremos entonces por diseñar la estructura que va a representar a la definición de un nuevo tipo de datos específico para este problema: un registro de inventario.

Definiremos un *registro* como una estructura compuesta de un campo de tipo texto para almacenar el nombre del producto, y de un campo de tipo numérico para almacenar el valor de dicho producto. Así pues, diseñemos la estructura:

```
# registro:  producto(str) precio(int)
import estructura
estructura.crear("registro", "producto precio")
```

Más aún, podemos definir una *colección de registros* para almacenar toda la información que disponemos. A esto nos referiremos en este problema como *inventario*:

```
# inventario:  [registro]*
# inventario es una lista de registros de largo indeterminado
```

Es decir, un *inventario* está compuesto de:

1. Una lista vacía: `listaVacia`, o bien
2. Una lista que contiene un registro, encadenada al inventario: `crearLista(registro, inventario)`.

De la definición anterior, está claro que la forma más simple de *inventario* es la propia lista vacía: `listaVacia`. Si agregamos un registro, debemos entonces encadenar un nuevo eslabón de la lista a la lista vacía: `crearLista(registro("muneca", 2990), listaVacia)`. Notemos que el elemento que estamos agregando a la lista es una estructura de tipo *registro*. Así, para seguir agregando elementos al inventario, basta con crear un nuevo registro, y encadenarlo a la lista anterior:

```
crearLista(registro("robot", 5990), crearLista( registro("muneca", 2990), listaVacia))
```

Ahora, para hacer las cosas más interesantes, podemos implementar una función que calcule la suma total de los precios que están registrados en el inventario (es decir, calcular el valor total de los productos de la juguetería). En primer lugar, debemos definir el contrato, el propósito, ejemplos, y el encabezado de la función:

```
# suma: inventario -> int
# calcula la suma de todos los precios en unInventario
# suma(listaVacia) = 0
# suma(crearLista(("muneca", 2990), listaVacia)) = 2990
# suma(crearLista(("robot", 5990), \
#   (crearLista("muneca", 2990), listaVacia))) = 8980
def suma(unInventario):
        ...
```

Dado que la definición de un inventario es básicamente la misma que para las listas, nos podemos basar en la plantilla de listas para proponer la plantilla para los inventarios. Así:

```
# def suma(unInventario):
#       if  vacia(unInventario):
#               ...
#       else:
#               ...  cabeza(unInventario)
#               ...  suma(cola(unInventario)) ...
```

Siguiendo la receta de diseño, recordemos que la plantilla sólo refleja la definición de los datos para la entrada, no para sus componentes.

Finalmente, para definir el cuerpo de la función, debemos considerar cada una de las ramas en la condición de manera independiente. En primer lugar, si el inventario está vacío, naturalmente la suma total será 0. Si no, debemos tomar el elemento de la cabeza, acceder al campo `precio` del registro, y luego sumarlo al resultado que arroje la llamada recursiva de la función sobre la cola del inventario.

El cuerpo de la función resulta ser:

```
# suma: inventario -> int
# calcula la suma de todos los precios en unInventario
# suma(listaVacia) == 0
# suma(lista(registro("muneca", 2990), lista_vacia)) == 2990
# suma(lista(registro("robot", 5990), \
#    lista(registro("muneca", 2990), lista_vacia))) == 8980
def suma(unInventario):
    if vacia(unInventario):
        return 0
    else:
        item = cabeza(unInventario)
        return item.precio + suma(cola(unInventario))

# Tests
suma(lista_vacia) == 0
suma(lista(registro("muneca", 2990), listaVacia)) == 2990
suma(lista(registro("robot", 5990), \
   lista(registro("muneca", 2990), listaVacia))) == 8980
```

9.6 Modulo de listas

¿Como será implementar un módulo tan útíl como el de las listas? Utilizando estructuras de datos, la respuesta es simple:

Contenido del archivo lista.py

```
import estructura

# Diseno de la estructura
# lista : valor (any = cualquier tipo) siguiente (lista)
estructura.crear("lista", "valor siguiente")

# identificador para listas vacias
listaVacia = None

# crearLista: any lista -> lista
# devuelve una lista cuya cabeza es valor
# y la cola es resto
def crearLista(valor, resto):
        return lista(valor, resto)

# cabeza: lista -> any
# devuelve la cabeza de una lista (un valor)
def cabeza(lista):
        return lista.valor

# cola: lista -> lista
```

Contenido del archivo lista.py (cont)

```
22 # devuelve la cola de una lista (una lista)
23 def cola(lista):
24         return lista.siguiente
25
26 # vacia: lista -> bool
27 # devuelve True si la lista esta vacia
28 def vacia(lista):
29         return lista == listaVacia
30
31
32 # Tests
33
34 test_lista = lista(1, lista(2, lista(3, listaVacia)))
35
36 assert cabeza(test_lista) == 1
37 assert cabeza(cola(test_lista)) == 2
38 assert cabeza(cola(cola(test_lista))) == 3
39 assert cola(cola(test_lista)) == lista(3, listaVacia)
40
41 assert vacia(listaVacia)
42 assert not vacia(test_lista)
43 assert vacia(cola(cola(cola(test_lista))))
```

9.7 Otras definiciones de datos recursivas

Hasta ahora hemos visto solamente tipos de datos compuestos que requieren una única autorreferencia (como es el caso de las listas). Sin embargo, a menudo nos podemos topar con la necesidad de contar con definiciones más complejas, que requieran de más de una referencia recursiva. Luego, necesitaremos contar con un protocolo para poder definir estas nuevas estructuras, a partir de descripciones informales proporcionadas por el problema a resolver.

Veamos un ejemplo práctico. Típicamente, los médicos utilizan árboles genealógicos para investigar patrones de ocurrencia hereditarios en una familia. Por ejemplo, un árbol genealógico podría servir para ver el seguimiento en los colores de ojos en una familia. ¿Cómo podríamos representar esta estructura en el computador?

Una forma de mantener este árbol genealógico puede ser agregar un nodo cada vez que un niño nace. Desde este nodo, podemos dibujar conexiones al nodo del padre y al nodo de la madre, lo que nos indica que estos nodos están relacionados entre sí. Naturalmente, no dibujamos conexiones para los nodos que no conocemos sus ancestros o descendientes. Más aún, podemos agregar a cada nodo más información relacionada con la persona en cuestión: su nombre, color de ojos, color de pelo, etc.

El ejemplo de la Figura 9.1 nos muestra un árbol genealógico de ancestros de una familia. Esto es, se indica la relación entre los nodos desde los hijos hacia sus padres. Así, Andrés es el hijo de Carlos y Beatriz, tiene ojos pardos y nació en 1950. De manera similar, Gustavo nació en 1996, tiene ojos pardos y es hijo de Eva y Federico. Para representar a un nodo en el árbol combinamos en un único dato compuesto: el nombre, el año de nacimiento, el color de ojos, quién es el padre y quién es la

madre.

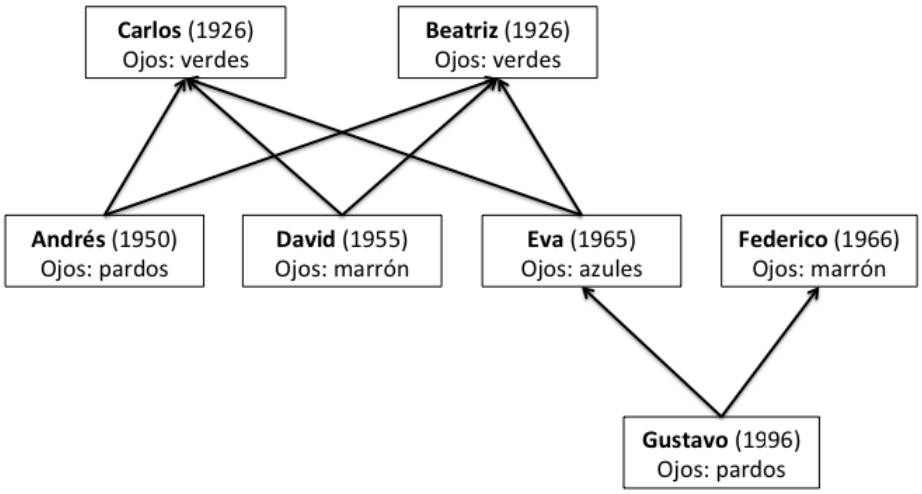


Figura 9.1: Un árbol genealógico de ancestros

Esto sugiere que debemos diseñar una nueva estructura:

```
# hijo:  padre(hijo) madre(hijo) nombre(str) nacimiento(int) ojos(str)
estructura.crear("hijo", "padre madre nombre nacimiento ojos")
```

En otras palabras, un *hijo* se construye a partir de cinco elementos, donde:

1. `padre` y `madre` son: (a) vacíos (`listaVacia`), o bien, (b) otro `hijo`; `nombre` y `ojos` son strings; `nacimiento` es un entero.

Esta definición es especial en dos puntos. Primero, es una definición de datos recursiva que involucra estructuras. Segundo, la definición de datos tiene una sola cláusula y menciona dos alternativas para el primer y segundo componente. Esto viola la receta de diseño vista en este capítulo para definiciones de datos recursivas.

Sin embargo, notemos que podemos resolver este problema definiendo simplemente una colección de nodos en un árbol genealógico (`nodoAG`) de la siguiente manera:

```
# nodoAG: padre(nodoAG) madre(nodoAG) nombre(str) nacimiento(int) ojos(str)
```

donde la estructura puede ser:

1. vacía (`nodoAGVacio`), o bien
2. otro `nodoAG`

Notemos que, a diferencia del caso anterior, un `nodoAG` no especifica explícitamente cuáles son los tipos de cada una de sus componentes, sino que es recursiva (un `nodoAG` se define en términos de sí mismo). Así, esta nueva definición sí satisface la convención que impusimos de tener al menos dos cláusulas: una no recursiva y una recursiva.

Veamos cómo podemos crear el árbol del ejemplo anterior. Primero, tenemos que crear cada uno de los nodos:

- Carlos: `nodoAG(nodoAGVacio, nodoAGVacio, "Carlos", 1926, "verdes")`
- Beatriz: `nodoAG(nodoAGVacio, nodoAGVacio, "Beatriz", 1926, "verdes")`
- Federico: `nodoAG(nodoAGVacio, nodoAGVacio, "Federico", 1966, "marron")`

Por otro lado, tenemos que definir los nodos más profundos en términos de otros nodos que hay que referenciar en el camino. Esto puede resultar bastante engorroso, por lo que lo más sencillo es introducir una definición de variable por cada nodo, y luego utilizar esta variable para referirnos a los distintos componentes del árbol.

Luego, la definición completa del árbol la podemos transcribir como sigue:

```
# Primera generacion:
carlos = nodoAG(nodoAGVacio, nodoAGVacio, "Carlos", 1926, "verdes")
beatriz = nodoAG(nodoAGVacio, nodoAGVacio, "Beatriz", 1926, "verdes")

# Segunda generacion:
andres = nodoAG(carlos, beatriz, "Andres", 1950, "pardos")
david = nodoAG(carlos, beatriz, "David", 1955, "marron")
eva = nodoAG(carlos, beatriz, "Eva", 1965, "azules")
federico = nodoAG(nodoAGVacio, nodoAGVacio, "Federico", 1966, "marron")

# Tercera generacion:
gustavo = nodoAG(federico, eva, "Gustavo", 1996, "pardos")
```

Ahora que ya contamos con una descripción de cómo podemos manipular árboles genealógicos, nos gustaría poder definir funciones sobre ellos. Consideremos de manera genérica una función del tipo:

```
# funcionAG:  nodoAG -> ???
def funcionAG(unAG):
```

Notemos que deberíamos ser capaces de definir la plantilla de la función, aun sin saber qué es lo que debería producir como resultado. Recordemos que, dado que la definición de la estructura para los `nodoArbol` tiene dos cláusulas, la plantilla debe ser entonces una condición con dos ramas, una para cada cláusula. La primera tiene que verificar el caso `nodoAGVacio` (notar que es necesario implementar la función `vacio`), mientras que la segunda tiene que ocuparse de las componentes de `nodoAG`:

```
# unaFuncion:  nodoAG -> ???
# def unaFuncion(unAG):
#       if vacio(nodoAG):
#       ...
#       else:
#       ...  unaFuncion(unAG.padre) ...
#       ...  unaFuncion(unAG.madre) ...
#       ...  unAG.nombre ...
#       ...  unAG.nacimiento ...
#       ...  unAG.ojos ...
```

Veamos ahora un ejemplo concreto. Diseñemos una función que determine si alguien en la familia tiene ojos azules. Como ya debería resultar natural, sigamos la receta de diseño:

```
# ancestroOjosAzules: nodoAG -> bool
# determina si nodoAG tiene algun ancestro con color de ojos azul
def ancestroOjosAzules(unAG):
        ...
```

Ahora tenemos que desarrollar algunos ejemplos. Consideremos el nodo de Carlos: no tiene ojos azules y como no tienen ningún ancestro (conocido) en la familia, el árbol representado por este nodo no tiene una persona con ojos azules. Luego, `ancestroOjosAzules(carlos)` se evalúa a `False`. Por otro lado, el árbol genealógico representado por Gustavo tiene una referencia al nodo de Eva, quien sí tiene ojos azules. Así, `ancestroOjosAzules(gustavo)` se evalúa a `True`.

La plantilla de la función es similar a la que vimos anteriormente, salvo que ahora nos referimos específicamente a la función `ancestroOjosAzules`. Como siempre, usaremos la plantilla para guiar el diseño del cuerpo de la función:

1. En primer lugar, si `unAG` está vacío, entonces naturalmente nadie puede tener los ojos azules en ese árbol (pues no hay nadie), y la evaluación da `False`.
2. Por otro lado, la segunda cláusula de la plantilla tiene varias expresiones, que debemos revisar una a una:
   (a) `ancestroOjosAzules(unAG.padre)` verifica si alguien en el árbol del padre del nodo tiene ojos azules;
   (b) `ancestroOjosAzules(unAG.madre)` verifica si alguien en el árbol de la madre del nodo tiene ojos azules;
   (c) `unAG.nombre` recupera el nombre del nodo;
   (d) `unAG.nacimiento` recupera el año de nacimiento del nodo;
   (e) `unAG.ojos` recupera el color de ojos del nodo;

Nuestra misión es ahora utilizar apropiadamente estos valores. Claramente, si el nodo tiene el string `"azules"` en el campo `ojos`, entonces el resultado de la función debe ser `True`. En caso contrario, la función debe producir `True` sólo si en el árbol del `padre` o de la `madre` alguien tiene los ojos azules. Este último caso nos indica que debemos verificar *recursivamente* sobre los componentes del árbol para determinar el color de ojos.

Finalmente, dada la discusión anterior, el cuerpo de la función resulta como sigue:

```
# ancestroOjosAzules: nodoAG -> bool
# determina si nodoArbol tiene algun ancestro con color de ojos azul
def ancestroOjosAzules(unAG):
        if vacio(unAG):
                return False
        else:
                if unAG.ojos == "azules":
                        return True
                if ancestroOjosAzules(unAG.padre):
                        return True
                if ancestroOjosAzules(unAG.madre):
```

```
                        return True
                # si se llega este punto, es porque
                # ningun ancestro tenia ojos azules
                return False
#Tests
assert not ancestroOjosAzules(carlos)
assert not ancestroOjosAzules(beatriz)
assert ancestroOjosAzules(eva)
assert not ancestroOjosAzules(david)
assert ancestroOjosAzules(gustavo)
```

9.8 Definiciones mutuamente recursivas

En el ejemplo de la sección anterior, construimos el árbol genealógico desde la perspectiva de los hijos. Sin embargo, es usual construir *árboles genealógicos descendentes*, es decir, indicar quién es descendiente de quién (en lugar de quién es ancestro de quién). La Figura 9.2 muestra el árbol con una perspectiva descendente.

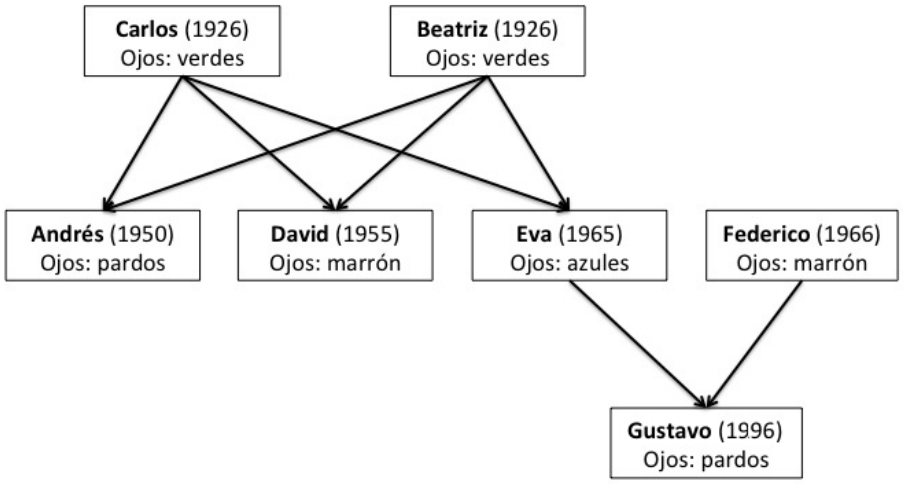


Figura 9.2: Un árbol genealógico de descendientes

No es difícil darse cuenta que para representar este tipo de árboles en el computador necesitamos una estructura de datos diferente al que usamos para el árbol genealógico de ancestros. Esta vez, los nodos van a incluir información sobre los hijos (que a priori no sabemos cuántos son) en lugar de los dos padres.

La estructura se define como sigue:

```
# padre: hijos(???) nombre(str) nacimiento(int) ojos(str)
```

Los últimos tres campos de la estructura contienen la misma información básica correspondiente a los nodos de tipo `hijo`. Sin embargo, el primer componente es particular, en cuanto no sabemos cuántos hijos hay (pueden ser 0 o más). Así, la opción natural es definir en este campo una *lista* de nodos de tipo *hijo*. Si una persona no tiene hijos, entonces la lista será `listaVacia`. Luego, debemos

definir la estructura que va a representar a esta lista de hijos.

Una `listaHijos` será:

1. Una lista vacía `listaVacia`, o bien,
2. Una lista de hijos: `crearLista(padre, listaHijos)`.

Notemos que ambas estructuras se refieren mutuamente: para definir a un padre necesitamos una lista de hijos, y para definir a una lista de hijos necesitamos un padre. Así, la única forma que estas dos definiciones hagan sentido es que se introduzcan *simultáneamente*:

```
import estructura

# padre: hijos(listaHijos) nombre(str) nacimiento(int) ojos(str)
estructura.crear("padre", "hijos nombre nacimiento ojos")

# listaHijos: lista(padre)
# listaHijos es una lista de padres de largo indeterminado
```

Traduzcamos ahora el árbol descendente que presentamos más arriba. Naturalmente, antes de poder crear una estructura de tipo `padre` debemos definir todos los nodos que van a representar a los hijos. Al igual que en la sección anterior, esta definición puede volverse rápidamente muy engorrosa, por lo que lo mejor es introducir variables para llamar a las distintas referencias.

Veamos la generación más joven: tenemos que Gustavo es hijo de Federico y de Eva. Luego, lo primero que debemos hacer es crear un nodo para Gustavo:

```
gustavo = padre(listaVacia, "Gustavo", 1996, "pardos")
```

Creemos ahora el nodo que representa a Federico: tiene como único hijo a Gustavo (que ya definimos previamente). Luego:

```
federico = padre(crearLista(gustavo, listaVacia), "Federico", 1966, "marron")
```

Notemos que en este caso utilizamos la función `crearLista` provista por el módulo `lista.py`, que precisamente permite crear una lista a partir de los valores que recibe como parámetro (en este caso, una estructura de tipo `listaHijos`).

Luego, la traducción completa del árbol es:

```
# Nietos:
gustavo = padre(listaVacia, "Gustavo", 1996, "pardos")
# Padres:
hijosFedericoEva = crearLista(gustavo, listaVacia)
federico = padre(hijosFedericoEva, "Federico", 1966, "marron")
eva = padre(hijosFedericoEva, "Eva", 1965, "azules")
david = padre(listaVacia, "David", 1955, "marron")
andres = padre(listaVacia, "Andres", 1950, "pardos")
# Abuelos:
hijosBeatrizCarlos =
crearLista(andres, crearLista(david, crearLista(eva, listaVacia)))
```

```
beatriz = padre(hijosBeatrizCarlos, "Beatriz", 1926, "verdes")
carlos = padre(hijosBeatrizCarlos, "Carlos", 1926, "verdes")
```

Veamos ahora el desarrollo de la función `descendienteOjosAzules`, que verifica dentro del árbol si hay algún descendiente que tenga los ojos de color azul. Naturalmente, recibe como parámetro una estructura de tipo `padre` y devuelve un valor de tipo booleano.

```
# descendienteOjosAzules:  padre -> bool
# determina si unPadre o cualquiera de sus
# descendientes tiene color de ojos azul
def descendienteOjosAzules(unPadre):
        ...
```

Formulemos directamente tres tests para probar nuestra función:

```
# Tests
assert not descendienteOjosAzules(gustavo)
assert descendienteOjosAzules(eva)
assert descendienteOjosAzules(beatriz)
```

Veamos cómo definir la plantilla. Dado que la estructura `padre` tiene cuatro componentes, entonces debemos desagregar cuatro expresiones para poner en la plantilla. Además, como la función no recibe más parámetros, los selectores de la plantilla serán precisamente las cuatro expresiones que se desprenden de las componentes de la estructura:

```
# def unaFuncion(unPadre):
# ...  unPadre.listaHijos ...
# ...  unPadre.nombre ...
# ...  unPadre.nacimiento ...
# ...  unPadre.ojos ...
```

Comencemos ahora a escribir el cuerpo de la función. De la plantilla, notamos que podemos recuperar el color de ojos del padre. Luego, basta con preguntar si esta información es igual a `"azules"` para determinar si el padre tiene los ojos azules:

```
def descendienteOjosAzules(unPadre):
    if unPadre.ojos == "azules":
        return True
    else:
#       ...  unPadre.hijos ...
#       ...  unPadre.nombre ...
#       ...  unPadre.nacimiento ...
```

Dado que ni el campo `nombre` ni el campo `nacimiento` nos aportan información relevante respecto al color de ojos de una persona, los descartamos. Así, nos queda finalmente un único selector: `listaHijos`, una estructura que extrae la lista de hijos asociada a un padre dado.

Si el color de ojos de `unPadre` no es `"azules"`, entonces debemos buscar en la `listaHijos` si existe algún descendiente con ojos azules. Siguiendo nuestra guía para diseñar funciones complejas, lo mejor es definir una *función auxiliar* que se encargue de esta tarea y podemos continuar con la resolución del problema. Recordemos que por cada función auxiliar que definamos también tenemos que seguir la receta de diseño:

```
# hijoOjosAzules:  listaHijos -> bool
# determina si alguna de las estructuras en listaHijos tiene
# ojos azules
def hijoOjosAzules(unaLista):
        ...
```

Usando, pues, la función `hijoOjosAzules` podemos completar entonces la definición de nuestra función principal `descendienteOjosAzules`:

```
def descendienteOjosAzules(unPadre):
    if unPadre.ojos == "azules":
        return True
    else:
        return hijoOjosAzules(unPadre.hijos)
```

Ocupémonos ahora de la función `hijoOjosAzules`. Como es natural, veamos primero algunos tests:

```
# Tests
assert not hijoOjosAzules(hijosFedericoEva) # [Gustavo]
assert hijoOjosAzules(hijosBeatrizCarlos) # [Andres, David, Eva]
```

En el ejemplo, Gustavo no tiene ojos azules y no tiene registrado ningún descendiente. Luego, la función debería arrojar el valor `False` para la lista formada por Gustavo. Por otro lado, Eva tiene ojos azules, por lo que el evaluar la función con la lista `[andres, david, eva]` debería arrojar `True`.

Dado que el parámetro que recibe la función es una lista, la plantilla a utilizar es la estándar:

```
# def unaFuncion(unaLista):
#       if vacia(unaLista) == True:
#               ...
#       else:
#               ...  cabeza(unaLista) ...
#               ...  unaFuncion(cola(unaLista)) ...
```

Consideremos los dos casos:

1. Si `unaLista` está vacía, entonces la respuesta es `False`;
2. Si no, deberíamos preguntar si la cabeza de la lista (de tipo `padre`) tiene ojos azules. Esto propone dos salidas:
   (a) Si efectivamente la cabeza tiene de la lista tiene ojos azules, retornamos `True`;
   (b) Si no, preguntamos recursivamente sobre la cola de la lista.

Al respecto, recordemos que la función `descendienteOjosAzules` precisamente verifica si un elemento de tipo `padre` tiene o no los ojos azules. Notemos que la particularidad de esta definición es que ambas funciones son *mutuamente recursivas*, pues las definiciones de estructuras también lo son.

Finalmente, las definiciones de ambas funciones resultan ser:

```
# descendienteOjosAzules:  padre -> bool
# determina si unPadre o cualquiera de sus
# descendientes tiene color de ojos azul
```

```
4  def descendienteOjosAzules(unPadre):
5      if unPadre.ojos == "azules":
6          return True
7      else:
8          return hijoOjosAzules(unPadre.hijos)
9
10 # hijoOjosAzules:  listaHijos -> bool
11 # determina si alguna de las estructuras
12 # en listaHijos tiene ojos azules
13 def hijoOjosAzules(unaLista):
14     if vacia(unaLista):
15         return False
16     else:
17         if descendienteOjosAzules(cabeza(unaLista)):
18             return True
19         else:
20             return hijoOjosAzules(cola(unaLista))
```

9.9 Árboles binarios

En computación, no es muy común trabajar con una estructura del tipo `nodoAG` como la que que hemos visto de ejemplo en las Sección 9.7. No obstante, una forma particular de árbol que si es muy utilizada es conocida como *árbol binario*. En particular, discutiremos sobre un tipo de árboles denominado *árboles de búsqueda binaria*.

Suponga que se debe implementar una estructura que permita almacenar los datos de personas. En este contexto, un árbol binario es similar a un árbol genealógico, pero ahora utilizaremos la siguiente estructura para sus nodos:

```
1  # nodo: rut(int) nombre(str) izq(nodo) der(nodo)
2  estructura .crear("nodo", "rut nombre izq der")
3
4  # identificador para nodos vacios
5  nodoVacio = None
```

La definición correspondiente de un nodo de árbol binario es:

1. vacío (`nodoVacio`), o bien
2. otro `nodo`, donde `rut` es un número entero (no consideraremos el digito verificador), `nombre` es un string, y `izq` y `der` son nodos.

Nos va a resultar útil contar con una función que nos permita crear nuevos nodos y otra función que nos diga si un árbol binario es vacío:

```
1  # crearNodo: int str nodo nodo -> nodo
2  # devuelve un nodo ABB con datos rut, nombre
3  # y los nodos izq y der, respectivamente
4  def crearNodo(rut, nombre, izq, der):
5      return nodo(rut, nombre, izq, der)
```

```
6
7  # vacio: nodo -> bool
8  # devuelve True si el ABB esta vacio
9  def vacio(nodo):
10     return nodo == nodoVacio
```

Veamos como crear dos árboles binarios con la definición de nodo propuesta:

```
1  unArbol = crearNodo(15, "Juan", nodoVacio, crearNodo(24, "Ivan",
   nodoVacio, nodoVacio))
2  otroArbol = crearNodo(15, "Juan", crearNodo(87, "Hector",
   nodoVacio, nodoVacio), nodoVacio)
```

Ahora introduciremos un tipo especial de árbol binario. Un árbol de búsqueda binaria (ABB) es un árbol binario que cumple con una característica de orden con respecto a uno de los valores almacenados (en nuestro ejemplo, el RUT): para todo nodo del árbol se cumple que su valor asociado es mayor que los valores de todos los nodos del lado izquierdo (árbol binario correspondiente a `izq`), y es menor que los valores de todos los nodos del árbol derecho (árbol binario correspondiente a `der`). De acuerdo a esta definición, el árbol `unArbol` sería un ABB, pero el árbol `otroArbol` no lo es. Formalmente, un ABB para nuestro ejemplo se define como:

1. vacío (`nodoVacio`), o bien
2. otro `nodo`, donde se cumple que:
   (a) `izq` y `der` son ABB
   (b) todos los números en `izq` son menores que `rut`, y
   (c) todos los números en `der` son mayores que `rut`.

Note que esta es una definición de datos recursivos válida, ya que contiene al menos dos cláusulas: una no recursiva y una recursiva. Las últimas dos condiciones de la cláusula recursiva son distintas a lo que habíamos visto previamente, ya que agregan restricciones sobre la construcción de un ABB: se debe garantizar que al inspeccionar el árbol `izq` (`der`) todos los valores guardados son menores (mayores) que `rut`.

Para ejemplificar el uso de un ABB, diseñaremos una función que nos permita encontrar en un ABB el nombre asociado a un RUT específico. La plantilla de la función sería la siguiente:

```
1  # encontrarNombre:  ABB int -> str
2  # busca el nodo de un ABB cuyo rut es unRUT
3  # y retorna el texto asociado al nombre en dicho nodo
4  # ejemplo:
5  # si ABB = crearNodo(15, "Juan", nodoVacio, crearNodo(24, "Ivan",
   nodoVacio, nodoVacio))
6  # entonces encontrarNombre(ABB, 24) devuelve "Ivan"
7  # def encontrarNombre(unNodo, unRUT):
8  #       if vacio(unNodo):
9  #                 ...
10 #       else:
11 #                 ...  encontrarNombre(unNodo.izq) ...
12 #                 ...  encontrarNombre(unNodo.der) ...
13 #                 ...  unNodo.rut ...
14 #                 ...  unNodo.nombre ...
```

Para el caso base, adoptaremos la convención que si el ABB es vacío no hay un nombre asociado al RUT buscado, y la función retornará "" (texto vacío). Para el caso recursivo, vamos a requerir de una expresión condicional: verificamos si el RUT del nodo corresponde al RUT buscado. Si esto es cierto, la función retorna el nombre guardado en el nodo. En caso contrario, la búsqueda en el ABB debe proseguir en forma recursiva, pero la pregunta es, ¿se debe seguir buscando hacia el lado izquierdo o hacia el lado derecho? En este punto es en donde sacamos ventaja que la estructura es un ABB: si el RUT buscado es *menor* que el RUT del nodo, necesariamente debe estar hacia el lado izquierdo del ABB (sino, violaría las restricciones impuestas a un ABB); en caso contrario, si el RUT buscado es *mayor* que el RUT del nodo, necesariamente debe estar hacia el lado derecho del ABB.

Una vez que hemos entendido bien el procedimiento de búsqueda, procedemos a implementarlo:

```
# encontrarNombre:  ABB int -> str
# busca el nodo de un ABB cuyo rut es unRUT
# y retorna el texto asociado al nombre en dicho nodo
# ejemplo:
# si ABB = crearNodo(15, "Juan", nodoVacio, crearNodo(24, "Ivan",
nodoVacio, nodoVacio))
# entonces encontrarNombre(ABB, 24) devuelve "Ivan"
def encontrarNombre(unNodo, unRUT):
    if vacio(unNodo):
        return ""
    else:
        if unRUT == unNodo.rut:
            return unNodo.nombre
        elif unRUT < unNodo.rut:
            return encontrarNombre(unNodo.izq, unRUT)
        else: #unRUT > unNodo.rut
            return encontrarNombre(unNodo.der, unRUT)

# Tests
ABB = crearNodo(15, "Juan", nodoVacio, crearNodo(24, "Ivan",
nodoVacio, nodoVacio))
assert encontrarNombre(ABB, 24) == "Ivan"
assert encontrarNombre(ABB, 50) == ""
```

Capítulo 10

Abstracción Funcional[1]

10.1 Similitudes en definiciones

Muchas de las definiciones que creamos, ya sea de datos o de funciones, son parecidas entre sí. Por ejemplo, la definición de una lista de strings se diferencia de una lista de número solo en dos puntos: el nombre de la clase de los datos (es decir, lista-de-strings y lista-de-números) y las palabras "string" y "número". De igual manera, una función que busca un string específico en una lista de strings es casi indistinguible de una que busque un número específico en una lista de números.

Las repeticiones son la causa de muchos errores de programa. Por lo tanto, los buenos programadores tratan de evitar las repeticiones lo más posible. Cuando desarrollamos un conjunto de funciones, especialmente funciones derivadas de una misma plantilla, pronto aprendemos a encontrar similitudes. Una vez desarrolladas, es tiempo de revisarlas para eliminar las repeticiones lo más posible. Puesto de otra manera, un conjunto de funciones es como un ensayo o una novela u otro tipo de pieza escrita: el primer borrador es sólo un borrador. Es un sufrimiento para otros el tener que leerlos. Dado que las funciones son leídas por muchas otras personas y porque las funciones reales son modificadas después de leerse, debemos aprender a "editar" funciones.

La eliminación de repeticiones es el paso más importante en el proceso de edición de un programa. En esta sección discutiremos las similitudes en la definición de una función y en la definición de datos, y cómo las podemos evitar.

Similitudes entre funciones

El uso de nuestra receta de diseño determina por completo la plantilla de una función, o la organización básica, desde la definición de los datos para la entrada. De hecho, la plantilla es un modo alternativo de expresar lo que sabemos de nuestros datos de entrada. No es de extrañar que las funciones que consumen el mismo tipo de datos se vean parecidas:

[1]Parte de este capítulo fue traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

```
# hayPelotas :  lista(str) -> bool
# Determina si lista contiene
# el string pelota
# ejemplo hayPelotas(crearLista('pelota',
listaVacia)) devuelve True
def hayPelotas(unaLista):
    if vacia(unaLista):
       return False
    else:
       if cabeza(unaLista) == "pelota":
         return True
       else:
         return hayPelotas(cola(unaLista))
```

```
# hayAutos :  lista(str) -> bool
# Determina si lista contiene
# el string auto
# ejemplo hayAutos(crearLista('auto',
listaVacia)) devuelve True
def hayAutos(unaLista):
    if vacia(unaLista):
       return False
    else:
       if cabeza(unaLista) == "auto":
         return True
       else:
         return hayAutos(cola(unaLista))
```

Veamos las dos funciones anteriores: ambas consumen una lista de strings (nombres de juguetes) y buscan un juguete en particular. La función de la izquierda busca una pelota y la de la derecha busca un auto en una lista de strings (`unaLista`). Las dos funciones son casi indistinguibles. Cada una consume una lista de strings; cada cuerpo de la funcion consiste en una expresión condicional con dos cláusulas. Cada una produce `False` si la entrada es vacía; cada una consiste en una segunda expresión condicional anidada para determinar si el primer item es el que se busca. La única diferencia es el string que se usa en la comparación de la expresión condicional anidada: `hayPelotas` usa `"pelota"` y `hayAutos` usa `"auto"`.

Los buenos programadores son demasiado perezosos para definir muchas funciones que están estrechamente relacionadas. En lugar de eso, definen una sola función que puede buscar tanto `"pelota"` como `"auto"` en una lista de juguetes. Esa función más general debe consumir un dato adicional, el string que estamos buscando, pero por lo demás es igual que las dos funciones originales.

```
1  # contiene : str lista(str) -> bool
2  # Determina si lista contiene el string s
3  # ejemplo contiene('auto', crearLista('auto', listaVacia)) retorna True
4  def contiene(s, unaLista):
5    if vacia(unaLista):
6      return False
7    else:
8      if cabeza(unaLista) == s:
9        return True
10     else:
11       return contiene(s, cola(unaLista))
```

Ahora podemos buscar `"pelota"` aplicando la funcion `contiene` a `"pelota"` y a una lista de strings. Pero `contiene` también funciona para cualquier otro string. Definir una sola versión de la función nos permite solucionar muchos problemas relacionados de una sola vez.

El proceso de combinar dos funciones relacionadas en una sola función se denomina *abstracción funcional*. Definir versiones abstractas de funciones es sumamente beneficioso. El primero de estos beneficios es que la función abstracta puede realizar muchas tareas diferentes. En nuestro ejemplo, `contiene` puede buscar muchos strings deferentes en vez de solamente uno particular.

```
# inferiores:  lista(num) num -> lista(num)
# Construye una lista de aquellos numeros
# de unaLista que sean inferiores a n
# ejemplo:  inferiores(crearLista(1,
# crearLista(2, listaVacia)), 2)
# devuelve (1, listaVacia)
def inferiores(unaLista, n):
    if vacia(unaLista):
       return listaVacia
    else:
       if cabeza(unaLista) < n:
         return crearLista(cabeza(unaLista),
inferiores(cola(unaLista), n))
       else:
         return inferiores(cola(unaLista), n)
```

```
# superiores:  lista(num) num -> lista(num)
# Construye una lista de aquellos numeros
# de unaLista que sean superiores a n
# ejemplo:  superiores(crearLista(2,
# crearLista(4, listaVacia)), 2)
# devuelve (4, listaVacia)
def superiores(unaLista, n):
    if vacia(unaLista):
       return listaVacia
    else:
       if cabeza(unaLista) > n:
         return crearLista(cabeza(unaLista),
superiores(cola(unaLista), n))
       else:
         return superiores(cola(unaLista), n)
```

En el caso de `hayPelotas` y `hayAutos` la abstracción es sencilla. Sin embargo hay casos más interesantes, como son las dos funciones anteriores. La función de la izquierda consume una lista de números y un número, y produce una lista de todos aquellos números de la lista que son inferiores a ese número; la función de la derecha produce una lista con todos aquellos números que están por encima de ese número.

La diferencia entre ambas funciones es el operador de la comparación. La de la izquierda ocupa $<$ y la de la derecha $>$. Siguiendo el primer ejemplo, abstraemos las dos funciones usando un parámetro adicional que indica el operador relacional en `inferiores` y `superiores`:

```
1 def filtro(operador, unaLista, n):
2   if vacia(unaLista):
3     return listaVacia
4   else:
5     if operador(cabeza(unaLista), n) :
6       return crearLista(cabeza(unaLista), filtro(operador,
  cola(unaLista), n))
7     else:
8       return filtro(operador, cola(unaLista), n)
```

En Python no es posible pasar operadores como argumentos de una función de manera directa. Luego, para poder crear operadores específicos, y posteriormente darlos como argumento a la función `filtro`, crearemos funciones que los definan. Para nuestro ejemplo podemos crear la función `menorQue`, que toma dos argumentos numéricos, y devuelve `True` si el primer argumento es menor que el segundo y `False` en el caso contrario:

```
1 # menorQue: num num -> bool
2 # devuelve True si el primer argumento es menor que el segundo
3 # y False en el caso contrario
4 # Ejemplo: menorQue(4,2) -> False
5 def menorQue(x,y):
6   return x < y
```

Luego, para aplicar esta nueva función, debemos proporcionar tres argumentos: un operador relacional R que compara dos números, una lista L de números, y un número N. La función entonces extrae todos aquellos elementos i en L para las cuales $R(i, N)$ evalúa `True`. Por ahora omitiremos cómo

escribir contratos para funciones como `filtro`, puesto que no sabemos aún como hacerlo. Discutiremos este problema en las secciones que siguen.

Veamos cómo trabaja `filtro` en un ejemplo. Claramente, si la lista que se entrega como parámetro es listaVacia, el resultado también será listaVacia, sin importar cuáles ean los otro argumentos:

```
>>> filtro(menorQue, listaVacia, 5)
listaVacia
```

Ahora veamos un ejemplo un poco más complejo:

```
>>> filtro(menorQue, crearLista(4, listaVacia), 5)
```

El resultado debería ser (4, listaVacia) porque el único elemento de esta lista es 4 y menorQue(4, 5) se evalúa como `True`. El primer paso de esta evaluación está basada en la regla de aplicación:

```
filtro(menorQue, crearLista(4, listaVacia), 5)
=
if vacia(crearLista(4, listaVacia)):
  return listaVacia
else:
  if menorQue(cabeza(crearLista(4, listaVacia)), 5) :
    return crearLista(cabeza(crearLista(4, listaVacia)), \
      filtro(menorQue, cola(crearLista(4, listaVacia)), 5))
  else:
    return filtro(menorQue, cola(crearLista(4, listaVacia)), 5)
```

Esto significa que es el cuerpo de `filtro` con todas las ocurrencias de `operador` reemplazadas por `menorQue`, `n` reemplazadas por 5, y `unaLista` reemplazada por `crearLista(4, listaVacia)`. El resto de la evaluación es directa:

```
if vacia(crearLista(4, listaVacia)):
  return listaVacia
else:
  if menorQue(cabeza(crearLista(4, listaVacia)), 5) :
    return crearLista(cabeza(crearLista(4, listaVacia)), \
      filtro(menorQue, cola(crearLista(4, listaVacia)), 5))
  else:
    return filtro(menorQue, cola(crearLista(4, listaVacia)), 5)
=
if menorQue(cabeza(crearLista(4, listaVacia)), 5) :
  return crearLista(cabeza(crearLista(4, listaVacia)), \
    filtro(menorQue, cola(crearLista(4, listaVacia)), 5))
else:
  return filtro(menorQue, cola(crearLista(4, listaVacia)), 5)
=
if menorQue(4, 5) :
  return crearLista(cabeza(crearLista(4, listaVacia)), \
    filtro(menorQue, cola(crearLista(4, listaVacia)), 5))
else:
  return filtro(menorQue, cola(crearLista(4, listaVacia)), 5)
=
```

```
if True :
  return crearLista(cabeza(crearLista(4, listaVacia)), \
    filtro(menorQue, cola(crearLista(4, listaVacia)), 5))
else:
  return filtro(menorQue, cola(crearLista(4, listaVacia)), 5)
= crearLista(4, filtro(menorQue, cola(crearLista(4, listaVacia)), 5))
= crearLista(4, filtro(menorQue, listaVacia, 5))
= crearLista(4, listaVacia)
= (4, listaVacia)
```

Nuestro ejemplo final es la aplicación de `filtro` a una lista de dos elementos:

```
filtro(menorQue, crearLista(6, crearLista(4, listaVacia)), 5)
= filtro(menorQue, crearLista(4, listaVacia), 5)
= crearLista(4, filtro(menorQue, listaVacia, 5))
= crearLista(4, listaVacia)
```

El único paso nuevo es el primero. Este dice que `filtro` determina que el primer elemento en la lista no es menor que el número entregado como parámetro de comparación, y que por lo tanto no debe ser agregado al resultado de la recursión.

Repeticiones dentro de una función

La repetición de código no sólo se da entre funciones relacionadas, sino que también puede darse dentro de una misma función. Observemos la siguiente función (notar que el código es un *pseudo–Python*, no funciona si Ud. lo transcribe):

```
# mayorLargo: lista(any) lista(any) -> num
# Devuelve el largo de la lista mas larga, si ambas son vacias
# devuleve -1
# Ejemplo: mayorLargo (crearLista(5, listaVacia), listaVacia) -> 1
def mayorLargo(x, y):
  if vacia(x) and vacia(y):
    return -1
  elif len lista(x) > len lista(y):
    return len lista(x)
  else:
    return len lista(y)
```

En este ejemplo podemos observar que el largo de ambas listas se calcula dos veces. La primera vez se hace para determinar cuál de las dos listas tiene el largo mayor, y la segunda para obtener el valor que retornará la función. Esto significa que existen repeticiones en nuestro código, lo cual no solo afecta a la definición de nuestra función en términos sintácticos, sino que además hace que ésta sea menos eficiente puesto que tiene que evaluar la misma expresión más de una vez. Esto lo podemos mejorar utilizando una *función auxiliar*, que calcule el máximo entre dos valores, y luego utilizarla en nuestra función original para eliminar las repeticiones observadas:

```
# maximo: num num -> num
# Devuelve el maximo entre x e y
# ejemplo: maximo(4, 2) -> 4
def maximo(x, y):
```

```
5   if x > y:
6     return x
7   else:
8     return y
9
10  # listaMasLarga: lista lista -> numero
11  # Devuelve el largo de la lista mas larga, si ambas son vacias
12  # devuelve -1
13  # Ejemplo: listaMasLarga (crearLista(5, listaVacia), listaVacia) -> 1
14  def listaMasLarga(x, y):
15    if vacia(x) and vacia(y):
16      return -1
17    else:
18      return maximo(len lista(x), len lista(y))
```

10.2 Similitudes en definición de datos

Veamos las dos siguientes definiciones de estructuras de datos recursivas:

Una lista-de-números puede ser:	Una lista-de-*RI*s puede ser:
• listaVacia • crearLista(n, l): donde n es un número y l es una lista de números	• listaVacia • crearLista(n, l): donde n es un RI y l es una lista de RI

Ambas definen un tipo de listas. La de la izquierda es la definición de un dato que representa una lista de números; la de la derecha describe una lista de registros de inventario (RI), que representaremos con una estructura de dato compuesto. La definición de la estructura está dada como sigue:

```
1  import estructura
2  estructura.crear("ri", "nombre precio")
```

Dada la similitud entre la definición de ambas estructuras de datos recursivas, las funciones que consumen elementos de estas clases también son similares. Miremos el siguiente ejemplo. La función de la izquierda es `inferiores`, la cual filtra números de una lista de números. La función de la derecha es `inf-ri`, que extrae todos aquellos registros de inventario de una lista cuyo precio esté por debajo de un cierto número. Excepto por el nombre de la función, la cual es arbitraria, ambas definiciones difieren solo en un punto: el operador relacional.

```
# inferiores :  lista(num) num -> lista(num)
# Construye una lista de aquellos numeros
# de unaLista que sean inferiores a n
# ejemplo:  inferiores(crearLista(1,
# crearLista(2, listaVacia)), 2)
# devuelve (1, listaVacia)
def inferiores(unaLista, n):
    if vacia(unaLista):
       return listaVacia
    else:
       if cabeza(unaLista) < n:
         return crearLista(cabeza(unaLista),
inferiores(cola(unaLista), n))
       else:
         return inferiores(cola(unaLista), n)
```

```
# inf-ri :  lista(RI) num -> lista(RI)
# Construye una lista de aquellos registros
# en unaLista con precio inferior a n
# ej.:  inf-ri(crearLista(ri("ri1", 1),
# crearLista(ri("ri2", 2), listaVacia)), 2)
# devuelve (ri("ri1", 1), listaVacia)
def inf-ri(unaLista, n):
    if vacia(unaLista):
       return listaVacia
    else:
       if cabeza(unaLista) <ri n:
         return crearLista(cabeza(unaLista),
inf-ri(cola(unaLista), n))
       else:
         return inf-ri(cola(unaLista), n)
```

Si abstraemos las dos funciones, obviamente obtenemos la función `filtro`. Sin embargo, podemos escribir `inf-ri` en términos de `filtro`:

```
def inf-ri(unaLista, n):
   return filtro(<ri, unaLista, n)
```

No nos debería sorprender encontar otros usos para `filtro` puesto que ya argumentamos que la abstracción funcional fomenta el reuso de funciones para distintos propósitos. Acá podemos ver que `filtro` no solamente filtra lista de números, sino que cosas arbitrarias. Esto se cumple siempre y cuando podamos definir una función que compare estas cosas arbitrarias con números.

En efecto, todo lo que necesitamos es una función que pueda comparar elementos de una lista con elementos que pasamos a `filtro` como el segundo argumento. Acá presentamos una función que extrae todos los elementos con el mismo nombre de una lista de registros de inventario:

```
# encontrar : lista(RI) str -> bool
# determina si es que unaListaRI contiene un registro de s
# ejemplo: encontrar(crearLista(ri("auto", 100), listaVacia), "auto")
# devuelve True
def encontrar(unaListaRI, s):
  return not vacia(filtro(igual-ri?, unaListaRI, s))

# igual-ri? : RI str -> bool
# comparar el nombre de un ri y p
# ejemplo: igual-ri?(ri("auto", 100), "auto") devuelve True
def igual-ri?(ri, p):
  return ri.nombre == p
```

10.3 Formalizar la abstracción a partir de ejemplos

En los ejemplos vistos anteriormente, partimos de dos definiciones concretas de funciones, las comparamos, marcamos las diferencias, y realizamos la abstracción. Ahora, formularemos una receta para realizar todos estos pasos.

10.3.1 Comparación

Cuando encontramos dos definiciones de funciones que son casi idénticas, salvo algunas pocas diferencias en los nombres, se comparan las funciones y se marcan las diferencias, encerrándolas en una caja. Si las cajas sólo contienen valores, se puede hacer la abstracción. Veamos un ejemplo:

```
# convertirCF : lista(num) -> lista(num)
# Convierte los valores en unaLista
# de grados C a grados F
# ejemplo:  convertirCF(crearLista(0,
# listaVacia)) devuelve como resultado
# (32, listaVacia)
def convertirCF(unaLista):
    if vacia(unaLista):
       return listaVacia
    else:
       return
crearLista(CaF(cabeza(unaLista)),
convertirCF(cola(unaLista)))
```

```
# nombres :  lista(RI) -> lista(str)
# Produce una lista de strings con la lista
# de registros en unaLista
# ejemplo:  nombres(crearLista(ri("ri1", 1),
# listaVacia)) devuelve como resultado
# ("ri1 1"), listaVacia)
def nombres(unaLista):
    if vacia(unaLista):
       return listaVacia
    else:
       return
crearLista(RIaString(cabeza(unaLista)),
nombres(cola(unaLista)))
```

Ambas funciones aplican una función a cada valor de la lista. Se distinguen en un solo punto: la función que aplican (`CaF` y `RIaString`, respectivamente), que están marcadas en cajas. Ambos cajas contienen valores funcionales, por lo que se puede realizar la abstracción.

10.3.2 Abstracción

A continuación, se reemplazan los contenidos de los pares de cajas correspondiente con nuevos identificadores, y se agregan dichos nombres a los parámetros. Por ejemplo, si hay tres pares de cajas se necesitan tres identificadores nuevos. Ambas definiciones deben ser ahora idénticas, excepto por el nombre de la función. Para obtener la abstracción, se reemplazan sistemáticamente los nombres de función por uno nuevo.

Para nuestro ejemplo, se obtiene el siguiente par de funciones:

```
def convertirCF(f, unaLista):
    if vacia(unaLista):
       return listaVacia
    else:
       return
crearLista(f(cabeza(unaLista)),
convertirCF(f, cola(unaLista)))
```

```
def nombres(f, unaLista):
    if vacia(unaLista):
       return listaVacia
    else:
       return
crearLista(f(cabeza(unaLista)),
nombres(f, cola(unaLista)))
```

Note que se reemplazaron los nombres en la cajas con `f`, y se agregó `f` como parámetro de ambas funciones. Ahora se reemplazan `convertirCF` y `nombres` con un nuevo nombre de función, obteniéndose la función abstracta `mapa`:

```
1 def mapa(f, unaLista):
2   if vacia(unaLista):
3     return listaVacia
4   else:
5     return crearLista(f(cabeza(unaLista)), mapa(f, cola(unaLista)))
```

Usamos el nombre `mapa` para la función resultante en nuestro ejemplo, dado que es el nombre tradicional en los lenguajes de programación que tiene esta función específica.

10.3.3 Test

Ahora debemos validar que la nueva función es una abstracción correcta de la funciones originales. La definición misma de abstracción sugiere que se puede validar definiendo las funciones originales en términos de la función abstracta, probando las nuevas versiones con los ejemplos de test originales. En la mayoría de los casos, esto se puede realizar en forma directa. Supongamos que la función abstracta se llama `f-abstracta`, y suponga que una de las funciones originales se llama `f-original` que recibe un argumento, llamémosle `valor`. Si `f-original` difiere de las otras funciones originales en el uso de un valor, entonces se define la siguiente función:

```
1 def f-desde-abstracta(x):
2   return f-abstracta(valor, x)
```

Para cada valor (correcto) `v`, `f-desde-abstracta(v)` debiera producir el mismo resultado que `f-original(v)`.

Para nuestro ejemplo, las nuevas definiciones serían las siguientes:

```
def convertirCF-desde-mapa(unaLista):
    return mapa(CaF, unaLista)
```

```
def nombres-desde-mapa(unaLista):
    return mapa(RIaString, unaLista)
```

Para asegurarse que estas dos definiciones son equivalentes a las originales, y de paso mostrar que `mapa` es una abstracción correcta, se aplican estas dos funciones a los ejemplos especificados en los contratos originales de `convertirCF` y `nombres`.

10.3.4 Contrato

Para hacer la abstracción útil, debemos formular un contrato adecuado. Si los valores en las cajas de la abstracción (paso dos de esta receta) son funciones, como en el ejemplo, el contrato requiere definirlas, para lo cual utilizaremos tipos de datos con flechas. Adicionalmente, para obtener un contrato flexible, debemos definir y usar definiciones de datos paramétricas y formular un tipo paramétrico.

Veamos como hacer el contrato para la función `mapa` de nuestro ejemplo. Por una parte, si vemos a `mapa` como una abstracción de `convertirCF`, el contrato podría definirse como:

```
# mapa :  (num -> num) lista(num) -> lista(num)
```

Por el contrario, si vemos a `mapa` como una abstracción de `nombres`, el contrato quedaría como sigue:

```
# mapa :  (RI -> str) lista(RI) -> lista(str)
```

El primer contrato sería inútil en el segundo caso, y viceversa. Para acomodar ambos casos, es necesario entender qué es lo que hace `mapa` y luego definir el contrato. Mirando a la definición de `mapa`, podemos ver que aplica su primer argumento (una función) a cada elemento de su segundo argumento (una lista). Esto implica que la función debe "consumir" el tipo de dato que la lista contenga. Por ejemplo, sabemos que `f` tiene el contrato:

```
# f :  X -> ???
```

si unaLista contiene datos de tipo X. Más aún, mapa crea una lista con el resultado de aplicar f a cada elemento. Por lo tanto, si f produce datos de tipo Y entonces mapa produce una lista de datos de tipo Y. Al agregar todo esto el contrato queda como sigue:

```
# mapa :  (X -> Y) lista(X) -> lista(Y)
```

Este contrato indica que mapa produce una lista de datos de tipo Y a partir de una lista de datos de tipo X y una función de X a Y, independiente de lo que signifique X e Y.

Una vez que tenemos la abstracción de dos o más funciones, debemos verificar si hay otros usos para la función abstracta. En muchos casos, una función abstracta es útil en una amplia gama de contextos que van más alla de lo que uno inicialmente había anticipado. Esto hace que las funciones sean mucho más fáciles de leer, entender y mantener. Por ejemplo, ahora podemos usar la función mapa cada vez que necesitemos una función que produzca una lista nueva a partir del procesamiento de los elementos de una lista ya existente.

10.3.5 Formulando contratos generales

Para aumentar la utilidad de una función abstracta, se debe formular un contrato que describa su aplicabilidad en los términos más generales posible. Para esto, se requiere realizar una abstracción de los contratos, para lo cual se debe seguir la misma receta usada para abstraer funciones. Se comparan y contrastan los contratos originales, luego se reemplazan las diferencias con variables. Realizar este proceso es complejo y requiere práctica.

Veamos un ejemplo con los contratos de las funciones convertirCF y nombres:

```
# convertirCF : lista(num) -> lista(num)
# nombres :  lista(RI) -> lista(str)
```

Al comparar los contratos, se observa que difieren en dos lugares. A la izquierda de `->` se tiene `lista(num)` y `lista(RI)`, a la derecha se tiene codigolista(num) y `lista(str)`. Considere el segundo paso de la receta de abstracción. Los contratos quedan como:

```
# mapa :  (num -> num) lista(num) -> lista(num)
# mapa :  (RI -> str) lista(RI) -> lista(str)
```

Estos contratos sugieren un patrón: el primer argumento, una función, consume los valores en el segundo argumento, una lista; adicionalmente, los resultados producidos en este proceso conforman la salida, otra lista. Fijándonos en el segundo contrato, si se reemplaza `RI` y `str` con variables, se obtiene un contrato abstracto, que de hecho es el contracto formulado para la función mapa:

```
# mapa :  (X -> Y) lista(X) -> lista(Y)
```

Es fácil comprobar que, reemplazando `X` con `num` e `Y` con `num`, se obtiene el contrato para la función convertirCF. Veamos otro ejemplo con los contratos de las funciones inferiores e inf-ri:

```
# inferiores :  lista(num) num -> lista(num)
# inf-ri :  lista(RI) num -> lista(RI)
```

Los contratos se diferencian en dos partes: la lista consumida y la lista producida. Como ya hemos visto, las funciones de la segunda etapa del proceso de abstracción reciben un argumento adicional:

```
# filtro :  (num num -> bool) lista(num) num -> lista(num)
# filtro :  (num RI -> bool) lista(RI) num -> lista(RI)
```

El argumento añadido es una función, que en el primer caso recibe un `num` y en el segundo un `RI`. Comparando ambos contratos, se observa que `num` y `RI` ocupan la misma posición en el contrato, y podemos entonces reemplazarlos por una variable. Al hacer esto, el contrato, que corresponde al de la función `filtro` queda como:

```
# filtro :  (num X -> bool) lista(X) num -> lista(X)
# filtro :  (num X -> bool) lista(X) num -> lista(X)
```

Observando los nuevos contratos, vemos que ahora es posible reemplazar `num` por otra variable `Y`, con lo que obtenemos el contrato final:

```
# filtro :  (Y X -> bool) lista(X) Y -> lista(X)
```

El resultado del primer argumento debe ser `bool` (no puede ser reemplazado por una variable), dado que es utilizado en una condición. Por lo tanto, éste es el contrato más abstracto posible para la función `filtro`.

Resumiendo, para encontrar contratos generales se requiere comparar los contratos de los ejemplos que tengamos para crear abstracciones. Reemplazando valores distitnos en posiciones correspondientes por variables, una a la vez, se logra hacer un contrato más genérico en forma gradual. Para validar que la generalización del contrato es correcta, se debe verificar que el contrato describe correctamente las instancias específicas de las funciones originales.

10.4 Otro ejemplo de función abstracta

Otros problemas relacionados con listas que se pueden abstraer en una única función son los siguientes:

- Sumar/multiplicar todos los valores de una lista.
- Concatenar todas las palabras de una lista.

Estos problemas implican procesar los elementos de la lista para obtener un único valor. Esto se puede abstraer a una función que llamaremos `fold` ("reducir"), que recibe una lista, un valor inicial y una función de dos argumentos, procesa los elementos de la lista y devuelve un único valor. Esta función tiene una pre-condición: la lista debe poseer al menos un valor para poder ser procesada. La función toma el valor inicial que se pasó como parámetro y el primer valor de la lista, se invoca la función de dos argumentos y ésta retorna un valor. Dicho valor se ocupa para procesar el segundo valor de la lista usando la función de dos argumentos, y asi sucesivamente hasta que se haya procesado la lista completa. El valor que retorna la última invocación a la función de dos argumentos es el valor que devuelve `fold`.

Por ejemplo, para obtener la suma de los valores de una lista, se puede programar de la siguiente forma utilizando `fold`:

```
1 # Funcion de dos argumentos requerida
2 def sumar(x, y):
3   return x + y
4
5 # sumarValoresLista: lista -> num
6 # suma los valores dentro de la lista y devuelve el resultado
```

```
# ejemplo: si unaLista = lista(10, lista(20, lista(30, listaVacia)))
# sumarValores(unaLista) devuelve 60
def sumarValoresLista(unaLista):
  return fold(unaLista, 0, sumar)
```

Para multiplicar los valores de una lista, se puede usar el siguiente código:

```
# Funcion de dos argumentos requerida
def multiplicar(x, y):
  return x * y

#s umarValoresLista: lista -> num
# multiplica los valores dentro de la lista y devuelve el resultado
# ejemplo: si unaLista = lista(5, lista(3, lista(3, listaVacia)))
# sumarValores(unaLista) devuelve 45
def sumarValoresLista(unaLista):
  return fold(unaLista, 1, multiplicar)
```

Veremos la implementación de `fold` en la Sección 10.6.

10.5 Funciones anónimas

Para evitar el tener que estar definiendo funciones auxiliares que luego son utilizadas sólo como parámetro de las funciones `filtro`, `mapa`, o `fold`, es posible definir *funciones anónimas*. Estas funciones tienen una declaración muy compacta en el código y son funciones "sin nombre", ya que están pensadas como funciones que se utilizan *una única vez*.

Para definir funciones anónimas en Python se utiliza la instrucción `lambda`. La sintáxis es la siguiente:

```
lambda id1, id2, ...: expresion
```

Los identificadores `id1, id, ...` corresponde a los parámetros de la función anónima, y `expresion` es la expresión que evalúa la función (y devuelve el resultado). Por ejemplo, una función anónima que suma dos valores se implementa como:

```
lambda x,y: x + y
```

Una función anónima booleana que verifica si un número es menor que 5 se implementa como:

```
lambda x: x < 5
```

Recuerde que las funciones anónimas están pensadas como funciones que se utilizan una sola vez y luego se desechan. Si la función debe invocarse más de una vez, debe definirse de la manera usual siguiendo la receta de diseño *y no declarar dos veces la misma función anónima*, ya que esto correspondería a duplicación de código, lo que es una mala práctica de programación.

10.6 Resumen de funciones abstractas para listas

A continuación se muestran las implementaciones de las funciones `filtro`, `mapa` y `fold` en su forma más genérica. Los tests se implementan usando funciones anónimas.

10.6.1 Función `filtro`

Note que el contrato en esta implementación es una versión más general que el visto en la Sección 10.3.5, ya que ahora la función booleana recibe un único parámetro.

```
# filtro: (X -> bool) lista(X) -> lista(X)
# devuelve lista con todos los valores donde operador devuelve True
def filtro(operador, unaLista):
  if vacia(unaLista):
    return listaVacia
  else:
    if operador(cabeza(unaLista)):
      return lista(cabeza(unaLista), filtro(operador, cola(unaLista)))
    else:
      return filtro(operador, cola(unaLista))

# Tests
valores = lista(6, lista(4, lista(8, listaVacia)))
assert filtro(lambda x: x < 5, valores) == lista(4, listaVacia)
valores = lista('a', lista('b', lista('c', lista('d', listaVacia))))
assert filtro(lambda x: x >= 'b' and x < 'd', valores) == \
             lista('b', lista('c', listaVacia))
```

10.6.2 Función mapa

```
# mapa : (X -> Y) lista(X) -> lista(Y)
# devuelve lista con funcion aplicada a todos sus elementos

def mapa(f, unaLista):
  if vacia(unaLista):
    return listaVacia
  else:
    return lista(f(cabeza(unaLista)), mapa(f, cola(unaLista)))

# Tests
valores = lista(1, lista(2, lista(3, lista(4, listaVacia))))
assert mapa(lambda x: 10*x, valores) == \
            lista(10, lista(20, lista(30, lista(40, listaVacia))))
valores = lista("pedro", lista("juan", lista("diego", listaVacia)))
assert mapa(lambda x: x.upper(), valores) == \
            lista("PEDRO", lista("JUAN", lista("DIEGO", listaVacia)))
```

10.6.3 Función fold

```
# fold: (X X -> X) X lista(X) -> X
# procesa la lista con la funcion f y devuelve un unico valor
# el valor init se usa como valor inicial para procesar el primer
# valor de la lista y como acumulador para los resultados
# parciales
# pre-condicion: la lista debe tener al menos un valor

def fold(f, init, unaLista):
  if vacia(cola(unaLista)): # un solo valor
    return f(init, cabeza(unaLista))
  else:
    return fold(f, f(init, cabeza(unaLista)), cola(unaLista))
```

```
13
14 # Tests
15 valores = lista(1, lista(2, lista(3, lista(4, listaVacia))))
16 assert fold(lambda x, y: x + y, 0, valores) == 10
17 valores = lista("pedro", lista("juan", lista("diego", listaVacia)))
18 assert fold(lambda x, y: x + y, "", valores) == "pedrojuandiego"
```

Unidad III: Programación Imperativa

Capítulo 11

Mutación[1]

11.1 Memoria para funciones

Sin importar qué tan seguido usemos una función con el mismo argumento, siempre obtendremos el mismo resultado. Las funciones simplemente no tienen memoria sobre sus usos anteriores.

Sin embargo, muchos programas deben recordar algo sobre sus aplicaciones anteriores. Recuerde que un programa típicamente consiste en muchas funciones. En el pasado siempre hemos supuesto que existe una función principal, y que todas las otras funciones auxiliares son invisibles al usuario. En algunos casos, un usuario puede esperar más de un servicio o funcionalidad a partir de un programa, y cada servicio es mejor implementarlo como una función. Cuando un programa provee más de una función como un servicio al usuario, es común que, por conveniencia o porque agregamos alguna interfaz de usuario, las funciones deban tener memoria.

Como este punto es difícil de comprender, estudiaremos un ejemplo sencillo: manejar números de teléfono en una agenda o libreta de direcciones. Un programa de libreta de direcciones provee usualmente al menos dos servicios:

1. un servicio para buscar el número de teléfono de cierta persona; y
2. un servicio para agregar un nombre y un número de teléfono a la libreta.

Estos dos servicios corresponden a dos funciones. Además, introducimos una definición de una estructura para mantener la asociación nombre-número:

```
1  import estructura
2  from lista import *
3
4  #registro: nombre(str) numero(int)
5  estructura.crear("registro", "nombre numero")
6  libretaDeDirecciones = crearLista(registro("Maria", 1),\
7                crearLista(registro("Pedro", 2), listaVacia))
8
9  # buscar: lista(registro) str -> num or False
10 # busca nombre en libreta y devuelve el numero correspondiente
11 # si no encuentra nombre, retorna False
```

[1]Parte de este capítulo fue traducido al español y adaptado de: M. Felleisen et al.: *How to Design Programs*, MIT Press. Disponible en: `www.htdp.org`

```
12 def buscar(libreta, nombre):
13     ...
14
15 # agregarALibreta: str num -> None
16 # agrega nombre y numero a libretaDeDirecciones
17 def agregarALibreta(nombre, numero):
18     ...
```

La primera función recibe como parámetro una lista de asociaciones nombre-número (registro), llamada `libretaDeDirecciones`, y un nombre. La función produce un número, si el nombre está en la lista, o `False` en caso contrario. La segunda función es radicalmente distinta de lo que hemos visto hasta el momento. El usuario debe invocarla con un nombre y un número; cualquier búsqueda posterior (invocación a `buscar`) con ese nombre debería producir ese número.

Imaginemos una interacción con nuestra libreta:

```
1 >>> buscar(libretaDeDirecciones, "Maria")
2 1
3 >>> buscar(libretaDeDirecciones, "Claudio")
4 False
5 >>> agregarALibreta("Claudio", 5)
6 >>> buscar(libretaDeDirecciones, "Claudio")
7 5
```

Las dos primeras instrucciones confirman que María tiene el número 1 y que no hay un número registrado para Claudio. La tercera instrucción agrega el número 5 para Claudio. La última instrucción muestra que el mismo uso de nuestra función `buscar` ahora produce el número de teléfono esperado.

En el pasado, la única forma de lograr el mismo efecto es que el usuario hubiera editado en el codigo de nuestro programa la definición de `libretaDeDirecciones`. Sin embargo, no queremos que los usuarios tengan que editar nuestros programas para obtener el resultado deseado. De hecho, ellos no *deberían* tener acceso al código de nuestros programas. Por lo tanto, estamos forzados a proveer una interfaz con una función que permita tales cambios.

Veamos la definición de nuestra libreta de direcciones en el siguiente código. La función `agregarALibreta` consume un nombre (string) y un número de teléfono (entero). Su cuerpo lo que hace es una asignación de la misma variable agregando el registro correspondiente. La función `buscar` consume una libreta de direcciones y un nombre; su resultado es el número correspondiente o `False` si el nombre no está en la libreta.

```
1  import estructura
2  from lista import *
3
4  #registro: nombre(str) numero(int)
5  estructura.crear("registro", "nombre numero")
6  libretaDeDirecciones = listaVacia
7
8  # agregarALibreta: str num -> None
9  # agrega nombre y numero a libretaDeDirecciones
10 def agregarALibreta(nombre, numero):
11     global libretaDeDirecciones
12     libretaDeDirecciones = crearLista(registro(nombre, numero),\
13             libretaDeDirecciones)
```

```
14
15 # buscar: lista(registro) str -> num or False
16 # busca nombre en libreta y devuelve el numero correspondiente
17 # si no encuentra nombre, retorna False
18 def buscar(libreta, nombre):
19     if vacia(libreta):
20         return False
21     elif cabeza(libreta).nombre == nombre:
22         return cabeza(libreta).numero
23     else:
24         return buscar(cola(libreta), nombre)
```

11.2 Diseñar funciones con memoria

La sección anterior motivó la idea de funciones con memoria. En esta sección discutiremos el diseño de funciones con memoria.

Diseñar funciones con memoria requiere tres pasos importantes:

1. Debemos determinar que un programa requiere memoria.
2. Debemos identificar los datos que irán en la memoria.
3. Debemos entender cuál o cuáles servicios están destinados a modificar la memoria, y cuál o cuáles van a usar la memoria.

La necesidad por el primer paso es obvia. Una vez que sabemos que un programa requiere memoria, debemos realizar un análisis de los datos para la memoria de éste. Es decir, debemos determinar qué clase de datos se pondrán y se sacarán de ahí. Finalmente, debemos diseñar cuidadosamente las funciones para el programa que van a cambiar la memoria.

11.2.1 La necesidad de tener memoria

Afortunadamente, es relativamente fácil darse cuenta de cuándo un programa necesita mantener memoria. Como se discutió anteriormente, hay dos situaciones. La primera involucra programas que proveen más de un servicio a los usuarios. Cada servicio corresponde a una función. Un usuario puede aplicar esas funciones en el intérprete de Python, o pueden ser aplicadas en respuesta a la acción de un usuario en una interfaz gráfica. La segunda involucra un programa que provee un solo servicio y que es implementado con una sola función a nivel de usuario. Pero el programa puede producir diferentes resultados cuando es aplicada con los mismos argumentos o parámetros.

Volvamos al ejemplo de la libreta de direcciones. Vimos cómo una función agrega registros a la libreta de direcciones y otra busca nombres en ella. Claramente, el uso del "servicio de adición" afecta los usos posteriores del "servicio de búsqueda", y por lo tanto requiere memoria. De hecho, la memoria en este caso corresponde a un objeto físico natural: la libreta de direcciones que la gente guardaba antes de la existencia de dispositivos electrónicos.

La segunda clase de memoria también tiene ejemplos clásicos. Uno de ellos es el típico contador. Cada vez que se aplica la función producirá un número distinto:

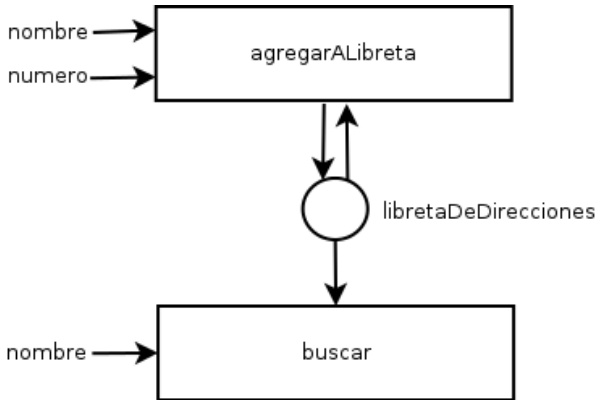


Figura 11.1: Organización de programas con memoria, para el ejemplo de la libreta de direcciones.

```
1 contador = 0
2 # cuenta: None -> int
3 # En la invocacion i devuelve i
4 def cuenta():
5     global contador
6     contador = contador + 1
7     return contador
```

Cada vez que se aplique la función, devolverá un valor distinto. En este caso la mutación o modificación de la memoria se hace al modificar la variable `contador`. En Python es necesario usar la instrucción `global` antes de la variable que se quiere modificar. Si no se incluye la instrucción, el intérprete supondrá que se está definiendo una nueva variable. El ejemplo de arriba no funcionaría sin esta instrucción, dado que `contador` aún no ha sido definida.

En general, cuando se analiza el enunciado de un problema, es conveniente dibujar la organización del programa con memoria, como en la Figura 11.1. Cada servicio es representado como una caja rectangular, las flechas que van hacia las cajas indican qué tipo de dato consume el servicio; las flechas que salen indican lo que producen. La memoria se representa con círculos. Una flecha de un círculo hacia una caja significa que el servicio usa la memoria como una entrada; una flecha hacia un círculo significa que el servicio cambia la memoria. La figura muestra que los servicios comúnmente usan la memoria y la modifican.

11.2.2 Memoria y variables de estado

La memoria se implementa con definiciones de variable. En principio, una sola variable es suficiente para implementar todas las necesidades de memoria, como hemos visto hasta ahora, pero usualmente es inconveniente. Típicamente, el análisis de memoria sugiere cuántas variables necesitamos y cuáles servicios necesitan esas variables. Cuando la memoria cambia, las variables correspondientes adquieren un nuevo valor, o dicho de otra forma, el estado de la declaración de la variable cambia y refleja el cambio de memoria en el tiempo. Por lo tanto, nos referiremos a las variables que implementen memoria como *variables de estado*.

Cada servicio en un programa corresponde a una función que puede usar funciones auxiliares. Un servicio que cambia la memoria de un programa es implementado con una función que usa la asignación (`=`) en alguna o algunas de las variables de estado. Para entender cómo una función debería modificar una variable de estado, necesitamos saber qué tipo de valores puede representar esta variable y cuál

es su propósito. En otras palabras, debemos definir un contrato y un propósito para las variables de estado de la misma manera en que desarrollamos contratos para definir una función.

Volvamos nuevamente al ejemplo de la libreta de direcciones. Ésta tiene una variable de estado: `libretaDeDirecciones`. Su propósito es representar una lista de registros, donde cada uno de éstos consiste en dos campos: un nombre y un número. Para documentar que `libretaDeDirecciones` puede representar sólo una lista de esta naturaleza, debemos agregar un contrato, tal como sigue:

```
1 import estructura
2 from lista import *
3
4 # registro: nombre(str) numero(int)
5 estructura.crear("registro", "nombre numero")
6
7 # libretaDeDirecciones: [registro]*
8 # para mantener los pares de nombres y numeros de telefono
9 libretaDeDirecciones = listaVacia
```

Por la definición del contrato, es posible usar `listaVacia` como el valor inicial de `libretaDeDirecciones`.

Del contrato de la variable de estado, podemos concluir que la siguiente asignación no tiene sentido:

```
1 libretaDeDirecciones = 5
```

La instrucción anterior asigna `5` a `libretaDeDirecciones`, lo cual no es una lista. Por lo tanto, la expresión viola el contrato. Pero la instrucción

```
1 libretaDeDirecciones = listaVacia
```

es válida, porque vuelve a asignar a `libretaDeDirecciones` su valor inicial. A continuación, otra asignación:

```
1 libretaDeDirecciones = lista(registro("Gonzalo", 42), \
2     libretaDeDirecciones)
```

Esto nos ayuda a entender cómo las funciones pueden cambiar el valor de `libretaDeDirecciones` de manera útil. Dado que `libretaDeDirecciones` consiste en una lista de registros, al ejecutar `lista(registro("Gonzalo", 42), libretaDeDirecciones)` se construye una lista más larga del tipo adecuado.

11.2.3 Funciones que inicializan memoria

Después de haber definido contratos y propósitos para las variables de estado de un programa, inmediatamente se define una función que asigna a la variable de estado un valor adecuado. Llamaremos a esta función como *función de inicialización* o *inicializador*. El inicializador de un programa es la primera función que es usada durante una ejecución; un programa también puede proveer de otros medios para invocar al inicializador.

Para nuestros ejemplos, los inicializadores son directos:

```
# inicLibretaDeDirecciones: None -> None
def inicLibretaDeDirecciones():
  global libretaDeDirecciones
  libretaDeDirecciones = listaVacia

# inicContador: None -> None
def inicContador():
  global contador
  contador = 0
```

A primera vista, pareciera que estos inicializadores no agregan mucho valor a nuestros programas: Ambos asignan a las respectivas variables de estado sus valores iniciales. Para ambos casos, sin embargo, el inicializador podría hacer más trabajo útil. En el primer ejemplo, se podría crear una interfaz gráfica para el usuario maneje la libreta de direcciones.

11.2.4 Funciones que cambian la memoria

Una vez que tenemos las variables de estado y sus inicializadores, nuestra atención se debe enfocar en diseñar las funciones que modifican la memoria de un programa. Estas funciones no sólo producen o consumen datos, sino que también afectan las definiciones de las variables de estado. Por lo tanto, hablaremos del *efecto* que estas funciones tienen sobre las variables de estado.

Revisemos las etapas más básicas de la receta de diseño y de cómo podemos acomodar los efectos en las variables de estado:

- **Análisis de Datos:**

  Las funciones que afectan el estado de las variables pueden consumir y (posiblemente) producir datos. Por lo tanto, aún necesitamos analizar cómo representar la información y, si es necesario, introducir definiciones de estructuras y de datos.

  En el ejemplo de la `libretaDeDirecciones`, la definición de los datos debe considerar el tipo de datos de los elementos de la lista y que la lista puede ser de un largo arbitrario.

- **Contrato, propósito y efecto:**

  El primer cambio grande concierne al segundo paso. Además de especificar qué consume una función y qué produce, debemos describir adicionalmente a cuáles variables de estado afecta y cómo afecta a estas variables. El efecto de una función debe ser consistente con el propósito de la variable.

  Por ejemplo, la especificación para la `libretaDeDirecciones` es la siguiente:

```
# agregarALibreta: str num -> None
# efecto: agregar registro(nombre, telefono) al comienzo de
# la libretaDeDirecciones
def agregarALibreta(nombre, telefono): ...
```

  Podemos ver que la definición de `libretaDeDirecciones` se modifica de forma que es coherente con su propósito y su contrato.

- **Ejemplos del programa:**

  Los ejemplos son tan importantes como siempre, pero formularlos puede ser más difícil. Como antes, debemos definir ejemplos que muestren la relación entre entradas y salidas, pero debido a

que las funciones ahora tienen efectos, necesitamos ejemplos que los ilustren.

Debido a que la variable de estado `libretaDeDirecciones` puede tener infinitos valores, es imposible hacer una lista comprehensiva de ejemplos. Lo importante es establecer algunos de ellos, dado que los ejemplos harán más fácil el desarrollar después el cuerpo de la función:

```
# si libretaDeDirecciones es lista vacia y
# evaluamos agregarALibreta("Maria", 1),
# entonces libretaDeDirecciones es lista(registro("Maria", 1),
# listaVacia)

# si libretaDeDirecciones es lista(registro("Claudio", 5),
# listaVacia) y evaluamos agregarALibreta("Gonzalo", 42),
# entonces libretaDeDirecciones es lista(registro("Gonzalo", 42),
# lista(registro("Claudio", 5"), listaVacia))

# si libretaDeDirecciones es lista(registro1, lista(registro2,
# lista(..., listaVacia))...), y evaluamos
# agregarALibreta("Benjamin", 1729), entonces
# libretaDeDirecciones es lista(registro("Benjamin",  1729),
# lista(registro1, lista(registro2, lista(..., listaVacia)))...)
```

El lenguaje de los ejemplos involucran palabras de naturaleza temporal. Esto no debería ser sorprendente, ya que las asignaciones enfatizan la noción de tiempo en la programación.

- **La Plantilla:**

  La plantilla de una función que cambia el estado de las variables es igual que en una función ordinaria, pero el cuerpo debería contener expresiones de asignación (=) sobre variables de estado para especificar que las variables de estado están siendo modificadas. En Python se utiliza la instrucción `global` para identificar una variable de estado:

```
def funcionQueCambiaEstado(x, y, z):
  global variableDeEstado
  variableDeEstado = ...
```

  En ocasiones, también pueden haber condiciones de por medio. Por ejemplo, se puede querer cambiar el estado de una variable dependiendo del valor de un parámetro o de otra variable de estado.

- **El cuerpo de la función:**

  Como siempre, el desarrollo de la función completa requiere de un entendimiento sólido de los ejemplos (cómo son calculados) y de la plantilla. Para las funciones con efectos, la expresión de asignación es el paso más demandante. En algunos casos, el lado derecho de la asignación involucra sólo operaciones primitivas, los parámetros de la función y la variable de estado (o varias de ellas). En otros casos, lo mejor es escribir una función auxiliar (sin efecto) que consuma el valor actual de la variable de estado y los parámetros de la función, y que produzca un nuevo valor para la variable de estado.

La función `agregarALibreta` es un ejemplo del primer tipo. El lado derecho de la asignación consiste en la variable `libretaDeDirecciones`, `lista` y la creación de un `registro`.

- **Testing:**

  Es una tarea complicada verificar que las funciones en estos casos tengan el efecto deseado.

  Hay dos maneras de testear funciones con efectos. Primero, podemos asignar la variable de estado a un estado deseado, aplicar la función, y luego verificar que la función tiene el efecto deseado y el resultado esperado.

  Para el ejemplo del contador es bastante sencillo:

```
contador = 3
siguiente = cuenta()
assert siguiente == 4
```

  El caso de la libreta de direcciones es más complicado:

```
libretaDeDirecciones = listaVacia
agregarALibreta("Benjamin", 1729)
assert lista(registro("Benjamin", 1729), listaVacia) \
    == libretaDeDirecciones
```

  En este test sólo verificamos que `"Benjamin"` y `1729` hayan sido agregados a una lista vacía.

  En la segunda forma de testear, podemos capturar el estado de una variable de estado antes de que sea testada, luego aplicar la función que cambia la memoria y finalmente conducir a los tests apropiados. Considere la siguiente expresión:

```
libretaActual = libretaDeDirecciones
agregarALibreta("John", 10)
assert lista(registro("John", 10), libretaActual) \
     == libretaDeDirecciones
```

  Ésta define en `libretaActual` el valor de `libretaDeDirecciones` al comienzo de la evaluación, y al final verifica que la entrada o registro ha sido agregada al frente de la lista y que nada ha cambiado para el resto de la libreta.

  Para conducir a pruebas sobre funciones con efectos, especialmente a pruebas del segundo tipo, es útil de abstraer la expresión del test en una función:

```
# testLibretaDeDirecciones: str num -> bool
# para determinar si agregarALibreta tiene el efecto deseado
# sobre libretaDeDirecciones y no mas que eso
# efecto: el mismo que agregarALibreta(nombre, numero)
def testLibretaDeDirecciones(nombre, numero):
  libretaActual = libretaDeDirecciones
  agregarALibreta(nombre, numero)
  return lista(registro(nombre, numero), libretaActual) \
      == libretaDeDirecciones
```

Usando esta función, ahora podemos fácilmente testear agregarALibreta varias veces y asegurarnos que cada vez el efecto es el adecuado:

```
assert testLibretaDeDirecciones("Claudio", 5)
assert testLibretaDeDirecciones("Maria", 1)
assert testLibretaDeDirecciones("Gonzalo", 42)
assert testLibretaDeDirecciones("John", 10)
```

- **Reuso futuro:**

  Una vez que tenemos un programa completo y testeado, deberíamos recordar *qué* calcula y cuáles son sus *efectos*. Sin embargo, no necesitamos saber cómo funciona. Si nos encontramos con una situación donde se necesita el mismo cálculo y los mismos efectos, podemos rehusar el programa como si fuera una operación primitiva. En presencia de efectos, es mucho más difícil reusar una función que en el mundo de los programas algebraicos o sin efectos, es decir, utilizando sólo programación funcional.

```
import estructura
from lista import *

# registro: nombre(str) numero(int)
estructura.crear("registro", "nombre numero")

# Variables de estado:
# libretaDeDirecciones: [registro]*
# para mantener los pares de nombres y numeros de telefono
libretaDeDirecciones = listaVacia

# agregarALibreta: str num -> None
# proposito: la funcion siempre produce None
# efecto: agregar registro(nombre, numero) al principio de
# libretaDeDirecciones

# Encabezado:
# def agregarALibreta(nombre, numero): ...

# Ejemplos:
# si libretaDeDirecciones es lista vacia y
# evaluamos agregarALibreta("Maria", 1),
# entonces libretaDeDirecciones es
# lista(registro("Maria", 1), listaVacia)

# si libretaDeDirecciones es lista(registro("Claudio", 5), \
# listaVacia) y evaluamos agregarALibreta("Gonzalo", 42),
# entonces libretaDeDirecciones es lista(registro("Gonzalo", 42),
# lista(registro("Claudio", 5"), listaVacia))

# Plantilla: omitida

# Definicion:
def agregarALibreta(nombre, telefono):
```

```
35    global libretaDeDirecciones
36    libretaDeDirecciones = lista(registro(nombre, telefono), \
37      libretaDeDirecciones)
38
39  # Tests:
40  # testLibretaDeDirecciones: str num -> bool
41  # para determinar si agregarALibreta tiene el efecto deseado
42  # sobre libretaDeDirecciones y no mas que eso
43  # efecto: el mismo que agregarALibreta(nombre, numero)
44  def testLibretaDeDirecciones(nombre, numero):
45    libretaActual = libretaDeDirecciones
46    agregarALibreta(nombre, numero)
47    return lista(registro(nombre, numero), libretaActual) \
48      == libretaDeDirecciones
49
50  assert testLibretaDeDirecciones("Claudio", 5)
51  assert testLibretaDeDirecciones("Maria", 1)
52  assert testLibretaDeDirecciones("Gonzalo", 42)
53  assert testLibretaDeDirecciones("John", 10)
```

11.3 Estructuras mutables

Consideremos las estructuras que se pueden definir con el módulo `estructura`:

```
1  >>> import estructura
2  >>> estructura.crear("artista", "nombre instrumento")
3  >>> p = artista("Michael Weikath", "guitar")
4  >>> p.instrumento
5  "guitar"
6  >>> p.instrumento = "bass"
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't␣set␣attribute
```

El error significa que nuestras estructuras no son *mutables*, esto es, sus campos no son variables de estado. Para manejar estructuras mutables, el módulo `estructura` provee la función `mutable`:

```
1  >>> import estructura
2  >>> estructura.mutable("artista", "nombre instrumento")
3  >>> p = artista("Michael Weikath", "guitar")
4  >>> p.instrumento
5  "guitar"
6  >>> p.instrumento = "bass"
7  >>> p.instrumento
8  "bass"
```

La modificación de la variable de estado de la estructura modifica el valor sólo en la definición de `p`, no la estructura completa.

Estudiemos en mayor profundidad el comportamiento "inusual" al realizar asignaciones en estructuras mutables. Aquí hay un primer ejemplo:

```
>>> estructura.mutable("artista", "nombre instrumento")
>>> p = artista("Michael Weikath", "guitar")
>>> q = p
>>> p.instrumento = "bass"
>>> q.instrumento
"bass"
```

Este ejemplo difiere del primero en dos formas. Primero, define a `q` como `p`. Segundo, la última expresión se hace sobre `q` y no sobre `p`.

Lo que acabamos de ver es el efecto de *compartir* el efecto de una asignación, lo que significa que una modificación de una estructura afecta al programa en más de un lugar. El efecto de compartir también es visible dentro de listas, como muestra el siguiente ejemplo:

```
>>> estructura.crear("lista", "val sig")
>>> estructura.mutable("artista", "nombre instrumento")
>>> unaLista = lista(artista("Michael Weikath", "guitar"), None)
>>> (unaLista.val).artista = "Roland Grapow"
>>> (unaLista.val).artista
"Roland Grapow"
```

Finalmente, los efectos pueden ser compartidos entre los elementos de diferente listas:

```
>>> estructura.crear("lista", "val sig")
>>> estructura.mutable("artista", "nombre instrumento")
>>> q = lista(artista("Michael", "guitar"), None)
>>> r = lista(q.val, lista((q.val).instrumento, None))
>>> (q.val).instrumento = 'vocals'
>>> (r.val).instrumento
"vocals"
```

La nueva definición introduce la variable `r`, una lista con dos elementos. Como `r` contiene al `artista` como primer elemento y como el campo `instrumento` del `artista` es `"vocals"`, el resultado es `"vocals"`. Sin embargo, el programa todavía tiene conocimiento de `"guitar"` en alguna parte. ¿Puede encontrar dónde?

En resumen, la asignación sobre estructuras mutables nos da bastante poder. No sólo podemos crear nuevas estructuras y revelar sus contenidos, sino que también podemos *cambiar* sus contenidos, mientras que las estructuras se mantienen igual. Ahora tenemos que comprender qué significa esto para el diseño de programas.

11.4 Diseñar funciones que modifican estructuras

En la sección anterior se introdujo la idea de estructuras mutables. Ahora necesitamos saber cuándo y cómo aplicar este nuevo poder.

11.4.1 ¿Por qué mutar estructuras?

Cada vez que aplicamos el constructor de estructuras, estamos creando una nueva estructura. En algunas ocasiones, esto es justo lo que queremos. Considere una función que consume una lista de registros de personal y produce una lista de registros telefónicos. Los registros de personal pueden contener información como la dirección de cada persona, el número de teléfono, la fecha de nacimiento, salario, etc. Un registro telefónico debería contener el nombre y el número de teléfono y nada más. Este tipo de programas definitivamente debería crear una nueva estructura por cada valor en la lista dada.

Sin embargo, en otras ocasiones crear una nueva estructura no corresponde a la intuición. Suponga que queremos darle un aumento a alguien. La única forma de lograr esto hasta el momento era crear un nuevo registro de personal que contuviera toda la información anterior y la nueva información del salario. O, suponga que alguien se cambió de casa y tiene un nuevo número de teléfono, y nos gustaría actualizar nuestra lista de contactos. Tal como el programa que cambia el nivel de salario de una persona, el programa que actualiza la lista de contactos crearía un nuevo registro. En la realidad, sin embargo, no crearíamos un nuevo registro de personal ni una nueva entrada en la lista de contactos. En vez de eso, corregiríamos el registro actual en ambos casos. Un programa debería ser capaz de hacer la misma acción correctiva, y con mutación, podemos desarrollar tales programas.

11.4.2 Receta de diseño estructural y con mutación

No necesitamos una nueva receta si consideramos la mutación dentro de nuestra programación, siempre y cuando los campos mutados contengan valores atómicos (por ejemplo, no consisten en otras estructuras). Mientras el diseño de programas sin mutación requiere la combinación de valores, programar con mutación requiere la combinación de efectos, por lo que la clave es agregar una descripción bien formulada del efecto de una función en su contrato, y declarar ejemplos que ilustren tales efectos.

Suponga que nos dan una estructura y una definición de datos para registros de personal:

```
1 import estructura
2
3 # Un registro de empleado (RE) es una estructura
4 # RE(n, d, s)
5 # donde n es un string, d es un string y s es un entero
6
7 # RE: nombre(str) direccion(str) salario(int)
8 estructura.mutable("RE", "nombre direccion salario")
```

Una función que consume uno de estos registros se basa en la siguiente plantilla:

```
1 def funcionParaEmpleado(re):
2   ...re.nombre...
3   ...re.direccion...
4   ...re.salario...
```

Considere una función para subir el salario a un empleado:

```
1 # aumentarSalario: RE int -> None
2 # efecto: modificar el campo salario de unEmpleado al sumar unAumento
3 def aumentarSalario(unEmpleado, unAumento):
4   ...
```

El contrato especifica que la función recibe un RE y un número entero. El propósito es también un efecto, que explica cómo el argumento de `aumentarSalario` se modifica.

Desarrollar ejemplos para `aumentarSalario` requiere las técnicas vistas anteriormente. Específicamente, debemos poder comparar el estado anterior y posterior de una estructura RE:

```
re1 = empleado("John", "Su casa", 1729)
aumentarSalario(re1, 3000)
assert re1.salario == 4729
```

Ahora podemos usar la plantilla y el ejemplo para definir la función:

```
# aumentarSalario: RE int -> None
# efecto: modificar el campo salario de unEmpleado al sumar unAumento
def aumentarSalario(unEmpleado, unAumento):
  unEmpleado.salario = unEmpleado.salario + unAumento
```

Como es usual, la definición completa de la función usa sólo una de las subexpresiones disponibles en la plantilla, pero ésta nos recuerda qué información podemos usar: los argumentos y sus partes, y qué partes podemos modificar.

¿Qué pasa si los campos de una estructura son, a su vez, estructuras? Suponga que queremos simular un juego de cartas. Cada carta tiene dos características importantes: su *tipo* y su *valor*. La colección de cartas de un jugador se llama *mano*. Vamos a suponer que la mano de un jugador nunca está vacía, esto es, siempre tiene al menos una carta en su mano.

Una mano consiste de una estructura `mano` con `valor`, `tipo` y `siguiente` como campos. El campo siguiente puede contener dos tipos de valores: vacío (`None`), que indica que no hay más cartas, y una estructura `mano`, que contiene las cartas restantes. De una perspectiva más general, una `mano` es una lista de cartas, pero sólo la última contiene vacío (`None`) como valor en `siguiente`.

Al principio, un jugador no tiene cartas. Al sacar la primera se creará su mano. Las otras cartas son puestas en la mano existente cuando sean necesarias. Esto llama a dos funciones: una para crear la mano y otra para insertar una carta en la mano. Debido a que la mano existe sólo una vez y corresponde a un objeto físico, es natural pensar que la segunda función es la que modifica un valor existente en vez de crear uno nuevo. Por ahora vamos a aceptar esta premisa y explorar sus consecuencias.

Crear una mano es simple y fácil de implementar:

```
# crearMano: valor tipo -> mano
# para crear una mano de una sola carta de v y t
def crearMano(v, t):
  return mano(v, t, None)
```

Agregar una carta a la mano es un poco más difícil. Para simplificar, diremos que un jugador siempre agrega nuevas cartas al final de la mano. En este caso debemos procesar un valor arbitrariamente grande, lo que significa que necesitamos una función recursiva:

```
# agregarAlFinal: valor tipo mano -> None
# efecto: agrega la carta con valor v y tipo t al final de unaMano
def agregarAlFinal(v, t, unaMano):
  ...
```

Esta especificación dice que la función tiene un valor invisible como resultado que se comunica con el resto del programa sólo por medio de sus efectos.

Veamos algunos ejemplos:

```
mano0 = crearMano(13, "trebol")
```

Si fuéramos a evaluar la siguiente expresión:

```
agregarAlFinal(1, "diamante", mano0)
```

en el contexto de esta definición, `mano0` se convierte en una mano con dos cartas: un káiser de trébol seguido de un as de diamante. Si agregamos:

```
agregarAlFinal(2, "corazones", mano0)
```

En este contexto, `mano0` es una mano con tres cartas. La última es un 2 de corazones. En términos de una evaluación, la definición de `mano0` debería cambiar a:

```
mano0 = crearMano(13, "trebol", crearMano(1, "diamante", \
    crearMano(2, "corazones", None)))
```

Dado que el `valor` y el `tipo` pasados a `agregarAlFinal` son valores atómicos, la plantilla debe estar basada en la definición de los datos de tipo `mano`:

```
def agregarAlFinal(valor, tipo, unaMano):
  if unaMano.siguiente == None:
    ...unaMano.valor ... unaMano.tipo ...
  else:
    ...unaMano.valor ... unaMano.tipo ...
    agregarAlFinal(valor, tipo, unaMano.siguiente)
```

La plantilla consiste en dos cláusulas, que verifican el contenido del campo `siguiente` de `unaMano`. Es recursiva en la segunda cláusula, debido a que la definición de datos de las `mano`s es auto-referencial en esa cláusula. En resumen, la plantilla es convencional.

El siguiente paso a considerar es cómo la función debería afectar a `unaMano` en cada cláusula:

1. En el primer caso, el campo `siguiente` de `unaMano` es vacío (`None`). En ese caso, podemos modificar el campo `siguiente` de forma que contenga la nueva carta:
   `unaMano.siguiente = crearMano(v, t)`
   Recuerde que la nueva `mano` creada contiene vacío (`None`) en su campo `siguiente`.
2. En el segundo caso, la recursión natural agrega una nueva carta la final de `unaMano`. De hecho, debido a que la `unaMano` dada no es la última en la cadena, la recursión natural resuelve el problema.

Esta es la definición completa de `agregarAlFinal`:

```
# agregarAlFinal: valor tipo mano -> None
# efecto: agrega la carta con valor v y tipo t al final de unaMano
```

```
3 def agregarAlFinal(valor, tipo, unaMano):
4   if unaMano.siguiente == None:
5     unaMano.siguiente = crearMano(valor, tipo)
6   else:
7     agregarAlFinal(valor, tipo, unaMano.siguiente)
```

Capítulo 12

Estructuras Indexadas

En este capítulo se estudiarán estructuras de datos indexadas, que permiten acceder a los valores de los datos almacenados en ellas a través de un índice. Algunas de estas estructuras son mutables, pero otras no. También se estudiarán instrucciones que permiten iterar sobre los valores de estas estructuras indexadas.

12.1 Arreglos

Un *arreglo* es una estructura de datos mutable que consiste en una secuencia contigua de un número fijo de elementos homogéneos almacenados en la memoria. En la siguiente figura se muestra un arreglo de enteros con diez valores:

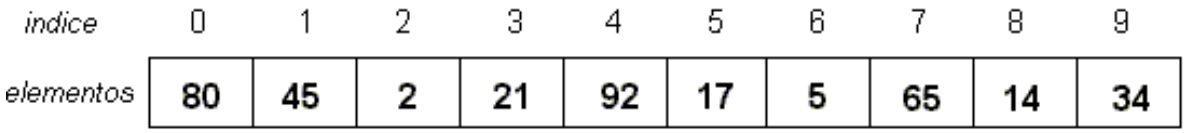


Figura 12.1: Ejemplo de un arreglo con diez valores.

Para acceder a un elemento del arreglo se utiliza un índice que identifica a cada elemento de manera única. Los índices son números enteros correlativos y, en la mayoría de los lenguajes de programación, comienzan desde cero. Por lo tanto, si el arreglo contiene n elementos el índice del último elemento del arreglo es $n - 1$. Una ventaja que tienen los arreglos es que el costo de acceso de un elemento del arreglo es constante, es decir no hay diferencias de costo entre accesar el primer, el último o cualquier elemento del arreglo, lo cual es muy eficiente. La desventaja es que es necesario definir a priori el tamaño del arreglo, lo cual puede generar mucha pérdida de espacio en memoria si se definen arreglos muy grandes para contener conjuntos pequeños de elementos.

Python posee una biblioteca para crear y manipular arreglos de variables numéricas, pero en vez de esta estructura estudiaremos otra que denominaremos *listas de Python*. Estas listas son mucho más flexibles que los arreglos, permiten guardar cualquier tipo de dato dentro de éstas (como las listas recursivas que ya conocen) y además proveen de varias funciones útiles.

12.2 Listas de Python

Las listas de Python son estructuras mutables definidas en el lenguaje Python que, al igual que los arreglos, indexan los elementos que se insertan en ella. Los índices son números enteros correlativos y parten en cero. Las siguientes instrucciones sirven para crear una lista vacía:

```
>>> unaLista = list() # lista vacia
>>> otraLista = [] # lista vacia
```

También se pueden crear listas con valores ya insertados, como se muestra en el ejemplo siguiente. Recuerde que las listas de Python permiten insertar datos de cualquier tipo en sus casilleros:

```
>>> Enteros = [1, 2, 3, 4, 5] # lista de int
>>> Strings = ['casa', 'arbol', 'planta', 'auto'] # lista de str
>>> Todo = ['a', 17, True, 9.5] # lista con datos de distinto tipo
```

Dos listas se pueden concatenar con el operador '+', o se puede repetir varias veces el contenido de una lista multiplicándola por un escalar:

```
>>> lista1 = [10, 20]
>>> lista2 = [50, 60]
>>> lista2 + lista1
[50, 60, 10, 20]
>>> lista1 * 3
[10, 20, 10, 20, 10, 20]
>>> lista1 + [30]
[10, 20, 30]
```

Para conocer el largo de una lista (esto es, cuántos valores contiene), se utiliza la función `len` de Python:

```
>>> lista = ['ana', 'maria', 'luisa', 'teresa']
>>> len(lista)
4
```

Para acceder a los valores en una lista se utiliza el índice del casillero correspondiente. No olvide que el primer valor de la lista corresponde al índice cero. Por ejemplo:

```
>>> lista = ['ana', 'maria', 'luisa', 'teresa']
>>> lista[0]
'ana'
>>> lista[3]
'teresa'
>>> lista[4]

Traceback (most recent call last):
File "<pyshell#20>", line 1, in <module>
lista[4]
IndexError: list index out of range
```

Note en el código anterior que cuando se intentó acceder a un casillero de la lista con un valor de índice inválido (por ejemplo, se sale del rango de los índces de la lista), se produjo un `IndexError`.

Las listas son estructuras mutables, por lo que se pueden modificar los valores almacenados en sus casilleros:

```
>>> lista = [10, 20, 30, 40, 50, 60]
>>> lista[3] = 'X'
>>> lista
[10, 20, 30, 'X', 50, 60]
```

Una función útil predefinida de Python es la función `range`, que retorna una lista de Python (`list`) que contiene una progresión aritmética de enteros. Esta función provee tres contratos distintos:

```
range(stop): int -> list(int)
range(start, stop): int int -> list(int)
range(start, stop, step): int int int -> list(int)
```

La primera versión retorna una lista que comiencia en cero y termina en `stop - 1`. La segunda versión retorna la lista `[start, start+1, ..., stop-2, stop-1]`. En la tercera versión, el parámtero `step` específica el incremento (o decremento) entre números consecutivos de la lista.

Note que para una lista de largo n, la función `range(n)` retorna la lista `[0, 1, 2, ..., n-1]`, que corresponden a los índices válidos para acceder a los casilleros de la lista.

12.3 Iterar sobre estructuras indexadas

Se define un ciclo como un bloque de instrucciones que se repiten de acuerdo a ciertas condiciones definidas. El ejecutar las instrucciones de un ciclo una vez se denomina iteración. Es necesario entonces contar con instrucciones que definan las condiciones que harán que se itere un ciclo o que se terminen las iteraciones. Para este fin, existen dos instrucciones principales: `for` y `while`.

12.3.1 Instrucción `for`

La instrucción `for` de Python permite iterar sobre una estructura indexada. La sintaxis de la instrucción `for` es la siguiente:

```
for identificador in estructura:
  # bloque de instrucciones
```

donde `identificador` es una variable y `estructura` es el identificador de la estructura sobre la cual se va a iterar. En cada iteración del ciclo, se le asignará a la variable un valor almacenado en la estructura indexada (por ejemplo, los valores almacenados en una lista), que se indica después de la instrucción `in`. Por ejemplo:

```
>>> lista = [10, 20, 30, 40, 50]
>>> for valor in lista:
...     print valor
10
20
30
40
50
```

En el ejemplo anterior, la instrucción del ciclo se ejecuta cinco veces, una vez por cada asignación realizada a la variable `valor`.

Note que es posible obtener el mismo resultado anterior si es que a la variable se le asigna un valor de una lista retornada por `range`, y luego dicho valor se ocupa como indice para acceder a los distintos casilleros de la lista:

```
>>> lista = [10, 20, 30, 40, 50]
>>> for indice in range(len(lista)):
...     print lista[indice]
10
20
30
40
50
```

Para acceder sólo a un cierto rango de la lista, se utiliza la notación `[i:j]`, donde `i` indica el primer índice a ser considerado, y `j-1` indica el último índice a ser considerado (note que el índice `j` se excluye del rango). Por ejemplo:

```
>>> cuadrados = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> # alternativamente: cuadrados = [x**2 for x in range(0,11)]
>>> cuadrados[3:6] # solo mostrar indices 3, 4 y 5
[9, 16, 25]
```

Finalmente, note que una lista puede contener a otra lista en sus casilleros. Con esto es posible implementar una matriz (una lista de listas de tamaño fijo). Para acceder a los valores de la matriz, se pueden escribir directamente los valores de los índices de la fila y la columna del valor requerido:

```
>>> matriz = [[10, 20, 30], [40, 50, 60], [70, 80, 90]]
>>> matriz[1][2]
60
```

12.3.2 Instrucción `while`

La instrucción `while` es más general que la instrucción `for` de Python, y permite iterar sobre un bloque de instrucciones. La sintaxis de la instrucción `while` es la siguiente:

```
while condicion: # condicion es de tipo boolean
  # bloque de instrucciones
```

Las instrucciones del ciclo `while` se ejecutrán mientras `condicion` (que corresponde a cualquier expresión condicional) sea `True`. En particular, si se coloca como condición el valor de verdad `True`, el ciclo se repetirá indefinidamente, lo que se denomina un *loop infinito*. Es responsabilidad del programador asegurarse que, salvo que por diseño se requiera programar un loop infinito, la condición es `False` en algún momento, lo que permite al programa salir del ciclo y seguir ejecutando el resto de sus instrucciones.

La instrucción `while` se puede utilizar para iterar sobre los valores de una lista de la siguiente forma:

```
>>> lista = [10, 20, 30, 40, 50]
>>> indice = 0
>>> while indice < len(lista):
...     print lista[indice]
...     indice = indice + 1
10
20
30
40
50
```

12.4 Strings como secuencias indexadas de caracteres

En Python, los strings también corresponden a una estructura indexada (pero inmutable, al contrario de las listas). En efecto, es posible utilizar la instrucción `for` para iterar sobre todas las letras de un string:

```
>>> palabra = 'algoritmo'
>>> for caracter in palabra:
...     print caracter
a
l
g
o
r
i
t
m
o
```

12.5 Diccionarios de Python

Un diccionario de Python es una estructura indexada mutable en donde los índices ya no son números correlativos (como en el caso de las listas de Python), sino que son *llaves* que corresponden a cualquier tipo de dato inmutable. Por ejemplo, los Strings y los números pueden ser llaves en un diccionario, pero una lista (que es mutable) no lo puede ser.

Un diccionario guarda pares `llave:valor` en un orden arbitrario. Las llaves deben ser únicas, de forma de poder identificar cada valor almacenado en el diccionario de forma unívoca. Para crear diccionarios, se utiliza la siguiente sintaxis:

```
>>> diccionario1 = {} # diccionario vacio
>>> diccionario2 = dict() # otro diccionario vacio
>>> telefonos = {'ana':21234567, 'pedro':29876543, 'sofia':23344556}
```

Para recuperar un valor se utiliza la misma notación que en las listas, pero colocando la llave correspondiente:

```
>>> telefonos['pedro']
29876543
>>> telefonos['ana']
21234567
```

Capítulo 13

Depuración[1]

13.1 ¿Qué es la depuración?

En un mundo perfecto, nuestros programas funcionan la primera vez que lo ejecutamos. Dada su experiencia, ya saben que, desafortunadamente, no vivimos en un mundo perfecto.

Depurar es el proceso de entender por qué un programa no está funcionando. Después de darse cuenta que el programa no funciona (con el testing), queremos entender por qué el programa no funciona como esperábamos. Después de entender la fuente del error, viene el paso de arreglar el error, pero esto es usualmente mas fácil que entender por qué el programa falló.

Testear y depurar son cosas muy diferentes. Cuando testeamos, estamos comparando la entrada y la salida de un programa con su especificación. Cuando depuramos, estamos estudiando los eventos que hicieron surgir un error.

Hay dos tipos principales de testing: el test unitario y el test de integración. El test unitario prueba una pieza simple de un programa, idealmente aislada del resto del programa. Éste es el tipo de testing que hemos estudiado hasta ahora. El otro tipo de testing es el test de integración, que prueba un programa grande completo. Al probar un programa completo pueden surgir nuevos errores, incluso cuando hemos testeado bien cada función en el programa individualmente. Esto es porque al unir las funciones en un programa grande, es mas probable que encontremos casos excepcionales que no hemos incluido en nuestros tests. Cuando esto ocurre, tenemos que depurar nuestro programa.

Depurar es una habilidad que se aprende, nadie lo hace bien la primera vez, pero es una de las características que diferencia un buen programador de otros. Usualmente, aprender a depurar es un proceso lento, y toma tiempo hasta que uno puede hacer mucho progreso muy rápido. Un aspecto positivo de la depuración es que no sólo se aplica en programación, sino que esta habilidad se puede transferir a otras áreas. Depurar es un proceso de ingeniería que se puede ocupar en muchos sistemas complejos, como por ejemplo en experimentos de laboratorio.

Existen dos herramientas básicas para hacer depuración: la instrucción `print` y la lectura de código. Ser capaz de entender qué hace exactamente el código (no lo que uno piensa que hace) es probablemente la habilidad más importante para hacer depuración.Por esto, la instrucción `print` puede ser útil para conocer, por ejemplo, el valor actual de una variable de estado, y si ésta se modifica correctamente después de alguna instrucción o invocación a función. Una alternativa a la instrucción

[1]Parte de este capítulo fue traducido al español y adaptado de: Eric Grimson and John Guttag, 6.00 Introduction to Computer Science and Programming, Fall 2008. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed 11 04, 2011). License: Creative Commons Attribution-Noncommercial-Share Alike.

`print` es la instruccion `assert`, que nos permite parar el programa cuando algo inesperado ocurre. Hay herramientas más avanzadas para realizar este tipo de procedimiento de depuración, llamadas *depuradores* (en inglés *debuggers*); IDLE tiene uno. Estas herramientas son un poco más complejas, así que no vamos a verlas, pero a los interesados les recomendamos encarecidamente que las investiguen.

Lo más importante que uno tiene que recordar cuando depura es ser sistemático. Esto es lo que separa buenos depuradores de malos depuradores: los buenos depuradores han encontrado una manera sistemática de buscar errores en programas. Lo que hacen es reducir el espacio de búsqueda en el programa en donde puede estar el error, hasta encontrar la fuente de la error. Visto de esta manera, depurar es un proceso de búsqueda en el código de un programa. De la misma manera que cuando una busca una valor en una lista, uno no toma elementos al azar, sino que hace un recorrido sistemático de la lista. Desfortunadamente, esta búsqueda al azar es lo que hace mucha gente cuando están buscando errores.

13.2 El proceso de depuración

Depurar empieza cuando nos damos cuenta que hay una error en el programa. Lo primero que se debe hacer es estudiar el texto del programa, preguntándose: ¿Como el programa podría haber dado este resultado? Es importante notar que la pregunta no es "¿Por qué el programa no ha dado el resultado esperado?',' sino que "¿Por qué ha producido este resultado?" Formular la pregunta de esta manera permite encontrar más fácilmente el error, porque la pregunta es más cercana a lo que hace el programa.

La segunda pregunta es: ¿Será este error parte de una familia de errores? La idea de esta pregunta es que uno no quiere arreglar un defecto nada más, sino que quiere asegurarse que el programa está libre de errores. Si un error proviene, por ejemplo, de un manejo equivocado de la mutación en listas, conviene verificar todas las ocurrencias de manejo de listas en el programa, para ver si el mismo error está presente en otros contextos. Más que arreglar sólo un defecto, uno tiene que detenerse y verificar si el defecto ocurre más de una vez, arreglando todas las instancias del error de una vez, y no empezar la búsqueda de cero cuando un error parecido aparece más adelante. Esto también es parte de ser sistemático en la búsqueda de errores.

La última pregunta es ¿cómo localizar y corregir el error? Para esto se usa el método científico. El método científico empieza estudiando los datos disponibles. En este caso, los datos disponibles son los resultados de los tests del programa. Estos son los resultados equivocados, pero también los resultados correctos. Esto es porque el programa puede funcionar en algunos casos y en otros no; a veces, entender por qué el program funcionó en un caso y no en el otro permite entender la causa del defecto, basado en la diferencia entre el caso correcto y el caso equivocado.

La otra gran pieza de información es el texto del programa. Cuando estudien el programa, recuerden que no lo entienden, porque si lo hubieran entendido de verdad el programa no tendría un defecto. Entonces, hay que leer el program con un ojo crítico.

Mientras uno estudia los datos, el método científico nos hace formular una hipótesis consistente con los datos. No sólo una parte de los datos; todos los datos. Basado en esta hipótesis, se diseña un experimento repetible.

13.3 Depurar programas con el método científico

Les recuerdo que un experimento científico tiene el potencial de demostrar que la hipótesis es falsa. Si no es posible hacer esto, el experimento no es válido científicamente. Para esto, también tenemos

que saber cuál es el resultado esperado. Uno tiene típicamente una hipótesis, y la hipótesis dice que el resultado a obtener tiene que ser un resultado X. Si el resultado no es X, se comprobó que la hipótesis es falsa. Este es el punto donde mucha gente falla en la depuración: no se toman el tiempo de pensar qué es lo que tendría que devolver el programa, o sea, de formar una hipótesis. Si éste es el caso, no están siendo sistemáticos al interpretar los resultados del experimento. Antes de ejecutar cualquier test, tienen que saber que esperan que realice el programa.

Un problema potencial es el problema de la reproducibilidad de los resultados. En general tratamos de tener programas que hacen la misma cosa cada vez que se ejecutan, porque es mas fácil de depurarlos de esta manera. Sin embargo, esto no es siempre posible. Algunos programas hacen uso de números aleatorios, lo que impide que cada ejecución del programa sea idéntica. Dado que para depurar uno tiene que tratar de hacer la ejecución reproducible, algo que se puede hacer en este caso es definir la semilla del algoritmo de generación de números aleatorios. Un algoritmo de generación de números aleatorios genera una secuencia compleja, pero determinística, de números, y si lo inicializamos en un mismo estado (la semilla), siempre reproduce la misma secuencia de números.

Otra cosa que impide la reproducibilidad son las interacciones con el usuario. Si un programa reacciona a interacciones del usuario, no va a ser reproducible. En este caso, hay que hacer que estas interacciones sean definidas en una pequeña parte del programa, de tal manera que la mayor parte del programa sea testeable y depurable de una manera reproducible.

Si pensamos en cómo diseñar un experimento de depuración de un programa, hay varios objetivos:

- El primero es de encontrar la entrada más pequeña posible que conduzca al error. En muchos casos, un programa se va a ejecutar por mucho tiempo antes de encontrar un error. Esto es poco práctico, así que hay que tratar de reducir el problema para encontrarlo de manera más rápida. Por ejemplo, si un juego de palabras no funciona con palabras de 12 letras, podemos probar como funciona con palabras de 3 letras. Si la misma falla ocurre con 3 letras en vez de 12, esto simplifica mucho la resolución del problema, ya que el espacio de búsqueda se reduce bastante. El proceso usual es de tratar de reducir la entrada paso a paso: primero con 11 letras, despues 10, etc., hasta que el programa empeza a funcionar de nuevo.

- El segundo objetivo es encontrar la parte del programa que es la más probable de ser “culpable”. En ambos casos, es recomendable hacer una búsqueda binaria. La idea es que tratamos de descartar la mitad de los datos, o la mitad del programa a cada paso. Esto permite encontrar el lugar del problema rápidamente, incluso si el programa o los datos son grandes. En el caso de un programa, la idea es de ocupar la instrucción `print` para imprimir valores intermedios del programa, y determinar si están correctos. La búsqueda binaria en este caso nos va a hacer empezar en la mitad del programa. Si no encontramos el error, esto quiere decir que se ubica en la segunda mitad del programa. Si lo encontramos, quiere decir que está ubicado en la primera mitad. Después seguimos buscando en cuartos del programa, y asi sucesivamente, hasta llegar a la línea que contiene el error. Con este proceso, a cada paso descartamos la mitad del programa como ”inocente” del error.

En resumen, depurar programas es una tarea compleja que requiere ser sistemático y entender bien qué es lo que hace el programa. De esta forma, será posible encontrar el error para corregirlo. Programar es una tarea compleja y suceptible a muchos errores, incluso los mejores programadores cometen errores al escrbir programas, por lo que es muy importante para todo programador el practicar la habilidad de depuración.

Unidad IV: Programación Orientada al Objeto

Capítulo 14

Objetos y Clases

Hasta ahora en el curso hemos visto dos paradigmas de programación: un paradigma funcional, en donde los problemas se modelan como funciones que toman datos de entrada y retornan un valor simple o compuesto que sólo depende de la entrada, y un paradigma imperativo, en donde hay estructuras que actúan como memoria y por lo tanto los resultados retornados por las funciones no sólo dependen de los valores de los parámetros de la función, sino que también dependen del estado actual de estas estructuras. A partir de este capítulo se estudiará un tercer paradigma de programación, conocido como *programación orientada al objeto*. En este paradigma de programación se utilizan dos conceptos fundamentales: objetos y clases. Éstos forman la base de toda la programación en lenguajes orientados a objetos.

Un *objeto* es un modelo computacional de un ente o concepto que posee ciertos atributos y con el cual podemos realizar ciertas operaciones. Hasta el momento, hemos visto datos complejos (*structs*) que nos permiten modelar los atributos del concepto, y definimos funciones sobre éstos para implementar las distintas operaciones posibles. En cambio, en la programación orientada a objetos, los objetos contienen tanto los datos asociados a éstos como las operaciones que se pueden realizar con ellos. Una *clase* permite describir en forma abstracta los atributos y operaciones del concepto modelado, que luego se instancia en un objeto. En este capítulo estudiaremos como crear objetos y como interactuar con ellos, dejando para el próximo capítulo cómo se define una clase en Python.

14.1 Un ejemplo: automóviles

Suponga que necesita escribir un programa que permita hacer una simulación sobre tráfico de automóviles. Por cada automóvil de la simulación, es necesario almacenar información como: color, velocidad actual, velocidad máxima, cantidad de gasolina en el estanque, etc. Además, se requiere poder realizar las siguientes operaciones con los automóviles: acelerar, frenar, apagar, consultar cuánta gasolina queda en el estanque, cambiar su color, etc. En programación orientada a objetos, para resolver este problema es necesario definir una clase `Automovil` de la cual podamos crear objetos (instancias especificas de automóviles), y luego a través de interacciones con estos objetos programar la simulación.

14.2 Crear e interactuar con objetos

Como ya se mencionó, antes de ver cómo podemos crear nuestras propias clases vamos a ver cómo crear objetos. Supongamos que ya tenemos implementada la clase `Automovil`. Vamos a ver en un pequeño ejemplo cómo *instanciar* distintos objetos.

```
1 >>> unAutomovil = Automovil()
```

Con este pequeño código lo que hemos hecho ha sido crear un objeto de la clase `Automovil`.

Nuestro automóvil también puede moverse, acelerar, frenar, etc. Los objetos tienen *comportamiento*, siguiendo nuestro modelo de la realidad. Supongamos que nuestros objetos de la clase `Automovil` tienen distintas funciones, o *métodos*, que realizan acciones sobre el objeto:

```
>>> unAutomovil.encender()
>>> unAutomovil.acelerar()
>>> unAutomovil.frenar()
>>> unAutomovil.apagar()
```

Nuestros métodos, al ser funciones, también pueden recibir parámetros. Un método indica qué parámetros recibe (en su contrato). Por ejemplo, nuestro automóvil puede esperar saber cuánto tiempo presionar el acelerador:

```
unAutomovil.acelerar(30) # presiona el acelerador durante 30 segundos
```

Estas llamadas a métodos del objeto son aplicaciones de funciones como las hemos visto siempre, salvo que su contexto está unido al *estado* de cada objeto.

En el ejemplo anterior, supongamos que nuestro automóvil entrega el nivel de gasolina *actual* tras un tiempo en movimiento:

```
>>> gasolina = unAutomovil.obtenerNivelGasolina()
>>> if gasolina > 0:
  unAutomovil.acelerar()
else:
  unAutomovil.frenar()
```

Yendo más allá, los objetos también son valores, al mismo nivel que los enteros, strings, estructuras, etc. Por lo mismo, podemos incluso pasar un objeto *como parámetro* y *retornar* objetos. Por ejemplo, nuestros métodos `acelerar` y *frenar* pueden retornar como resultado el *mismo* objeto, lo que resulta en un patrón particular como el que sigue a continuación:

```
>>> unAutomovil.acelerar().acelerar().frenar().acelerar().frenar()
```

Esto es posible dado que al llamar a `acelerar` o `frenar`, el resultado de la llamada retorna la *misma* instancia de `unAutomovil`, a la cual se puede volver a llamar a los mismos métodos.

14.3 Múltiples instancias de una clase y estado de los objetos

Podemos crear distintas instancias de una clase como objetos. En nuestro ejemplo, podemos crear distintos automóviles:

```
>>> unAutomovil = Automovil()
>>> otroAutomovil = Automovil()
>>> # a hacerlos competir!
```

¿En qué se diferencian dos objetos instanciados de la misma clase? A simple vista no se ve mucha diferencia (ambos objetos pueden acelerar, frenar, encenderse, apagarse, etc.), pero lo que los caracteriza es lo que los diferencia. En lo que hemos visto hasta ahora, si el automóvil acelera

durante 30 segundos, podemos esperar que su nivel de gasolina disminuya. Como dijimos al comienzo, si queremos representar un automóvil por su color, su velocidad, etc. estamos diferenciándolos. Pero también podemos cambiar estos valores (como la gasolina). Estos valores constituyen el *estado* del objeto. Tal como en las estructuras mutables, el estado de un objeto también puede cambiar. Veamos un par de ejemplos.

Si queremos crear un automóvil de cierto color y cierta velocidad máxima, podríamos hacerlo al momento de instanciar el objeto:

```
>>> miAutomovil = Automovil(color="azul", velocidadMaxima=220)
```

Si queremos luego modificar estos valores (supongamos que *enchulamos* nuestro automóvil) también podemos hacerlo:

```
>>> miAutomovil.setColor("rojo")
>>> miAutomovil.setVelocidadMaxima(250)
```

14.4 Ejemplo: libreta de direcciones

Volvamos al ejemplo de la libreta de direcciones del capítulo de Mutación. Supongamos que tenemos dos clases: `Registro` y `Libreta`, con los cuales podemos crear objetos que representen registros en nuestra libreta, con nombre, teléfono y dirección; y una forma de crear distintas libretas con nombre:

```
>>> libretaPersonal = Libreta("personal")
>>> libretaTrabajo = Libreta("trabajo")
>>> registro1 = Registro(nombre="Juan Gonzalez", \
  telefono="777-7777", direccion="Calle ABC 123")
>>> libretaTrabajo.agregarRegistro(registro1)
```

Hasta ahora hemos creado distintas libretas y un registro, y hemos dado un estado específico a cada libreta y registro. Así como antes teníamos una función para buscar un registro dado un nombre, también podemos hacer lo mismo usando nuestra representación en objetos, suponiendo que tenemos los métodos adecuados:

```
>>> john = libretaTrabajo.buscar("John")
>>> john.setTelefono("133")
>>> john.getTelefono()
"133"
```

Capítulo 15

Definición de Clases

En este capítulo estudiaremos cómo definir una clase en Python, cómo definir los campos de la clase, cómo definir la construcción de un objeto, y cómo definir métodos en la clase. Como ejemplo, implementaremos una clase que nos permita crear objetos para manejar fracciones (como en el capítulo de datos compuestos).

15.1 Clase

Para definir una clase en Pyhon se utiliza la instrucción `class`. En esta instrucción se debe señalar el nombre que tendrá la clase. Por convención, los nombres de las clases comienzan con mayúscula. En este punto, y como parte de la receta de diseño, señalaremos los campos (también llamados *variables de instancia*) que tendrá la clase y sus tipos. Veremos una primera versión de nuesta clase para manejar fracciones, que denominaremos `FraccionV1`:

```
1 # Campos :
2 # numerador : int
3 # denominador : int
4 class FraccionV1:
```

15.1.1 Campos

Los campos de una clase son variables de estado que nos permiten almacenar información sobre los objetos de dicha clase. Para la clase `FraccionV1` necesitamos al menos dos campos: uno para almacenar el numerador de la fracción, y otro para almacenar el denominador de la fracción. Al ser variables de estado, su valor se puede modificar haciendo la asignación correspondiente al valor nuevo.

15.2 Constructor

El constructor es el primer método que uno debe definir en una clase. Este método se ejecuta cada vez que se crea un nuevo objeto de la clase. Usualmente, en este método es en donde se definen los campos de la clase y sus valores iniciales. Para la clase `FraccionV1`, el constructor es el siguiente:

```
1     # Constructor
2     def __init__(self, numerador = 0, denominador = 1):
3         # Inicializacion de campos
4         self.numerador = numerador
5         self.denominador = denominador
```

En Python, el método constructor siempre tiene el nombre `__init__`, y sólo se puede definir un constructor por clase. Todo método de una clase en Python (incluyendo al constructor) tiene como primer parámetro la palabra clave `self`, que es una referencia al objeto que se está creando, aunque cuando uno crea un objeto *no coloca nada para dicho parámetro*. Note que el constructor para la clase `FraccionV1` recibe además dos parámetros, el numerador y el denominador. Si el usuario no los específica al crear el objeto, se especifica que esas variables tendrán los valores 0 y 1 por default. Por ejemplo:

```
>>> f = FraccionV1(1, 2) # crea la fraccion 1/2
>>> f = FraccionV1() # crea la fraccion 0/1
```

Dentro del constructor se definen e inicializan las dos variables de instacias: `self.numerador` y `self.denominador`. Note que es necesario anteponer `self.` cada vez que se desee accesar o modificar dichos campos, sino Python interpreta que el programador se está refiriendo a variables locales del método. Es usual definir e inicializar todos los campos dentro del constructor, aunque es posible agregar campos a la clase posteriormente, definiendo nuevas variables de estado en otros métodos de la clase.

15.3 Métodos

Los métodos de una clase se definen igual que las funciones en Python, con la diferencia que se definen dentro del contexto de una clase y deben tener como primer parámetro la referencia `self`. Note que este primer parámetro `self` no es parte del contrato del método. Por ejemplo, definamos un método para la clase `FraccionV1` que nos permita sumar dos fracciones:

```
    # suma: FraccionV1 -> FraccionV1
    # devuelve la suma de la fraccion con otra fraccion
    def suma(self, fraccion):
        num = self.numerador * fraccion.denominador + \
              fraccion.numerador * self.denominador
        den = self.denominador * fraccion.denominador
        return FraccionV1(num, den)
```

El método es muy similar a la función `suma` que implementamos en el capítulo de datos compuestos. Note `self` corresponde al objeto que invoca al método `suma` y `fraccion` corresponde al objeto que se pasó por parámetro al método. Por ejemplo, en el siguiente código:

```
f1 = FraccionV1(1, 2)
f2 = FraccionV1(5, 6)
f3 = f1.suma(f2)
```

el objeto `f1` corresponde a `self` en el método `suma`, y el objeto `f2` corresponde al parámetro `fraccion` en dicho método.

15.3.1 Métodos accesores y mutadores

Los métodos de una clase se pueden dividir en dos categorías: accesores (*accesors*) y mutadores (*mutators*). Los métodos accesores son aquellos que sólo acceden al contenido de los campos del objeto, pero no los modifican. Por ejemplo, el método `suma` es un ejemplo de un método accesor. Por otra parte, los métodos mutadores son aquellos que modifican (o pueden modificar) los valores de los campos de la clase. Al igual que en las funciones con memoria que modifican el estado de una variable, debemos indicar el efecto que puede tener un método mutador sobre los campos de la clase.

Por ejemplo, veamos un método mutador que permite simplificar una fracción. Para esto, se debe modificar tanto el numerador como el denominador de la fracción, dividiendo ambos valores por el

máximo común divisor. Suponiendo que disponemos de la función `mcd(x,y)` que calcula el máximo común divisor entre dos números enteros `x` e `y`, podemos implementar el método `simplificar` de la siguiente forma:

```
    # simplificar: None -> None
    # efecto: simplifica la fraccion, puede modificar los
    # valores de los campos numerador y denominador
    def simplificar(self):
        valor = mcd(self.numerador, self.denominador)
        if valor > 1:
            self.numerador = self.numerador / valor
            self.denominador = self.denominador / valor
```

Es muy típico definir en una clase métodos accesores para obtener el valor de los distintos campos, y métodos mutadores para asignarles un nuevo valor. Por convención, los nombres de los métodos accesores comienzan con *get*, y los mutadores comienzan con *set*. Para nuesta clase `FraccionV1`, los métodos correspondientes son los siguientes:

```
    # getNumerador: None -> int
    # devuelve el valor del campo numerador
    def getNumerador(self):
        return self.numerador

    # getDenominador: None -> int
    # devuelve el valor del campo denominador
    def getDenominador(self):
        return self.denominador

    # setNumerador: int -> None
    # efecto: modifica el valor del campo numerador
    def setNumerador(self, numerador):
        self.numerador = numerador

    # setDenominador: int -> None
    # efecto: modifica el valor del campo denominador
    def setDenominador(self, denominador):
        self.denominador = denominador
```

15.4 Receta de diseño de clases

La receta de diseño para clases consiste en los siguientes pasos:

- Antes de definir la clase, se identifican los campos que tendrá y sus tipos correspondientes.
- La definición de los métodos sigue las reglas habituales de la receta de diseño para funciones, pero los cuerpos de los métodos y los tests correspondientes quedan pendientes.
- Una vez terminada la definición de métodos, fuera de la clase se implementan los tests para todos los métodos. Esto es así porque es necesario crear objetos de la clase con los cuales invocar los objetos, y dependiendo de los valores de los campos de cada objeto se puede determinar la respuesta esperada a cada método.
- Finalmente, se implementan los cuerpos de los métodos, y luego se ejecutan los tests. Se corrigen los errores detectados en los tests, y se itera nuevamente hasta que todos los tests sean exitosos.

A continuación se presenta la implementación completa de la clase `FraccionV1`. Note que la función `mcd` no es un método de la clase (no tiene como primer parámetro `self`), sino una función auxiliar que debe ser declarada como `global` para poder ser utilizada por los métodos.

Contenido del archivo FraccionV1.py

```
# Campos :
# numerador : int
# denominador : int
class FraccionV1:

    # Constructor
    def __init__(self, numerador = 0, denominador = 1):
        # Inicializacion de campos
        self.numerador = numerador
        self.denominador = denominador

    # getNumerador: None -> int
    # devuelve el valor del campo numerador
    def getNumerador(self):
        return self.numerador

    # getDenominador: None -> int
    # devuelve el valor del campo denominador
    def getDenominador(self):
        return self.denominador

    # setNumerador: int -> None
    # efecto: modifica el valor del campo numerador
    def setNumerador(self, numerador):
        self.numerador = numerador

    # setDenominador: int -> None
    # efecto: modifica el valor del campo denominador
    def setDenominador(self, denominador):
        self.denominador = denominador

    # toString: None -> str
    # devuelve un string con la fraccion
    def toString(self):
        return str(self.numerador) + "/" + str(self.denominador)

    # suma: FraccionV1 -> FraccionV1
    # devuelve la suma de la fraccion con otra fraccion
    def suma(self, fraccion):
        num = self.numerador * fraccion.denominador + \
              fraccion.numerador * self.denominador
        den = self.denominador * fraccion.denominador
        return FraccionV1(num, den)

    # mcd: int int -> int
```

Contenido del archivo FraccionV1.py (cont)

```
46      # devuelve el maximo comun divisor entre dos numeros x e y
47      # ejemplo: mcd(12, 8) devuelve 4
48      global mcd
49      def mcd(x, y):
50          if x == y:
51              return x
52          elif x > y:
53              return mcd(x-y, y)
54          else:
55              return mcd(x, y-x)
56
57      # Test
58      assert mcd(12, 8) == 4
59
60      # simplificar: None -> None
61      # efecto: simplifica la fraccion, puede modificar los
62      # valores de los campos numerador y denominador
63      def simplificar(self):
64          valor = mcd(self.numerador, self.denominador)
65          if valor > 1:
66              self.numerador = self.numerador / valor
67              self.denominador = self.denominador / valor
68
69  # Tests
70  f1 = FraccionV1(1, 2)
71  f2 = FraccionV1(5, 6)
72  # Test de accesors
73  assert f1.getNumerador() == 1
74  assert f2.getDenominador() == 6
75  # Test de mutators
76  f2.setNumerador(3)
77  f2.setDenominador(4)
78  assert f2.getNumerador() == 3 and f2.getDenominador() == 4
79  # Test de metodo suma
80  # El siguiente test es incorrecto
81  # assert f1.suma(f2) == FraccionV1(10, 8)
82  # El siguiente test es correcto
83  f3 = f1.suma(f2)
84  assert f3.getNumerador() == 10 and f3.getDenominador() == 8
85  # Test de metodo toString
86  assert f3.toString() == "10/8"
87  # Test de metodo simplificar
88  f3.simplificar()
89  assert f3.getNumerador() == 5 and f3.getDenominador() == 4
```

La clase `FraccionV1` contiene métodos accesadores y mutadores. Si se desea implementar la clase sólo usando métodos accesadores (es decir, se opera con los objetos de forma puramente funcional), se puede hacer eliminando todo método mutador de la clase o modificándolo a una versión accesadora,

en donde el resultado se almacena en un nuevo objeto de la misma clase. Como detalle adicional, para evitar que un usuario fuera de la clase pueda modificar los campos de un objeto, en Python se pueden defir con nombres que comiencen con los caracteres `__` (dos caracteres de guión bajo), y esto los hace inaccesibles fuera de la clase (si uno intenta modificarlos, Python arroja el error `AttributeError`). La implementación de la clase `FraccionV2`, que sólo utiliza métodos accesores es la siguiente:

Contenido del archivo FraccionV2.py

```
# Campos :
# numerador : int
# denominador : int
class FraccionV2:

    # Constructor
    def __init__(self, numerador = 0, denominador = 1):
        # Inicializacion de campos
        # campos invisibles al usuario
        self.__numerador = numerador
        self.__denominador = denominador

    # getNumerador: None -> int
    # devuelve el valor del campo numerador
    def getNumerador(self):
        return self.__numerador

    # getDenominador: None -> int
    # devuelve el valor del campo denominador
    def getDenominador(self):
        return self.__denominador

    # toString: None -> str
    # devuelve un string con la fraccion
    def toString(self):
        return str(self.__numerador) + "/" + str(self.__denominador)

    # suma: FraccionV2 -> FraccionV2
    # devuelve la suma de la fraccion con otra fraccion
    def suma(self, fraccion):
        num = self.__numerador * fraccion.__denominador + \
              fraccion.__numerador * self.__denominador
        den = self.__denominador * fraccion.__denominador
        return FraccionV2(num, den)

    # mcd: int int -> int
    # devuelve el maximo comun divisor entre dos numeros x e y
    # ejemplo: mcd(12, 8) devuelve 4
    global mcd
    def mcd(x, y):
        if x == y:
            return x
        elif x > y:
```

Contenido del archivo FraccionV2.py (cont)

```
44                 return mcd(x-y, y)
45             else:
46                 return mcd(x, y-x)
47
48         # Test
49         assert mcd(12, 8) == 4
50
51         # simplificar: None -> FraccionV2
52         # devuelve la fraccion simplificada
53         def simplificar(self):
54             valor = mcd(self.__numerador, self.__denominador)
55             num = self.__numerador / valor
56             den = self.__denominador / valor
57             return FraccionV2(num, den)
58
59     # Tests
60     f1 = FraccionV2(1, 2)
61     f2 = FraccionV2(3, 4)
62     # Test de accesors
63     assert f1.getNumerador() == 1
64     assert f2.getDenominador() == 4
65     # Test de metodo suma
66     f3 = f1.suma(f2)
67     assert f3.getNumerador() == 10 and f3.getDenominador() == 8
68     # Test de metodo toString
69     assert f3.toString() == "10/8"
70     # Test de metodo simplificar
71     f4 = f3.simplificar()
72     assert f4.getNumerador() == 5 and f4.getDenominador() == 4
```

Capítulo 16

Interacciones entre Objetos[1]

En capítulos anteriores hemos visto qué son los objetos y las clases, y cómo se implementan en Python. En particular, discutimos las nociones de campo, constructor y métodos cuando hablamos de definición de clases.

Ahora iremos un paso más adelante. Para construir aplicaciones interesantes, no es suficiente el construir objetos independientes. En efecto, nos interesaría que los objetos puedan combinarse entre sí, de tal manera que juntos puedan realizar una tarea común. En este capítulo desarrollaremos esta idea a través de una pequeña aplicación de ejemplo que involucra tres objetos y un conjunto de métodos que permitan cumplir con su tarea.

Consideremos un reloj digital. Este tipo de relojes tiene una pantalla en la cual se muestran las horas y los minutos, separados por el símbolo dos puntos (:). Así, estos relojes son capaces de mostrar la hora desde las 00:00 (medianoche) hasta las 23:59 (un minuto antes de medianoche).

16.1 Abstracción y modularidad con objetos

Una primera idea para implementar el reloj puede ser desarrollar una única clase. Después de todo, es precisamente este el enfoque que hemos seguido hasta ahora: cómo construir clases para desarrollar un trabajo. Sin embargo, inspirándonos en lo que ya sabemos sobre *abstracción* y *modularidad*, notamos que esta no es la mejor manera de proceder. La idea principal es identificar subcomponentes en el problema que se puedan descomponer en clases separadas. La razón de esto es poder manejar apropiadamente la complejidad. En efecto, mientras más grande es un problema, se vuelve cada vez más difícil llevar un registro de todos los detalles que hay que manejar al mismo tiempo.

La solución que usaremos para manejar la complejidad cuando desarrollamos programas usando objetos es la *abstracción*. Dividiremos el problema en subproblemas, y luego cada subproblema en sub-subproblemas hasta que los problemas individuales sean lo suficientemente pequeños y manejables como para poder desarrollarlos con una clase sencilla (a esto se le denomina *modularidad*). Una vez que hayamos resuelto uno de estos subproblemas, no nos preocuparemos más de los detalles de éste, sino que consideraremos esta solución como un elemento que podemos reutilizar en el subproblema siguiente. Típicamente, a esta estrategia la llamamos *dividir–y–conquistar* o *dividir–para–reinar*.

Así, la modularidad y la abstracción se complementan entre sí. La modularidad es el proceso de dividir problemas grandes en partes más pequeñas, mientras que la abstracción es la habilidad de

[1]Parte de este texto fue traducido al español y adaptado de: Chapter 3 "Object interaction", in David J. Barnes and Michael Kölling: *Objects First with Java - A Practical Introduction using BlueJ*, Fifth Edition, Prentice Hall.

ignorar detalles para enfocarse en el plano general.

Los mismos principios de modularidad y abstracción que acabamos de revisar se aplican en el desarrollo de software orientado a objetos. Para ayudarnos a mantener una visión general en programas complejos, intentamos identificar subcomponentes que podamos programar como entidades independientes. Luego, tratamos de usar esas subcomponentes como si fueran unidades simples, sin tener que preocuparnos de su complejidad interna.

En la programación orientada a objetos, estas componentes y subcomponentes son objetos. Si intentáramos por ejemplo construir un auto como si fuera un software, usando un lenguaje orientado a objetos, lo que haríamos sería construir objetos separados para el motor, la caja de cambios, una rueda, un asiento, entre otros, en lugar de modelar el auto como un objeto simple y monolítico. El identificar qué tipo de objetos (y qué clases) se deben incluir en un sistema para un problema dado no siempre es una tarea sencilla.

Volvamos al ejemplo del reloj digital. Usando los conceptos de abstracción que hemos revisado, nos gustaría encontrar la mejor manera de escribir una o más clases para implementarlo. Una forma de ver el problema es considerar al reloj como una pantalla con cuatro dígitos (dos para las horas y dos para los minutos). Si ahora realizamos una abstracción a más alto nivel, podemos ver al reloj como dos entidades distintas de dos dígitos cada una (un par para representar las horas, y otro par para los minutos). Así, un par empieza en 0, aumenta en 1 cada hora, y vuelve a 0 cuando alcanza su límite 23. El otro par vuelve a 0 cuando su valor alcanza el límite 59. Lo similar en el comportamiento de estas dos entidades nos da para pensar que podemos abstraer aún más el problema, y por ende, dejar de ver al reloj como una combinación de horas y minutos.

En efecto, podemos pensar que el reloj está formado por dos objetos que pueden mostrar valores enteros que comienzan en 0 hasta cierto límite. Este valor puede aumentar, pero, si alcanza el límite, se reinicia a 0. Ahora sí tenemos un nivel de abstracción apropiado que podemos representar como una clase: un par de números programables. Así, para programar la pantalla del reloj, primero debemos implementar una clase para manejar un par de números, luego darle un método para obtener su valor, y dos métodos para asignar un valor y aumentarlo. Una vez que hayamos definido esta clase, simplemente bastará con crear dos objetos de esta clase (cada cual con diferentes límites) para así construir el reloj completo.

16.2 Diagramas de clases y objetos

Tal como lo discutimos anteriormente, para programar el reloj, necesitamos primero construir una representación para un par de números. Este par tiene que almacenar naturalmente dos valores: uno es el límite hasta el cual se puede contar sin tener que reiniciar a cero; el otro es el valor actual. Representaremos estos dos valores como campos enteros en la clase `ParDeNumeros`:

```
1 # Campos:
2 # limite: int
3 # valor: int
4 class ParDeNumeros:
5     ...
```

Veremos más adelante los detalles de implementación de esta clase. Primero, asumamos que podemos construir esta clase, y pensemos un poco más en cómo podemos organizar el reloj completo.

Nos gustaría poder construir el reloj, a partir de un objeto que tenga internamente dos pares de números (uno para las horas y otro para los minutos). Así, cada uno de estos pares de números sería

un campo para un objeto de la clase `Reloj`:

```
# Campos:
# horas: ParDeNumeros
# minutos: ParDeNumeros
class Reloj:
   ...
```

La estructura de objetos descrita puede visualizarse usando el siguiente *diagrama de objetos*. En este diagrama apreciamos que un objeto de la clase `Reloj` se instancia utilizando dos objetos de la clase `ParDeNumeros`.

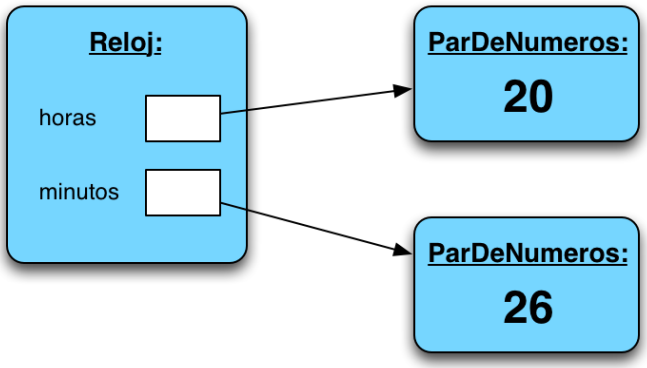


Figura 16.1: Diagrama de objetos

Asimismo, la siguiente figura muestra el *diagrama de clases* que modela el problema del reloj digital.

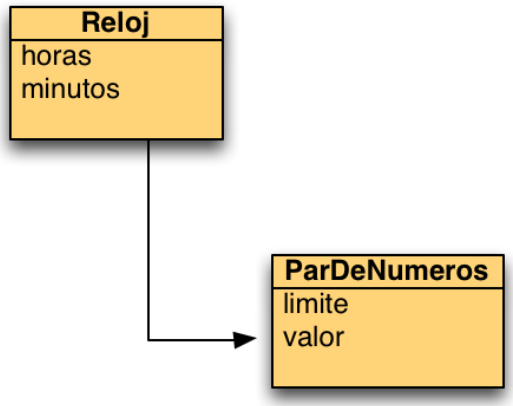


Figura 16.2: Diagrama de clases

Notemos que el diagrama de clases muestra únicamente dos clases, mientras que el diagrama de objetos muestra tres objetos. Esto se debe al hecho que podemos crear más de un objeto desde una misma clase. En este caso, creamos dos objetos `ParDeNumeros` desde la clase `ParDeNumeros`.

Estos dos diagramas ofrecen distintas vistas de la misma aplicación. El diagrama de clases muestra la *vista estática*. Muestra qué es lo que tenemos al momento de escribir el programa. Así, tenemos dos clases, y la flecha entre ellas indica que la clase `Reloj` hace uso de la clase ParDeNumeros. En otras palabras, en el código fuente de la clase `Reloj` aparecerá una o más referencias a la clase `ParDeNumeros`.

Para comenzar el programa, crearemos un objeto de la clase `Reloj`. Programaremos así la pantalla del reloj digital de tal manera que cree automáticamente dos objetos `ParDeNumeros`. En efecto, el diagrama de objetos muestra esta situación en tiempo de ejecución, es decir, cuando la aplicación está corriendo. Esto también recibe el nombre de *vista dinámica*.

El diagrama de objetos también muestra otro detalle importante: cuando una variable almacena un objeto, el objeto no es almacenado directamente en la variable, sino que una *referencia* al objeto es almacenado en la variable. En el diagrama, la variable se representa como una caja blanca, y el objeto es mostrado como una flecha. El objeto referenciado es almacenado fuera del objeto que referencia, y la referencia de objetos enlaza a ambos.

Ahora analizaremos la implementación del reloj digital. Primero, debemos programar la clase `ParDeNumeros`. En esta clase, notamos los dos campos que discutimos más arriba, un constructor, y cuatro métodos. El constructor recibe el valor del límite como parámetro. Así, por ejemplo, si recibe 24 como parámetro, el valor se reiniciará a 0 cuando se llegue a ese valor. De esta forma, el rango para el valor que se puede almacenar en este caso va de 0 a 23. Con esto, recordemos que podemos definir correctamente las horas y los minutos a manejar en el reloj: para las horas usamos un límite de 24, y para los minutos, un límite de 60.

```
1  # Campos:
2  # limite: int
3  # valor: int
4  class ParDeNumeros:
5
6    # Constructor: crea un objeto que almacena dos numeros y se reinicia
7    # a cero cuando se sobrepasa el limite
8    def __init__(self, limite):
9      self.limite = limite
10     self.valor = 0
11
12   # getValor: None -> int
13   # Retorna el valor actual
14   def getValor(self):
15     return self.valor
16
17   # setValor: int -> None
18   # efecto: Reemplaza el valor del par al nuevo valor indicado.
19   # Si el nuevo valor es menor que cero, o sobre el limite,
20   # no hacer nada.
21   def setValor(self, nuevoValor):
22     if (nuevoValor >= 0) and (nuevoValor < self.limite):
23       self.valor = nuevoValor
24
25   # toString: None -> str
26   # Retorna el valor almacenado en el par, esto es, un string que
27   # contiene los numeros del par; si el valor es menor que diez, se le
```

```
28    # debe anteponer un cero
29    def toString(self):
30      if self.valor < 10:
31        return "0" + str(self.valor)
32      else:
33        return str(self.valor)
34
35    # aumentar: None -> None
36    # efecto: Aumenta en una unidad el valor almacenado en el par,
37    # reiniciando a cero si se sobrepasa el limite
38    def aumentar(self):
39      self.valor = (self.valor + 1) % self.limite
```

Ahora que ya vimos cómo construir una clase que define un número de dos dígitos, miraremos más en detalle a la clase `Reloj`.

```
1  # Campos:
2  # horas: ParDeNumeros
3  # minutos: ParDeNumeros
4  # pantalla: str
5  class Reloj:
6
7    # Constructor: crea un objeto reloj. Si no recibe parametros,
8    # inicia el reloj a las 00:00; si no, a la hora indicada
9    def __init__(self, horas=0, minutos=0):
10     self.horas = ParDeNumeros(24)
11     self.minutos = ParDeNumeros(60)
12     self.setReloj(horas, minutos)
13
14   # tic: None -> None
15   # Se debe llamar cada minuto y hace que el reloj avance un minuto
16   def tic(self):
17     self.minutos.aumentar()
18     if self.minutos.getValor() == 0:
19       self.horas.aumentar()
20     self.actualizarPantalla()
21
22   # setReloj: int int -> None
23   # efecto: Fija la hora del reloj a la hora y minuto especificados
24   def setReloj(self, hora, minuto):
25     self.horas.setValor(hora)
26     self.minutos.setValor(minuto)
27     self.actualizarPantalla()
28
29   # getHora: None -> str
30   # Devuelve la hora actual del reloj en el formato HH:MM
31   def getHora(self):
32     return self.pantalla
33
34   # actualizarPantalla: None -> None
35   # efecto: Actualiza el string interno que lleva cuenta de la hora
36   # actual
```

```
37    def actualizarPantalla(self):
38      self.pantalla = self.horas.toString() + ":" + self.minutos.toString()
```

16.3 Objetos que crean objetos

Una pregunta que podemos hacernos es: ¿de dónde salen estos tres objetos? Cuando queramos usar un reloj digital, nosotros crearemos un objeto `Reloj`. Suponemos que nuestro reloj digital tiene horas y minutos. Luego, por el sólo hecho de crear un reloj digital, nosotros esperamos que se hayan creado implícitamente dos objetos de la clase `ParDeNumeros`, uno para las horas y otro para los minutos.

Como los implementadores de la clase `Reloj`, debemos asegurarnos que esto realmente pase. Para esto, simplemente escribimos código en el constructor de `Reloj` que crea y guarda dos objetos de `ParDeNumeros`. Ya que el constructor es automáticamente llamado cuando un objeto `Reloj` es creado, los objetos `ParDeNumeros` serán creados de manera automática también. Este es el código del constructor de `Reloj` que hace este trabajo:

```
1  # Campos:
2  # horas: ParDeNumeros
3  # minutos: ParDeNumeros
```

Otros campos omitidos

```
1  class Reloj:
2
3    # Constructor: crea un objeto reloj. Si no recibe parametros,
4    # inicia el reloj a las 00:00; si no, a la hora indicada
5    def __init__(self, horas=0, minutos=0):
6      self.horas = ParDeNumeros(24)
7      self.minutos = ParDeNumeros(60)
8      self.setReloj(horas, minutos)
```

Métodos omitidos

Cada una de estas dos líneas en el constructor crea un nuevo objeto `ParDeNumeros` y los asignan a una variable. Como ya hemos visto, la sintaxis para crear un nuevo objeto es:

NombreDeClase(lista-de-parámetros)

La construcción de un objeto hace dos cosas:

1. Crea un nuevo objeto de la clase llamada (en este caso `Reloj`)
2. Ejecuta el constructor de la clase

Si el constructor de la clase está definido para tener parámetros, entonces deben ser suministrados al crear el objeto. Por ejemplo, el constructor de la clase `ParDeNumeros` fue definido para esperar un parámetro del tipo entero:

```
1  class ParDeNumeros:
2    def init (self, limite):
```

Es importante notar que en Python la definición del constructor de una clase siempre contiene como primer parámetro `self` que no se considera como un argumento al momento de crear un objeto. Así, para crear un objeto de la clase `ParDeNumeros` debemos proveer de un parámetro de tipo entero:

```
1 ParDeNumeros(24)
```

Luego, con el constructor de la clase `Reloj` hemos conseguido lo que queríamos: cuando creamos un objeto de esta clase, su constructor será ejecutado automáticamente y creará dos objetos de la clase `ParDeNumeros`. Es decir, este objeto crea a su vez otros dos objetos cuando es creado y nuestra clase `Reloj` está lista para ser usada.

16.4 Llamadas a métodos

Un método puede llamar a otros métodos dentro de una misma clase como parte de su implementación. Esto se denomina *llamada de método interna*. Por otra parte, un método puede llamar a métodos de otros objetos usando un punto como notación. Esto se denomina *llamada de método externa*. En esta sección revisaremos ambos tipos de llamadas.

Llamadas de métodos internas

La última línea del método `tic` de la clase `Reloj` contiene la siguiente declaración:

```
1 self.actualizarPantalla()
```

Esta declaración es una *llamada a un método*. Como hemos visto hasta el momento, la clase `Reloj` tiene un método con la siguiente firma:

```
1 def actualizarPantalla(self):
```

La llamada al método de arriba invoca a este método. Ya que este método está en la misma clase que la llamada al método (en `tic`), también la denominamos *llamada de método interna*. Este tipo de llamadas tienen la siguiente sintaxis:

```
1 self.nombreDelMetodo( lista-de-parametros )
```

En nuestro ejemplo, el método no tienen ningún parámetro, así que la lista de parámetros está vacía. Esto está indicado por el par de paréntesis sin nada dentro de ellos.

Cuando en el código se encuentra una llamada a un método, el método correspondiente es ejecutado, y la ejecución retorna al método en donde fue ejecutada la llamada y continúa con la siguiente instrucción. Para que la firma (el contrato) de un método corresponda a la llamada de un método, tanto el nombre del método como la lista de parámetros deben corresponder. Aquí, ambas listas de parámetros están vacías (puesto que el parámetro `self` no se considera), por lo que corresponden. Esta necesidad de corresponder tanto con el nombre como con la lista de parámetros es importante puesto que la llamada de un método con un numero de parametros equivocados falla, como en el caso de funciones.

En nuestro ejemplo, el propósito de esta llamada al método es actualizar el string a desplegar en la pantalla. Luego que ambos objetos de la clase `ParDeNumeros` han sido creados, el string a desplegar es asignado para mostrar el tiempo indicado por el número de los objetos `ParDeNumeros`. La implementación del método `actualizarPantalla` será explicado a continuación.

Llamadas de métodos externas

Examinemos el siguiente método: `tic`. Su definición está dada por:

```
# tic: None -> None
# Se debe llamar cada minuto y hace que el reloj avance un minuto
def tic(self):
  self.minutos.aumentar()
  if self.minutos.getValor() == 0:
    self.horas.aumentar()
  self.actualizarPantalla()
```

Cuando la pantalla está conectada a un reloj de verdad, este método debería ser llamado cada 60 segundos por el timer electrónico del reloj. Por ahora, nosotros mismos haremos esta llamada en forma manual para probar nuestra clase `Reloj`. Cuando el método `tic` es llamado, la primera sentencia que es ejecutada es:

```
self.minutos.aumentar()
```

Esta declaración llama al método `aumentar` del objeto `minutos`. Luego, cuando uno de los métodos del objeto `Reloj` es llamado, a su vez él llama a un método de otro objeto para hacer parte del trabajo. Una llamada a un método de otro objeto se denomina *llamada de método externa*. La sintaxis de una llamada de este tipo esta dada por:

```
objeto.nombreDeMetodo( lista-de-parametros )
```

Esta notación es conocida como *notación de punto*. Consiste en el nombre de un objeto, un punto, el nombre del método a llamar, y los parámetros de la llamada. Es particularmente importante apreciar que aquí hablamos del nombre de un *objeto* y no del nombre de la clase: Usamos el nombre `minutos` en vez de `ParDeNumeros`.

El método `tic` tiene una sentencia `if` para comprobar que las horas también deben aumentar cuando pasan los 60 minutos. Como parte de esta condición se llama a otro método del objeto `minutos`: `getValor`. Este método retorna el valor actual de los minutos. Si el valor es cero, entonces sabemos que ya han pasado 60 minutos por lo que debemos incrementar las horas. Por otra parte, si el valor no es cero, entonces hemos terminado, puesto que no debemos aumentar las horas. Luego, la declaración `if` no necesita la parte `else`.

Ahora debemos poder entender los tres métodos que nos restan de la clase `Reloj`. El método `setReloj` toma dos parámetros –la hora y el minuto– y asigna el reloj con el tiempo especificado. Mirando al cuerpo del método, podemos ver que esto lo hace llamando a los métodos `setValor` de ambos objetos `ParDeNumeros` (uno para las horas y uno para los minutos). Luego, éste llama a `actualizarPantalla` para actualizar el string de la pantalla.

El método `getHora` es trivial, dado que sólo retorna el string actual de la pantalla. Ya que siempre mantenemos el string de la pantalla actualizado, es todo lo que se debe hacer ahí.

Finalmente, el método `actualizarPantalla` es el responsable de actualizar el string de la pantalla para que éste refleje el tiempo representado por los dos objetos `ParDeNumeros`. Este método es llamado cada vez que el tiempo del reloj cambia. Trabaja llamando a los métodos `getValor` de cada uno de los objetos `ParDeNumeros`. Estos métodos retornan el valor de cada par de número por separado, y luego los usa para crear el string de la pantalla con estos dos valores con una coma de separación.

16.5 Testing de clases

La tarea de prueba de programas orientados al objeto es más compleja que probar programas funcionales, por el hecho que los objetos contienen variables de estado mutables, y que los métodos de un objeto se usan en combinación. Por estas dos razones, un test usualmente necesita definir un escenario de prueba más largo que una simple llamada a una función.

Ahora presentamos una primera manera de probar clases sencillas; volveremos al tema de las pruebas más adelante. La idea es que cada clase de un programa pueden tener una clase de prueba que se encarga de: (1) crear objetos de la clase a probar, y poner estos objetos en estados que queremos probar; (2) ejercitar la funcionalidad de dichos objetos con varias secuencias de métodos; y (3) verificar que el comportamiento es correcto.

Vamos a ver dos ejemplos. El primero es el test de la clase `ParDeNumeros`, donde tenemos que probar que los números aumentan hasta llegar al límite, y que la representación textual de dichos números siempre tiene dos caracteres. Esto se puede hacer de la siguiente manera:

```
# Para simplificar la implementacion de los tests,
# este codigo se incluye en el archivo donde se
# encuentra la definicion de la clase ParDeNumeros
class TestParDeNumeros:

  def __init__(self):
    # crear un objeto con estado interesante
    self.par = ParDeNumeros(3)

  def test(self):
    # ejercitar funcionalidad,
    # y verificar el comportamiento
    assert self.par.getValor() == 0
    self.par.aumentar()
    assert self.par.getValor() == 1
    self.par.aumentar()
    assert self.par.getValor() == 2
    self.par.aumentar()
    assert self.par.getValor() == 0
    self.par.aumentar()
    assert self.par.toString() == "01"

# ejecucion del test
test = TestParDeNumeros()
test.test()
```

Observe que este escenario de test es más complejo que el test de una función única, sin efecto de borde. Seguimos con el ejemplo del test del Reloj, que aún más complejo, dado que tiene que probar que al avanzar los minutos se cambia de minuto y de hora, según el caso. Además, la pantalla tiene que tener el formato correcto.

```
class TestReloj:

  def __init__(self):
    # crear un objeto con estado interesante
    self.reloj = Reloj(23,58)

```

```
7     def test(self):
8       # ejercitar funcionalidad,
9       # y verificar el comportamiento
10      assert self.reloj.getHora() == "23:58"
11      self.reloj.tic()
12      assert self.reloj.getHora() == "23:59"
13      self.reloj.tic()
14      assert self.reloj.getHora() == "00:00"
15      for i in range(60):
16        self.reloj.tic()
17      assert self.reloj.getHora() == "01:00"
18      self.reloj.tic()
19      assert self.reloj.getHora() == "01:01"
20
21  test = TestReloj()
22  test.test()
```

Capítulo 17

Diseño de Clases[1]

En este capítulo veremos algunos de los factores que influencian en el diseño de una clase. ¿Qué hace que una clase esté bien o mal diseñada? Escribir una buena clase puede tomar más esfuerzo en el corto plazo que escribir una mala clase. Sin embargo, en el largo plazo ese esfuerzo extra es justificado con frecuencia. Para ayudarnos a escribir buenas clases existen algunos principios que podemos seguir. En particular, en este capítulo introduciremos la visión de que el diseño de una clase debe ser basado en responsabilidades, y que las clases deben encapsular sus datos.

17.1 Introducción

Es posible implementar una aplicación y conseguir que realice su tarea con clases mal diseñadas. El hecho de ejecutar una aplicación terminada usualmente no indica si está bien estructurada o no.

El problema típicamente aparece cuando un programador de mantenimiento desea hacer algunos cambios a la aplicación existente. Si, por ejemplo, un programador intenta arreglar un *bug* o quiere agregar nuevas funcionalidades a un programa existente, una tarea que debería ser fácil y obvia con clases bien diseñadas podrían ser muy difíciles de realizar y pueden involucrar una gran cantidad de trabajo si las clases están mal diseñadas.

En aplicaciones grandes, este efecto ocurre en etapas tempranas, durante la implementación original. Si la implementación parte con una mala estructura, terminar la aplicación puede convertirse en una tarea en extremo compleja, o imposible de realizar, o que tenga *bugs*, o tome mucho más tiempo para construirse que lo necesario. En el mundo laboral, las compañías que desarrollan software usualmente mantienen, extienden y venden una aplicación a lo largo de muchos años. No es poco común que una implementación de software que se puede comprar en una tienda hoy haya comenzado a construirse hace más de diez años. En esta situación, una compañía de software no puede permitirse tener código mal estructurado.

Dado que muchos de los efectos de una clase mal diseñada se hacen más evidentes cuando tratamos de adaptarla o extender la aplicación a la que pertenece, en este capítulo haremos exactamente eso: utilizaremos como ejemplo el juego *mundo-de-zuul*. Para esto, ocuparemos una implementación mucho más simple y rudimentaria del juego de aventura basado en texto. En su estado original, el juego no es demasiado ambicioso, de hecho está incompleto. Para el final del capítulo, sin embargo, estarás en una posición de ejercitar tu imaginación, diseñar tu propio juego y hacerlo realmente divertido e

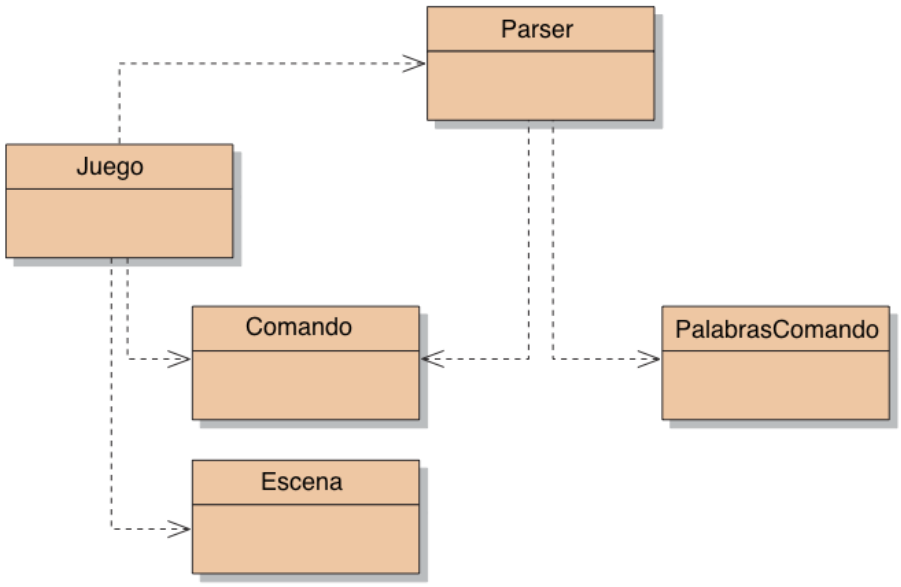
[1]Parte de este texto fue traducido al español y adaptado de: Chapter 6 "Designing classes", in David J. Barnes and Michael Kölling: *Objects First with Java - A Practical Introduction using BlueJ*, Fifth Edition, Prentice Hall.

interesante.

mundo-de-zuul Nuestro juego *mundo-de-zuul* está modelado a partir del juego *Aventura* original que fue desarrollado a principios de los 70' por Will Crowther y extendido por Don Woods. El juego original es también conocido como *Aventura Cueva Colosal*. Este fue un juego sofisticado y maravillosamente imaginativo para su tiempo, que incluía encontrar el camino en un complejo sistema de una cueva, encontrando tesoros escondidos, usando palabras secretas y otros misterios, para alcanzar el máximo de puntaje posible.

Mientras trabajamos en extender la aplicación original, aprovecharemos la oportunidad de discutir aspectos del diseño de clases existentes. Veremos que la implementación contiene ejemplos de mal diseño de clases, y podremos ver cómo esto impacta en nuestras tareas y cómo lo podemos arreglar.

Para partir, analizaremos la estructura de clases de la primera versión de nuestro juego que se observa a continuación:



Como podemos observar, el proyecto se compone de cinco clases. Estas son: `Parser`, `PalabrasComando`, `Comando`, `Escena` y `Juego`. El resumen de lo que cada una de estas clases hace se muestra a continuación:

- *PalabrasComando*: Define todas los comandos válidos en el juego. Esto lo hace almacenando un arreglo de strings que representan las palabras comando.
- *Parser*: El `Parser` lee las líneas desde la consola e intenta interpretarlas como comandos. Crea objetos de la clase `Comando` que representan el comando que fue ingresado.
- *Comando*: Un objeto `Comando` representa un comando que fue ingresado por el usuario. Tiene métodos que nos permite comprobar fácilmente si el comando ingresado es válido, y obtener la primera y segunda palabra que definen el comando como strings separados.

- *Escena*: Representa la ubicación del juego. Las escenas pueden tener salidas o llevar a otras escenas.
- *Juego*: La clase `Juego` es la clase principal del juego. Inicializa el juego y luego entra en un ciclo infinito que lee y ejecuta comandos. También contiene el código que implementa cada comando del usuario.

17.2 Introducción al acoplamiento y la cohesión

Si vamos a justificar nuestra afirmación de que algunos diseños son mejores que otros, debemos definir algunos términos que nos permitan discutir los factores que consideramos importantes en el diseño de una clase. Dos términos centrales cuando hablamos de la calidad del diseño de una clase son *acoplamiento* y *cohesión*.

El término *acoplamiento* se refiere a la interconexión de clases. Ya hemos discutido en capítulos anteriores acerca que nuestro objetivo es diseñar un conjunto de clases que cooperen entre sí y que se comuniquen a través de interfaces bien definidas. El grado de acoplamiento indica qué tan estrechamente estas clases están conectadas. Nos esforzamos para tener un bajo nivel de acoplamiento entre clases, o *acoplamiento holgado*.

El grado de acoplamiento determina qué tan difícil es hacer cambios a una aplicación. En un estructura con clases estrechamente acopladas, un cambio en una clase puede significar el cambio en muchas otras clases. Esto es lo que tratamos de evitar, puesto que el efecto de hacer un cambio pronto se expande a través de la aplicación completa. Además, encontrar todos los lugares donde es necesario hacer cambios y hacerlos puede ser difícil y consumir mucho tiempo.

Por otra parte, en un sistema con bajo nivel de acoplamiento frecuentemente podemos hacer cambios sin hacer ningún cambio a otras clases, y la aplicación aún funcionará. Discutiremos ejemplos de altos y bajos niveles de acoplamiento en este capítulo.

El término *cohesión* se relaciona con el número y diversidad de tareas que una sola unidad es responsable en la aplicación. La cohesión es relevante para unidades de una sola clase y métodos individuales.

Idealmente, una unidad debe ser responsable de una tarea cohesiva (esto es, una tarea que puede ser vista como una unidad lógica). Un método debe implementar una operación lógica, y una clase debe representar un tipo de entidad. La razón principal detrás de este principio es el reuso: si un método o una clase es responsable de sólo una tarea bien definida, entonces es más probable de que sea usada de nuevo en un contexto diferente. Una ventaja complementaria a seguir este principio es que cuando un cambio *es* necesario para algún aspecto de la aplicación, es más probable que encontremos que la piezas relevantes para hacerlo estén ubicadas en un solo lugar.

17.3 Duplicación de código

La duplicación de código es un indicador de un mal diseño. La clase `Juego`, que se muestra a continuación, contiene un caso de duplicación. El problema con duplicación de código es que un cambio en una versión debe ser aplicado en todos los lugares que representen el mismo código para evitar inconsistencias. Esto aumenta la cantidad de trabajo que un programador de mantenimiento debe hacer, e introduce el peligro de insertar *bugs*. Es muy fácil asumir que el trabajo está hecho, pero no hay nada indicando que una segunda copia de este código exista, pudiendo mantenerse sin cambios

de manera incorrecta.

```
class Juego:

  # ... Codigo omitido...
  def crearEscenas(self):
    afuera = Escena("Afuera de la entrada principal de la universidad")
    auditorio = Escena("Sala de conferencias")
    pub = Escena("En el pub del campus")
    lab = Escena("En un laboratorio de computacion")
    oficina = Escena("En la oficina principal de computacion")

    # Inicializar las salidas de la escena
    afuera.setSalidas(None, auditorio, lab, pub)
    auditorio.setSalidas(None, None, None, afuera)
    pub.setSalidas(None, afuera, None, None)
    lab.setSalidas(afuera, oficina, None, None)
    oficina.setSalidas(None, None, None, lab)
    self.escenaActual = afuera

  # ... Codigo omitido...

  # Imprime en pantalla el mensaje de bienvenida para el jugador
  def imprimeBienvenida(self):
    print ''
    print 'Bienvenido al Mundo de Zuul'
    print 'Zuul es un nuevo e increiblemente aburrido juego de aventura'
    print ''
    print 'Tu estas en ' + self.escenaActual.getDescripcion()
    print 'Salidas: '
    if self.escenaActual.salidaNorte != None
      print 'norte',
    if self.escenaActual.salidaEste != None
      print 'este',
    if self.escenaActual.salidaSur != None
      print 'sur',
    if self.escenaActual.salidaOeste != None
      print 'oeste',
    print ''

  # ... Codigo omitido...

  # Intentar ir en una direccion.
  # Si hay una salida, entrar a la nueva escena; sino imprimir mensaje de error
  def irAEscena(self, comando):
    if not comando.tieneSegundaPalabra():
      # Si no hay segunda palabra
      # no sabemos donde ir
      print 'Ir a donde?'
      return
    direccion = comando.getSegundaPalabra()
```

```
50      # Intentar salir de la escena
51      sgteEscena = None
52      if direccion == 'norte':
53          sgteEscena = self.escenaActual.salidaNorte
54      if direccion == 'este':
55        sgteEscena = self.escenaActual.salidaEste
56      if direccion == 'sur':
57        sgteEscena = self.escenaActual.salidaSur
58      if direccion == 'oeste':
59        sgteEscena = self.escenaActual.salidaOeste
60      if sgteEscena == None:
61        print 'No hay puerta!'
62      else:
63        self.escenaActual = sgteEscena
64        print 'Tu estas en ' + self.escenaActual.getDescripcion()
65        print 'Salidas: '
66        if self.escenaActual.salidaNorte != None:
67          print 'norte',
68        if self.escenaActual.salidaEste != None:
69          print 'este',
70        if self.escenaActual.salidaSur != None:
71          print 'sur',
72        if self.escenaActual.salidaOeste != None:
73          print 'oeste',
74        print ''
75
76    # ... Codigo omitido...
```

Ambos métodos `imprimeBienvenida` y `irAEscena` contienen las siguientes líneas de código:

```
1  print 'Tu estas en ' + self.escenaActual.getDescripcion()
2  print 'Salidas: '
3  if self.escenaActual.salidaNorte != None
4    print 'norte',
5  if self.escenaActual.salidaEste != None
6    print 'este',
7  if self.escenaActual.salidaSur != None
8    print 'sur',
9  if self.escenaActual.salidaOeste != None
10   print 'oeste',
11 print ''
```

La duplicación de código es usualmente un síntoma de baja cohesión. El problema aquí tiene sus raíces en el hecho que ambos métodos hacen dos cosas: `imprimeBienvenida` imprime el mensaje de bienvenida e información acerca de la ubicación actual, mientras que `irAEscena` cambia la ubicación actual y luego imprime la información de la (nueva) ubicación actual.

Ambos métodos imprimen información acerca de la ubicación actual, pero ninguno de ellos puede llamar al otro puesto que hacen otras cosas además. *Esto es mal diseño.*

Un mejor diseño sería usar un método separado y más cohesivo, cuya única tarea es imprimir la información de la ubicación actual. De esta forma, ambos métodos `imprimeBienvenida` y `irAEscena` pueden llamar a este método cuando necesitan imprimir la información de la escena actual en la que

se encuentra el usuario. De esta manera, evitamos escribir el código dos veces, y cuando necesitemos cambiarlo, sólo debemos hacerlo una vez.

```
def imprimeInfoUbicacion(self):
  print 'Tu estas en ' + self.escenaActual.getDescripcion()
  print 'Salidas: '
  if self.escenaActual.salidaNorte != None
    print 'norte',
  if self.escenaActual.salidaEste != None
    print 'este',
  if self.escenaActual.salidaSur != None
    print 'sur',
  if self.escenaActual.salidaOeste != None
    print 'oeste',
  print ''
```

17.4 Acoplamiento

Consideremos ahora la posibilidad de agregar dos nuevas salidas: *arriba* y *abajo*. El hecho que en nuestro código existan tantos lugares en donde todas las salidas están enumeradas es un síntoma de un diseño de clase pobre. En los métodos `irAEscena` y `imprimeInfoUbicacion` hay expresiones `if` por cada salida de la variable del tipo `Escena` que estamos analizando. Esta decisión de diseño ahora nos crea trabajo: cuando agregamos nuevas salidas, debemos encontrar todos estos lugares y agregar nuevos casos y expresiones `if` asociadas. ¡Imagina el efecto de esto si decidimos usar direcciones como noroeste, sudeste, etc!

Veamos la implementación del método `setSalidas` de la clase `Escena`:

```
class Escena:

  #... Codigo omitido...

  def setSalidas(self, norte, este, sur, oeste):
    if norte != None:
      self.norte = norte
    if este != None:
      self.este = este
    if sur != None:
      self.sur = sur
    if oeste != None:
      self.oeste = oeste
```

Para mejorar esta situación, utilizaremos un diccionario que almacene todas las salidas en vez de guardarlas en variables separadas. Al hacer esto, deberíamos poder escribir código que pueda manejar cualquier número de salidas sin la necesidad de hacer demasiadas modificaciones en el código. El diccionario contendrá una asociación entre el nombre de la dirección (por ejemplo, `'norte'`) y la escena que está contigua en esa dirección (un objeto de tipo `Escena`). Entonces, cada entrada tiene un string como la llave y un objeto `Escena` como el valor.

Este es un cambio con respecto a cómo una escena almacena información internamente acerca de las escenas vecinas. Teóricamente, este cambio debería afectar solo la *implementación* de la clase `Escena`

(el *cómo* la información de las salidas es almacenada), no la *interfaz* (el *qué* almacena la escena).

Idealmente, cuando sólo la implementación de una clase cambia, otras clases no deberían verse afectadas. Esto significaría tener un bajo nivel de acoplamiento.

En nuestro ejemplo, esto no funciona. Si sacamos las variables que representan las salidas en la clase `Escena` y las reemplazamos con un diccionario, el juego no podrá ejecutarse de nuevo. Esto es puesto que existen demasiadas referencias a las variables de las salidas de una escena, y todas causan error cuando se introduce el nuevo cambio.

Vemos que acá tenemos un caso de un alto nivel de acoplamiento. Para limpiar esto, las desacoplaremos antes de introducir el diccionario.

17.4.1 Usar encapsulamiento para reducir el acoplamiento

La guía del encapsulamiento sugiere que sólo la información del *qué* puede hacer una clase debe ser visible desde el exterior, no *cómo* lo hace. Esto tiene una gran ventaja: si ninguna de las otras clases saben como nuestros datos son almacenados, entonces podemos cambiarlos fácilmente sin romper otras clases.

A diferencia de otros lenguajes, en Python no es posible negar el acceso a los campos de una clase. Es decir, una clase *podría* acceder a nuestros datos si lo intenta. Sin embargo, podemos separar el *qué* y el *cómo* a través de un *accesor*, definiendo así cómo debería comunicarse una clase externa con `Escena`.

```
1  class Escena:
2
3    #... Codigo omitido...
4    def getSalida(self, direccion):
5      if direccion == 'norte':
6        return self.salidaNorte
7      if direccion == 'este':
8        return self.salidaEste
9      if direccion == 'sur':
10       return self.salidaSur
11     if direccion == 'oeste':
12       return self.salidaOeste
13     return None
```

Una vez hecho este cambio en la clase `Escena`, también debemos cambiar la clase `Juego`. Cada vez que se accedía a una variable de salida, ahora debemos usar el método de acceso. Por ejemplo, en vez de escribir:

```
1  sgteEscena = self.escenaActual.salidaEste
```

Ahora debemos escribir:

```
1  sgteEscena = self.escenaActual.getSalida('este')
```

Esto hace que escribir una parte de la clase `Juego` sea más fácil también. El cambio sugerido acá resultará en el siguiente código para el método `irAEscena`:

```
sgteEscena = None
if direccion == 'norte':
  sgteEscena = self.escenaActual.getSalida('norte')
if direccion == 'este':
  sgteEscena = self.escenaActual.getSalida('este')
if direccion == 'sur':
  sgteEscena = self.escenaActual.getSalida('sur')
if direccion == 'oeste':
  sgteEscena = self.escenaActual.getSalida('oeste')
```

En vez de esto, podemos reducir todo este código en una sola línea:

```
sgteEscena = self.escenaActual.getSalida(direccion)
```

Hasta ahora no hemos cambiado la representación de las salidas de la clase `Escena`. Solo hemos limpiado la interfaz. El *cambio* en la clase `Juego` fue mínimo –en vez del acceso a una variable, utilizamos la llamada a un método– pero la ganancia es enorme. Ahora podemos cambiar la manera en que las salidas son almacenadas en una escena, sin la necesidad de preocuparse por romper algo en la clase `Juego`. La representación interna en `Escena` ha sido completamente desacoplada de la interfaz. Ahora que el diseño es como debió haber sido desde el principio, cambiar los campos de las salidas en la clase `Escena` por un diccionario es fácil.

```
class Escena:

  def __init__(self, descripcion):
    self.descripcion = descripcion
    self.salidas = dict()

  # Define las salidas de esta escena. Cada direccion
  # lleva a otra escena o a None (cuando no hay salida)
  def setSalidas(self, norte, este, sur, oeste):
    if norte != None:
      self.salidas['norte'] = norte
    if este != None:
      self.salidas['este'] = este
    if sur != None:
      self.salidas['sur'] = sur
    if oeste != None:
      self.salidas['oeste'] = oeste

  # Retorna la escena a la que se llega si vamos desde
  # esta escena en la direccion "direccion". Si no hay
  # escena en esta direccion retornamos None
  def getSalida(self, direccion):
    if direccion in self.salidas:
      return self.salidas[direccion]
    else:
      return None

  # Retorna la descripcion de la escena (aquella que
  # fue definida en el constructor)
  def getDescription(self):
```

```
31      return self.descripcion
```

Es importante enfatizar nuevamente que podemos hacer este cambio sin siquiera comprobar si algo se romperá en otras partes de la aplicación. Puesto que sólo hemos cambiado aspectos privados de la clase `Escena`, las cuales por definición no pueden ser utilizadas en clases externas, este cambio no impacta en otras clases. La interfaz se mantiene sin cambios.

Una consecuencia de este cambio es que nuestra clase `Escena` es ahora incluso más corta. En vez de listar cuatros variables separadas, ahora tenemos sólo una. Además, el método `getSalida` fue simplificado de manera considerable.

Recordemos que el objetivo de esta serie de cambios era hacer más fácil la tarea de agregar dos posibles nuevas salidas: *arriba* y *abajo*. Esto ya se ha hecho mucho más fácil. Dado que ahora podemos usar el diccionario para almacenar las salidas de una escena, guardar estas dos nuevas direcciones funcionará sin hacer ningún cambio. También podemos obtener la información de la salida nueva a través del método `getSalida` sin ningún problema.

El único lugar en donde la información de la cantidad de salidas existentes (*norte*, *este*, *sur*, *oeste*) aún es parte del código es en el método `setSalidas`. Este es el último lugar que necesita ser mejorado. Por el momento, el contrato del método es de la siguiente manera:

```
1 def setSalidas(self, norte, este, sur, oeste)
```

Este método es parte de la interfaz de la clase `Escena`, por lo que cualquier cambio que hagamos afectará inevitablemente a otras clases debido al acoplamiento. Es importante notar que nunca podremos desacoplar por completo las clases de una aplicación; de otra manera objetos de clases diferentes nunca podrían interactuar entre sí. Más bien, lo que debemos intentar lograr es mantener el grado de acoplamiento lo más bajo posible. Si de todas maneras debemos hacer un cambio a `setSalidas` para acomodar una nueva dirección, nuestra solución preferida es reemplazarlo completamente por este método:

```
1 # Define la salida para esta escena
2 # direccion : la direccion de la salida
3 # vecina : la escena vecina en la direccion dada
4 def setSalidas(self, direccion, vecina):
5   self.salidas[direccion] = vecina
```

Ahora la salida de una escena puede ser asignada de manera individual por cada dirección, y cualquier dirección puede ser usada como una salida. En la clase `Juego`, el cambio que resulta de modificar la interfaz de `Escena` está dada de la siguiente manera. En vez de escribir:

```
1 lab.setSalidas(afuera, oficina, None, None)
```

Ahora escribimos:

```
1 lab.setSalida('norte', afuera)
2 lab.setSalida('este', oficina)
```

Con estos cambios hemos removido por completo la restricción que `Escena` puede contener sólo cuatro salidas. La clase `Escena` puede ahora guardar las salidas en las direcciones *arriba* y *abajo*, de igual manera que puede guardar cualquier otra dirección que se pueda pensar (noreste, sudeste, etc.).

17.5 Diseño basado en responsabilidades

Hemos visto en secciones anteriores que al usar apropiadamente el encapsulamiento podemos reducir el acoplamiento, y así podemos reducir significativamente la cantidad de trabajo necesaria para introducir cambios en una aplicación. Sin embargo, el encapsulamiento no es el único factor que influye en el grado de acoplamiento. Otro de estos aspectos es lo que se conoce con el nombre de *diseño basado en responsabilidades*.

El diseño basado en responsabilidades es el proceso de diseñar clases, asignando responsabilidades bien definidas a cada clase. Este proceso se puede usar para determinar cuál clase debe implementar qué parte de una función. En otras palabras, el diseño basado en responsabilidades expresa la idea que cada clase debe ser responsable de manejar sus propios datos. A menudo, cuando necesitamos agregar nuevas funcionalidades a una aplicación, lo que necesitamos es preguntarnos en qué clase debemos agregar un método para implementar esta nueva funcionalidad. ¿Cuál clase debe ser responsable de esta tarea? La respuesta es que la clase que es responsable de almacenar ciertos datos debe ser también responsable de manipularlos.

Hay que tener presente que la manera en que se usa el diseño basado en responsabilidades influye en el grado de acoplamiento, y así, en la facilidad con la que una aplicación se puede modificar o extender.

17.5.1 Responsabilidades y acoplamiento

Los cambios que introdujimos en la clase `Escena` hacen que ahora sea muy fácil agregar nuevas direcciones para los movimientos *arriba* y *abajo* en la clase `Juego`. Supongamos, por ejemplo, que queremos agregar una nueva escena (sótano) abajo de la oficina. Todo lo que tenemos que hacer es unos cambios menores en el método `crearEscena` de la clase para así crear la escena y dos salidas:

```
1 def crearEscena(self):
2   ...
3   sotano = Escena("en el sotano")
4   ...
5   oficina.setSalida("abajo", sotano)
6   sotano.setSalida("arriba", oficina)
```

Dada la nueva interfaz de la clase `Escena`, lo anterior funciona sin problemas. Así, los cambios resultan ahora muy fáciles de introducir, confirmando que el nuevo diseño es mejor.

Nuestro objetivo de reducir al máximo el acoplamiento implica que los cambios que hagamos en la clase `Escena` no incidan en tener que hacer cambios en la clase `Juego`.

Actualmente, aún tenemos en el código de la clase `Juego` la noción que la información que queremos de una escena consiste en un string de descripción y el string de salida:

```
1 print "Esta en " + escenaActual.getDescripcion()
2 print piezaActual.getStringSalida()
```

¿Qué pasaría si quisiéramos agregar objetos o tesoros en nuestro juego? Cuando describimos lo que vemos, la lista de tesoros, entre otros, toda esta información debe estar incluida en la descripción de la escena. Así, tendríamos que introducir no sólo cambios a la clase `Escena`, sino que también cambiar el código donde se imprime la descripción.

Este problema corresponde a una violación de la regla de diseño basado en responsabilidades. Dado que la clase `Escena` contiene la información de una escena, entonces también debiera ser capaz de dar una descripción de la misma. Esto lo podemos mejorar agregando el siguiente método a la clase:

```
# getDescripcionLarga: -> str
# Retorna la descripcion de una escena
def getDescripcionLarga(self):
  return "Esta en " + self.descripcion + ". Salida hacia: " + \
      self.getStringSalida()
```

Y así, en la clase `Juego` debiéramos escribir:

```
print self.escenaActual.getDescripcionLarga()
```

De esta manera, la *descripción larga* de una escena ahora incluye la descripción estándar y la información sobre las salidas. Además, es capaz de registrar en el futuro si se introducen nuevos cambios a la escena. Cuando hagamos estas extensiones, tendremos que modificar únicamente una clase: `Escena`.

17.6 Cohesión

Anteriormente en este capítulo introdujimos la idea de cohesión: una unidad de código debe ser responsable de una y sólo una tarea. A continuación veremos en más detalle el principio de cohesión y veremos ejemplos. Notemos que el principio de cohesión puede ser aplicado tanto a clases como a métodos.

17.6.1 Cohesión de métodos

Cuando hablamos de *cohesión de métodos*, buscamos expresar la idea que cada método debe ser responsable de una, y solamente una tarea bien definida.

Podemos ver un ejemplo de método cohesivo en la clase `Juego`. Esta clase tiene un método llamado `imprimirBienvenida` para mostrar el mensaje de bienvenida, y este método se llama cada vez que comienza el juego en el método `jugar`.

```
# jugar: ->
# Rutina principal: loop hasta el fin del juego
def jugar(self):
  self.imprimirBienvenida()
  terminado = False
  while not terminado:
    comando = self.parser.getComando()
    terminado = self.procesarComando(comando)
  print "Gracias por jugar. Adios!"

# imprimirBienvenida: ->
# Imprime el mensaje de bienvenida para el jugador
def imprimirBienvenida(self):
  print ""
  print "Bienvenido!"
  print "Escriba 'ayuda' si necesita ayuda."
```

```
17    print ""
18    print self.escenaActual.getDescripcionLarga()
```

Desde un punto de vista funcional, podríamos haber simplemente ingresado los comandos del método `imprimirBienvenida` directamente en el método `jugar` y así obtener exactamente el mismo resultado sin tener que definir un método adicional. Sin embargo, es mucho más fácil de entender y modificar el funcionamiento de un método si se utilizan definiciones cortas y cohesivas. Así, el contar con métodos razonablemente cortos, fáciles de entender, y con nombres que indican claramente el propósito, permiten mantener más fácilmente un programa.

17.6.2 Cohesión de clases

La regla de *cohesión de clases* indica que cada clase debe representar una única entidad bien definida en el dominio del problema.

Supongamos que queremos agregar nuevos elementos al juego. Por ejemplo, cada escena puede contener un elemento, y cada elemento tiene una descripción y un peso. El peso de un elemento puede ser usado más tarde para determinar si se puede recoger o no.

Una primera aproximación podría ser incluir dos campos a la clase `Escena`: `descripcionElemento` y `pesoElemento`. Esto podría funcionar. Luego, podríamos especificar los detalles de cada elemento en cada escena, y podríamos imprimirlos detalles cuando entremos a una escena en particular. Sin embargo, este enfoque no cuenta con un buen grado de cohesión: la clase `Escena` ahora describe una escena y un elemento. También sugiere que un elemento está unido a una escena en particular, lo que no necesariamente es cierto.

Un mejor diseño podría ser crear una clase aparte para los elementos, probablemente de nombre `Elemento`. Esta clase podría tener campos para una descripción y peso, y una escena simplemente tendría una referencia a tal objeto.

Los beneficios concretos de separar escenas y elementos en el diseño se puede ver si cambiamos un poco la especificación. En otra variante del juego, podríamos permitir no solamente tener un único elemento en cada escena, sino contar con un número indefinido de ellos. En el diseño usando una clase `Elemento` aparte, esto es fácil: basta con crear varios objetos y almacenarlos en una colección asociada a una escena. Por el contrario, con el primer enfoque de solución, este cambio sería casi imposible de realizar.

17.6.3 Cohesión para lograr legibilidad

Hay varias maneras en las que una alta cohesión mejora el diseño de software. Las dos más importantes son la *legibilidad* y la *reutilización*. En el ejemplo que vimos anteriormente, la cohesión del método `imprimirBienvenida` es claramente un ejemplo en el que al aumentar la cohesión logramos tener una clase más legible, y por ende, más fácil de entender y mantener.

17.6.4 Cohesión para lograr reutilización

La segunda gran ventaja de la cohesión es un alto potencial para la reutilización. Por citar un ejemplo, cuando creamos una nueva clase `Elemento` para manejar elementos en juego en las escenas, podemos crear múltiples de ellos y usar el mismo código para manejarlos.

La reutilización es también un aspecto importante de la cohesión de métodos. Consideremos un método de la clase `Escena` con la siguiente firma:

```
# dejarEscena: str -> Escena
  def dejarEscena(self, direccion):
  ...
```

Este método podría retornar la escena en una `direccion` dada, de tal manera que pueda ser usada como `self.escenaActual`, y además imprimir la descripción de la nueva escena a la cual el personaje del juego entra. Esta alternativa de diseño es posible, y puede efectivamente funcionar. Sin embargo, nosotros separamos esta tarea en dos métodos:

```
# getSalida: str -> Escena
def getSalida(self, direccion):
  ...

# getDescripcionLarga: -> str
def getDescripcionLarga(self):
  ...
```

El primer método es responsable de retornar la escena siguiente, mietras que el segundo genera un string con la descripción de la escena. La ventaja de este diseño es que las tareas así separadas pueden ser reutilizadas más fácilmente. Por ejemplo, el método `getDescripcionLarga` ahora se utiliza no sólo en el método `irAEscena`, sino que también en `imprimirBienvenida`. Esto es posible porque el método tiene un gran grado de cohesión. En efecto, no habríamos podido reutilizarlo en la versión con el método `dejarEscena`.

17.7 Refactoring

Cuando diseñamos aplicaciones, deberíamos intentar mirar más adelante, anticipar posibles cambios, y crear clases y métodos altamente cohesivos y no fuertemente unidos, de tal manera que podamos realizar modificaciones fácilmente. Si bien es una meta interesante, no siempre podemos anticiparnos a *todas* las posibles adaptaciones futuras, y luego no es factible desarrollar todas las posibles extensiones que quisiéramos. Es por esto que el *refactoring* es importante.

Refactoring (o *refactorización*) es la actividad de reestructurar clases y métodos existentes para adaptarlos a cambios en funcionalidad y requisitos. A menudo en el ciclo de vida de una aplicación, ciertas capas de funcionalidad se van agregando. Un efecto común de esto es que el tamaño de clases y métodos crece lentamente.

Es fácil caer en la tentación de agregar más código a las clases y métodos existentes. Sin embargo, el hacer esto, a menudo, disminuye el grado de cohesión. Cuando se agrega más y más código a una clase o método, es altamente posible que en cierto punto lleguen a representar más de una tarea o entidad claramente definida.

Así, el refactoring consiste en re-pensar y re-diseñar la estructura de clases y métodos. Más comúnmente, el efecto logrado es que las clases se dividen en dos, o que los métodos se dividen en dos o más. El refactoring también puede involucrar (en menor escala) la unión de múltiples clases en una sola, o de varios métodos en uno solo.

17.7.1 Refactoring y testing

Es importante que notemos que cada vez que realizamos refactoring en un programa, usualmente estamos proponiendo realizar cambios potencialmente grandes a algo que ya estaba funcionando correctamente. Como bien ya sabemos, cada vez que modificamos algo que funciona bien, es altamente

posible que se introduzcan errores de manera involuntaria. Luego, resulta crucial proceder con cautela, y antes de refactorizar un bloque de código, *asegurarnos que exista un conjunto de tests para el código actual*.

Una vez que dispongamos de un conjunto de tests, podemos comenzar el proceso de refactoring. Idealmente, involucra dos etapas:

1. El primer paso es refactorizar la estructura interna del código, sin introducir *ningún* cambio a la funcionalidad de la aplicación. En otras palabras, el nuevo programa debe comportarse en tiempo de ejecución exactamente igual que en su versión original. Una vez que hayamos completado esta etapa, debemos probar el nuevo código con el conjunto de tests que disponíamos originalmente, y asegurarnos que pasen todos, sin excepción.

2. La segunda etapa se realiza una vez que hayamos reestablecido la funcionalidad original en el código refactorizado. De esta manera, estamos en una posición segura para poder mejorar la expresividad del código del programa. Una vez que hayamos realizado esta etapa, claramente debemos volver a probar la funcionalidad con un conjunto de tests.

Capítulo 18

Interfaces y Polimorfismo

18.1 ¿Qué es una interfaz?

El concepto de interfaz es un concepto muy importante en la organización de programas. Veremos que el concepto de interfaz toma una dimensión adicional en el caso de la programación orientada al objeto, dado la vinculación con el concepto de polimorfismo, por lo que estudiaremos estos dos conceptos simultáneamente. Por el momento, nos enfocaremos en el concepto de interfaz.

Una interfaz es una especificación de como usar un módulo, sin especificar su implementacíon. Esto está basado en un principio fundamental de cómo organizar programas, que es el principio de separar interfaces de sus implementaciones. Por ejemplo, el módulo `math` de Python tiene una interfaz bien definida, en términos del conjuto de funciones (`math.sqrt(x)`, `math.cos(x)`, etc ...) y de constantes (`math.pi`, ...) que provee. Sin embargo, la documentación del módulo (disponible en `http://docs.python.org/2/library/math.html`), no dice nada sobre cómo esta implementada la función raíz cuadrado, o la función logaritmo, o ninguna de las otras funciones. Hay varias maneras de calcular la raíz cuadrada de un número, pero la interfaz no nos dice cuál en particular se usa. Esto es importante, porque al conocer detalles innecesarios sobre la implementación de un módulo uno puede "vincularse" con este módulo de manera muy fuerte. Como resultado, un cambio en la implementación del módulo puede repercutir en que los usuarios de este módulo tengan que modificar sus programas para adaptarse a dicho cambio.

Los usuarios de un módulo se llaman sus clientes, mientras que el módulo mismo corresponde al proveedor de funcionalidad basado en esta interfaz. En el caso de la programación orientada al objeto, la interfaz de un objeto es el conjunto de métodos que provee, es decir, el conjunto de métodos que uno puede ejecutar sobre el objeto.

18.2 Ejemplo: animales

Si estamos implementando una simulación de una granja, necesitamos una variedad de animales: cabras, vacas, cerdos, gallos, patos, caballos, conejos, etc. Además, cada tipo de animal tiene un comportamiento distinto: cada animal tiene un grito, tiene o no plumas, un número distinto de patas, etc. Implementando un programa orientado al objeto, podemos definir una interfaz general de nuestros animales. Por ejemplo, la siguiente definición de interfaz cubre varios casos:

```
1 # un animal de granja es un objeto que tiene las siguientes operaciones:
2
3 # grito: -> string
```

```
4  # devuelve una representacion textual del sonido que hace un animal
5
6  # plumas: -> bool
7  # devuelve True si el animal tiene plumas
8
9  # pelo: -> bool
10 # devuelve True si el animal tiene pelo
11
12 # patas: -> int
13 # devuelve el numero de patas del animal
14
15 # cola: -> bool
16 # devuelve si el animal tiene una cola o no
17
18 # nombre: -> string
19 # devuelve el nombre de la especie de animal
20
21 # comida: -> string
22 # devuelve el nombre de la comida favorita del animal
```

18.3 ¿Qué es el polimorfismo?

El polimorfismo es una propiedad de la programación orientada al objeto que permite interactuar con objetos con representación distinta de manera uniforme. Es decir, si varios objetos tienen una implementación distintas de la misma interfaz, el cliente debería ser capaz de procesarlos de la misma manera, sin saber que sus implementaciones varían.

Hasta ahora, si queremos procesar los animales de la granja, por ejemplo contar las patas en una lista de animales, podemos hacer algo similar a lo siguiente:

```
1  # contar_patas: list(animales) -> int
2  # cuenta el numero de patas de los animales en la lista
3
4  def contar_patas(animales):
5    patas = 0
6    for animal in animales:
7      if animal.nombre == 'cerdo':
8        patas += 4
9      elif animal.nombre = 'pato':
10       patas += 2
11     ...
12     else: # ultimo caso del conejo
13       patas += 4
14   return patas
```

La programación orientada al objeto nos hace ver el problema de una manera distinta. Como cada objeto es dueño de su comportamiento, cada animal tiene que saber cuántas patas tiene. Es decir, un cerdo sabe que tiene cuatro patas, mientras que un pata sabe que tiene dos patas. Intuitivamente, queremos algo similar a lo siguiente:

```
1  class Cerdo:
```

```
2
3     # ...
4
5     # patas: -> int
6     def patas(self):
7       return 4
8
9   class Pato:
10
11    # ...
12
13    # patas: -> int
14    def patas(self):
15      return 2
16
17  p = Pato()
18  c = Cerdo()
19  assert p.patas() == 2
20  assert c.patas() == 4
```

En Python, este código funciona sin ningún modificación, y es uno de los ejemplos más básicos de polimorfismo: dos objetos pueden responder al mismo mensaje con una implementación distinta del método, en este caso del método `patas`. Dinámicamente (es decir, durante la ejecución del programa), Python determina cuál es la clase a la cual pertenece cada objeto, y ejecuta la implementación del método relevante para dicha clase. De hecho, cada clase definida en Python tiene una tabla de métodos, que contiene los métodos asociado a cada clase. Su estructura es similar a un dicionario asociando nombres de métodos a los métodos actuales (la verdadera implementación de esto es más compleja, por un tema de eficiencia).

18.4 Beneficios del polimorfismo

Valiéndose de este nuevo conocimiento, podemos reescribir el código de la función `contar_patas` usando polimorfismo. Su implementación va a ser mucho más simple:

```
1  # contar_patas: lista(animales) -> int
2  # cuenta el numero de patas de los animales en la lista
3
4  def contar_patas(animales):
5    patas = 0
6    for animal in animales:
7      patas += animal.patas()
8    return patas
```

En esencia, en vez de hacer una cadena de instrucciones `if`/`else` nosotros mismos, dejamos que el intérprete de Python se encargue de esto usando el polimorfismo: cada animal va a ejecutar la versión apropriada del método `patas` según su clase; Python se encarga de buscar el método.

La única restricción es que cada animal *tiene* que tener una implementación de `patas`. Si no, el programa va a producir un error de tipo `AttributeError`.

Esto tiene varios beneficios:

- El código de `contar_patas` se simplifica, y por lo tanto es más abstracto y más entendible (si uno no está interesado en la implementación de `patas`).
- Este código también es más facilmente extensible. Si uno agrega un nuevo animal, no es necesario cambiar la implementación de `contar_patas`, basta con agregar la implementación correcta de `patas` para el nuevo tipo de animales.
- No tenemos que escribir la instrucción condicional, lo que nos permite evitar errores.
- Mientras más uso del método `patas` y más clases implementando `patas` hay en el programa, más importante son todos estos beneficios. Si hay diez usos de `patas`, ahorramos diez cadenas de instrucciones condicionales. Si hay cinco clases implementando patas, ahorramos diez cadenas de cinco instrucciones `if`/`else`, lo que es significativo. Ademas, las posibilidades de insertar errores en el código cuando uno agrega una nueva clase son mayores sin el uso del polimorfismo: es fácil cometer el error de cambiar nueve de las cadenas de instrucciones condicionales, y olvidar cambiar la última...

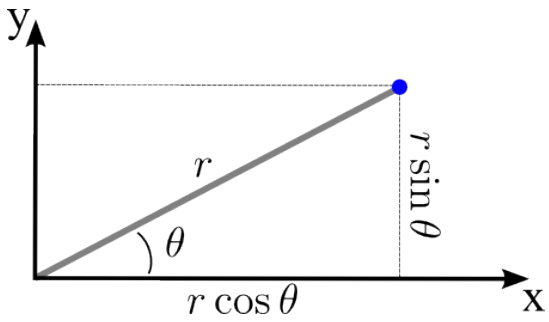
Sin embargo, el polimorfismo no es una solución perfecta. Hay varias consecuencias negativas que hay que tener en cuenta. En particular:

- Si uno está interesado en entender cómo funciona el comportamiento del método `patas`, uno tiene ahora que revisar múltiples ubicaciones en el código fuente, y no solamente una única ubicación como antes.
- Además, si uno quiere extender la interfaz de los animales (agregando el método `grito`, por ejemplo), es necesario modificar cada clase de animales, en ver de hacer el cambio en un único lugar.

18.5 Otro ejemplo: puntos y líneas

Los animales nos permitan ilustrar el polimorfismo de manera simple, pero un lector cuidadoso se habrá dado cuenta que en el caso de los animales el uso del polimorfismo no es tan necesario. Una manera eficiente de resolver el problema es agregar atributos a cada animal, y devolver dicho atributo. Una clase `Animal` general tendría un atributo `numeroDePatas`, un atributo `grito`, etc. Esto tendría beneficios parecidos al polimorfismo, salvo la desventaja que cada objeto tiene más variables, y ocupa más espacio en la memoria.

Sin embargo, en otros casos esta solución simple no es suficiente. Un ejemplo de esto es puntos en dos dimensiones. Hay dos maneras de representar coordenadas: como puntos cartesianos (con attributos `x` y `y`), o como puntos con notación polar (con attributos `angulo` y `distancia`). La figura siguiente nos muestra la diferencia:



El polimorfismo nos permite unificar estos dos tipos de puntos detrás de una misma interfaz. De esta forma, un usuario puede usar puntos cartesiano o polares de la misma forma, y es más fácil cambiar la implementación si es necesario. El código de ambas clases se encuentra a continuación:

```
1  class Cartesian:
2    def __init__(self, x, y):
3      self.__x = x
4      self.__y = y
5
6    # x: -> num
7    def x(self):
8      return self.__x
9
10   # y: -> num
11   def y(self):
12     return self.__y
13
14   # radius: -> num
15   def radius(self):
16     return math.sqrt(self.x() ** 2 + self.y() ** 2)
17
18   # angulo: -> num
19   def angulo(self):
20     return math.atan2(self.y(), self.x())
21
22   # __add__: punto -> punto
23   # suma de puntos
24   def __add__(self, p2):
25     return Cartesian(self.x() + p2.x(), self.y() + p2.y())
26
27   # __sub__: punto -> punto
28   # resta de puntos
29   def __sub__(self, p2):
30     return Cartesian(self.x() - p2.x(), self.y() - p2.y())
31
32   # __str__: -> string
33   # convierte un punto en un string para imprimirlo en la pantalla
34   def __str__(self):
35     return "(x=" + str(self.x()) + ", y=" + str(self.y()) + ")"
36
37 class Polar:
38   def __init__(self, d, a):
```

```
39      self.__distancia = d
40      self.__angulo = a
41
42    # x: -> num
43    def x(self):
44      return self.radius() * math.cos(self.angulo())
45
46    # y: -> num
47    def y(self):
48      return self.radius() * math.sin(self.angulo())
49
50    # radius: -> num
51    def radius(self):
52      return self.__distancia
53
54    # angulo: -> num
55    def angulo(self):
56      return self.__angulo
57
58    # __add__: punto -> punto
59    # suma de puntos
60    def __add__(self, p2):
61      return Cartesian(self.x() + p2.x(), self.y() + p2.y())
62
63    # __sub__: punto -> punto
64    # resta de puntos
65    def __sub__(self, p2):
66      return Cartesian(self.x() - p2.x(), self.y() - p2.y())
67
68    # __str__: -> string
69    # convierte un punto un string para imprimirlo en la pantalla
70    def __str__(self):
71      return "(r=" + str(self.distancia()) + ", theta=" + str(self.angulo()) + ")"
```

Cada clase tiene la misma interfaz, pero los comportamientos en cada casos son muy distintos. Cuando una clase hace un acceso a un atributo, la otra hace un cálculo, pero las interfaces son iguales. Otra clase, la clase `Linea`, puede ocupar cualquier tipo de puntos como atributos, dado que se conforman a la misma interfaz:

```
1   class Linea:
2     def __init__(self, p1, p2):
3       self.__p1 = p1
4       self.__p2 = p2
5
6     def p1(self):
7       return self.__p1
8
9     def p2(self):
10      return self.__p2
11
12    def largo(self):
13      return distancia(self.p1().x(), self.p1().y(), self.p2().x(), self.p2().y())
```

Adémas, veamos dos funcionalidades adicionales de Python: la definición de los metodos __add__ y otras, que permite a Python usar el operador + para los puntos. Hay varios operadores que se pueden definir en forma similar. Esto también es una aplicación del concepto de polimorfismo: todas las clases que tienen el método __add__ se invocan con el operador +, todas las que tienen __str__ se van a imprimir en pantalla como queremos al usar la función `str()`, etc.

```
>>> from puntos import Cartesian
>>> a = Cartesian(1,1)
>>> b = Cartesian(2,2)
>>> c = a + b
>>> c
(x=3, y=3)
```

La segunda funcionalidad es nombrar atributos como __x, lo que permite Python forzar la encapsulación. Tratar de acceder a un atributo con guiones bajos afuera de la clase resulta en una error:

```
>>> from puntos import Cartesian
>>> a = Cartesian(1,1)
>>> a.__x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Cartesian instance has no attribute '__x'
>>>
```

18.6 Reuso de código con delegación y herencia

Si miramos más de cerca el código de las clases para puntos cartesianos y polares, veremos que hay duplicación de código: en ambos casos, los metodos de suma y de resta son iguales. En este ejemplo particular esto no es un problema grande, pero existe la posibilidad que dos clases que tienen la misma interfaz tienen que compartir mucho código, lo que podría resultar en mucho código duplicado. Hay dos estrategias para resolver el problema:

- Crear una clase a parte que contiene el comportamiento común, y delegar el este comportamiento a esta clase.
- Compartir el comportamiento común con herencia.

18.6.1 Delegación

La primera posibilidad es crear una clase que maneje las operaciones de puntos. Por ejemplo, se podría llamar `PointOperator`, y esto es una posible implementación del método de suma. Métodos similares se pueden hacer para implementar el comportamiento compartido.

```
class PointOperator:

  # constructor: punto punto -> PointOperator
  def __init__(self, p1, p2):
    self.p1 = p1
    self.p2 = p2
```

```
7
8     # add: -> punto
9     def add(self):
10      return Cartesian(self.p1.x() + self.p2.x(), self.p1.y() + self.p2.y())
11
12    # ...
```

En seguida, podemos remplazar el codigo duplicado por un uso de la funcionalidad de `PointOperator`:

```
1  class Cartesian:
2    # ...
3
4    def __add__(self, p2):
5      return PointOperator(self, p2).add()
6
7  class Polar:
8    # ...
9
10   def __add__(self, p2):
11     return PointOperator(self, p2).add()
```

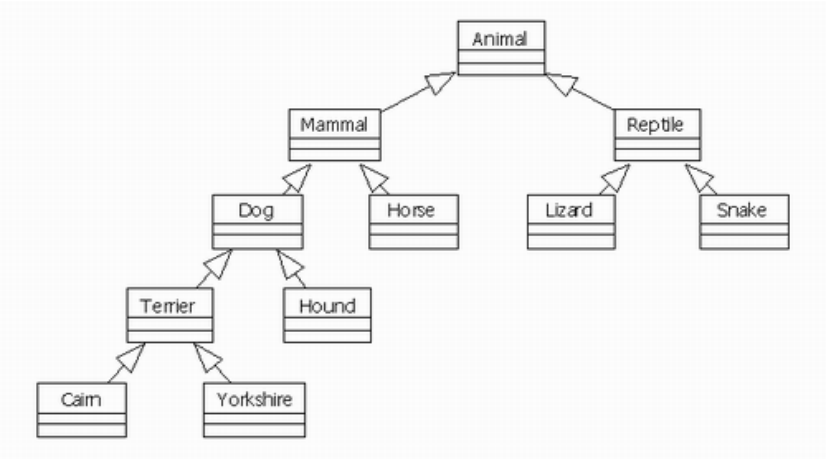
Una estrategia siempre exitosa es dividir las clases en entidades más pequeñas que se comunican. La delegación hace uso de este principio: el comportamiento común se traslada a una nueva clase, y ambas clases hacen uso de esta clase cuando necesitan el comportamiento común. Esta estrategia permite reutilizar el código duplicado a bajo costo. Notar que todavía queda un poco de duplicación de código, pero mucho menos que en la implementación original.

18.6.2 Herencia

La otra alternativa es definir una clase más general que contiene el comportamiento común, y ver las clases de puntos cartesianos y polares como casos especificos de esta clase general. Muchos lenguajes orientados al objeto (incluyendo Python) permiten hacer esto. En este caso, sería natural tener una clase general de puntos, y casos especificos de puntos cartesianos y polares.

El mecanismo de *herencia* nos permite hacer eso. La clase para punto va a ser llamada la superclase, y las clases de puntos cartesiano y polares van a ser las subclases, que hereden de la clase punto. El conjunto de clases forma una jerarquía de clases, con las clases las más generales arriba y las clases las más específicas abajo.

Otro ejemplo de herencia es hacer una jerarquía de animales, como la siguiente:



Notar que la herencia permite describir fácilmente estructuras de tipo árbol, es decir, cada clase tiene una superclase. Tener más de una superclase a la vez es algo mucho más complejo. En dichos casos, es mejor usar la composición y delegación.

Concretamente, ¿que nos permite hacer la herencia? Nos permite reutilizar código de manera sencilla: una subclase hereda todos los datos y el comportamiento definido en la superclase. Si una clase no define para sí misma un método, se busca este método en la superclase, si lo tiene. De esta manera, podemos definir los métodos de suma y resta en la clase punto, y agregar en este clase cualquier otro comportamiento compartido (por ejemplo, si los puntos tienen un color, este aspecto es independiente del hecho que sean cartesianos o polares). El codígo Python ocupando herencia es el siguiente:

```
1  class Point:
2
3    def __add__(p2):
4       return Cartesian(self.p1.x() + self.p2.x(), self.p1.y() + self.p2.y())
5
6  class Cartesian(Point):
7    # no define __add__
8
9  class Polar(Point):
10   # no define __add__
```

En caso que sea necesario, se puede llamar al constructor de la superclase, o cualquier otro método, de la siguiente manera:

```
1  class Point:
2     def __init__(self):
3        # codigo importante
4
5  class Cartesian(Point):
6    def __init__(self, x, y):
7       Point.__init__(self)
8       self.x = x
9       self.y = y
```

Finalmente, uno tiene que ser precavido al usar herencia. Esto es por varias razones. La primera es que la herencia es un mecanismo más complejo que la delegación, por los posibles problemas de

superclase, etc. La segunda es que es sólo es posible heredar de una superclase, mientras que una clase puede delegar a varias clases. Finalmente, y lo más importante, la herencia tiene un sentido muy específico.

Consideremos el ejemplo de las ruedas de una bicicleta. Uno podria pensar que una bicicleta herede de una rueda, para reutilizar su compartamiento, pero esto sería muy incorrecto. Para saber si una clase A hereda de otra clase B, hay que preguntarse: ¿Será un objeto de tipo A también un objeto de tipo B? En este ejemplo, ¿será una bicicleta una rueda? Absolutamente no. Una bicicleta *tiene* una rueda, *no es* una rueda. Esto nos muestra que la herencia en este caso es el mecanismo equivocado. Lo que en verdad queremos es la composicíon de objetos: una bicicleta contiene un atributo de tipo rueda. De hecho, la bicicleta tendría dos atributos de tipo rueda, algo que no podemos hacer con herencia.