Estructuras

Juan Álvarez Rubio

jalvarez@dcc.uchile.cl,

Todos los programas y funciones que se han escrito hasta ahora han operado con valores individuales (“escalares”) que se representan y almacenan en variables. Sin embargo, muchos problemas necesitan utilizar datos compuestos por una cantidad fija o variable de escalares. Por ejemplo, una fracción y una fecha son datos compuestos por dos y tres valores respectivamente.

Módulo Estructura

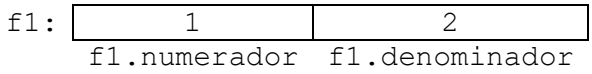
Con el propósito de facilitar el manejo de datos con más de un valor se ha complementado el lenguaje Python con un módulo que permite manejar datos compuestos. La función crear permite definir un tipo de datos para mantener más de un valor. Por ejemplo, las siguientes instrucciones permiten definir un tipo para operar con fracciones:

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

La función crear recibe dos parámetros de tipo str. El primer parámetro es el nombre que se dará al tipo de datos. El segundo parámetro contiene los nombres que tendrán las componentes de las variables de ese tipo separadas por espacios. Una vez definido un tipo, se pueden crear variables de ese tipo. Por ejemplo, para definir una variable del tipo fraccion se necesita la siguiente instrucción:

```
f1=fraccion(1,2)
```

La representación de la variable f1 se puede ilustrar de la siguiente manera:



Las componentes de una variable se pueden recuperar con la notación variable.componente. Por ejemplo, las siguientes instrucciones calculan y escriben una fracción f3 con el producto de las fracciones f1 y f2:

```
f2=fraccion(3,4)
f3=fraccion(f1.numerador*f2.numerador,f1.denominador*f2.denominador)
print("producto:",f3.numerador, "/",f3.denominador)
```

La componente de una variable se puede usar en todo contexto en que se permite una variable, salvo a la izquierda de una asignación, es decir, no se puede modificar. Por ejemplo, la siguiente instrucción de asignación no se puede efectuar y produce un error:

```
f1.numerador = f1.numerador*f2.numerador  #error
```

Módulo Fracción

La definición de un tipo de datos y las operaciones que pueden realizarse con operandos de ese tipo es conveniente agruparlas en un módulo. Al respecto, se recomienda que el módulo comience especificando el nombre del tipo y los nombres y los tipos de sus componentes en la forma:

#nombre: nombre(tipo) nombre(tipo) …

Por ejemplo, el tipo fracción, que se utiliza para representar el numerador y denominador de una fracción, se especifica y define de la siguiente manera:

```
#fraccion: numerador(int) denominador(int)
import estructura
estructura.crear("fraccion","numerador denominador")
```

Es conveniente agregar en el módulo una función que compruebe que una fracción esté bien definida: numerador y denominador de tipo int y denominador distinto de cero.

```
#esFraccion: fraccion -> bool
#True si x es una fraccion valida
#ej: esFraccion(fraccion(1,2)) -> True
#ej: esFraccion(fraccion(1,0)) -> False
def esFraccion(x):
    return type(x)==fraccion \
       and type(x.numerador)==int \
       and type(x.denominador)==int \
       and x.denominador != 0
assert esFraccion(fraccion(1,2))
assert not esFraccion(fraccion(1,0))
```

La función esFraccion debe ser utilizada en el resto de las funciones del módulo de manera de garantizar que las fracciones estén bien definidas. A continuación, el módulo debe contener funciones para cada una de las operaciones habituales con fracciones: operaciones aritméticas, simplificación, comparación, lectura y escritura.

Operaciones aritméticas entre fracciones

El módulo debe contener funciones para realizar las cuatro operaciones aritméticas entre fracciones: suma, resta, multiplicación y división. Las funciones deben definirse con su receta de diseño y se prueban con las siguientes fracciones:

```
f1=fraccion(1,2)
f2=fraccion(3,4)
```

```
#sumar: fraccion fraccion -> fraccion
# x + y
#ej: sumar(f1,f2)->fraccion(10,8)
def sumar(x,y):
  assert esFraccion(x)
  assert esFraccion(y)
  a = x.numerador   * y.denominador + x.denominador * y.numerador
  b = x.denominador * y.denominador
  return fraccion(a,b)
assert sumar(f1,f2)==fraccion(10,8)

#restar: fraccion fraccion -> fraccion
# x - y
#ej: restar(f1,f2)->fraccion(-2,8)
def restar(x,y):
  assert esFraccion(x)
  assert esFraccion(y)
  a = x.numerador   * y.denominador - x.denominador * y.numerador
  b = x.denominador * y.denominador
  return fraccion(a,b)
assert restar(f1,f2)==fraccion(-2,8)

#multiplicar: fraccion fraccion -> fraccion
# x * y
#ej: multiplicar(f1,f2)->fraccion(3,8)
def multiplicar(x,y):
  assert esFraccion(x)
  assert esFraccion(y)
  a = x.numerador   * y.numerador
  b = x.denominador * y.denominador
  return fraccion(a,b)
assert multiplicar(f1,f2)==fraccion(3,8)

#dividir: fraccion fraccion -> fraccion
# x / y
#ej: dividir(f1,f2)->fraccion(4,6)
def dividir(x,y):
  assert esFraccion(x)
  assert esFraccion(y)
  a = x.numerador   * y.denominador
  b = x.denominador * y.numerador
  assert b != 0
  return fraccion(a,b)
assert dividir(f1,f2)== fraccion(4,6)
```

Simplificación de fracciones

Para simplificar una fracción se define una función que recibe un fracción y entrega otra simplificada, sin modificar la fracción original.

```
from mcd import *
#simplificar: fraccion -> fraccion
#fraccion con valor de fraccion x simplificada
#ej: simplificar(fraccion(2,4)) -> fraccion(1,2)
def simplificar(x):
  assert esFraccion(x)
  m=mcd(abs(x.numerador), abs(x.denominador))#maximo común divisor
  return fraccion(x.numerador//m, x.denominador//m)
assert simplificar(fraccion(2,4)) == fraccion(1,2)
```

Comparación de fracciones

Las variables de tipo fraccion se puede comparar con los operadores de comparación o relación. Por ejemplo, fraccion(1,2)==fraccion(1,2) devuelve True y fraccion(1,2)==fraccion(1,3) devuelve False. Pero fraccion(1,2)==fraccion(2,4) devuelve False. Por lo tanto, es necesario definir una función:

```
#esIgual: fraccion fraccion -> bool
#True si x es igual a y
#ej: esIgual(fraccion(1,2),fraccion(2,4)) -> True
#    esIgual(fraccion(1,2),fraccion(1,3)) -> False
def esIgual(x,y):
    assert esFraccion(x)
    assert esFraccion(y)
    return simplificar(x)==simplificar(y)
assert esIgual(fraccion(1,2),fraccion(2,4))
assert not esIgual(fraccion(1,2),fraccion(1,3))
```

Por otra parte, fraccion(1,2)>fraccion(1,3) devuelve False puesto que los operadores predefinidos solo comparan la representación interna componente a componente, y, como 2 es menor que 3, considera que la primera fracción es menor que la segunda. Luego, es necesario definir una función especial para comparar fracciones y que devuelva uno de tres resultados:

```
#comparar: fraccion fraccion -> int
#0 si x=y, n°<0 si x<y, n°>0 si x>y
#ej: comparar(fraccion(1,2),fraccion(1,3)) -> n°>0
#ej: comparar(fraccion(1,3),fraccion(1,2)) -> n°<0
#ej: comparar(fraccion(1,2),fraccion(2,4)) -> 0
def comparar(x,y):
  assert esFraccion(x)
  assert esFraccion(y)
```

```
  return x.numerador * y.denominador - x.denominador * y.numerador
assert comparar(fraccion(1,2),fraccion(1,3)) > 0
assert comparar(fraccion(1,3),fraccion(1,2)) < 0
assert comparar(fraccion(1,2),fraccion(2,4)) == 0
```

La resta de los productos devuelve un número 0, menor que 0, o mayor que 0 dependiendo de la relación entre la primera y la segunda fracción.

Lectura y escritura de fracciones

El módulo fracción puede contener una función para leer una fracción de manera de facilitar esa tarea a los programas usuarios.

```
#leer: str str -> fraccion
#fraccion con numerador y denominador q se leen con preguntas x e y
#ej: leer("a?","b?")->fraccion(1,2) si lee los números 1 y 2
def leer(x="numerador?",y="denominador?"):
    assert type(x)==str
    assert type(y)==str
    a=int(input(x))
    b=int(input(y))
    assert b != 0
    return fraccion(a,b)
#assert leer("a?","b?")==fraccion(1,2)
```

Los parámetros x e y se definen con valores por omisión, de modo que, si se invocan sin argumentos, aparecerán las preguntas indicadas para incentivar el ingreso del numerador y denominador de una fracción. Por otra parte, la función se asegura de devolver una fracción correcta impidiendo un denominador cero. El comentario final indica que la instrucción assert fue utilizada en la etapa de prueba del módulo, pero no se debe ejecutar cuando el módulo se use, puesto que solicitaría ingresar dos números para el numerador y denominador de una fracción.

La escritura de una fracción se facilita al disponer de la siguiente función:

```
#escribir: str fraccion ->
#escribe x seguido de  numerador / denominador de y
#ej: escribir("f:",fraccion(1,2)) escribe "f: 1 / 2"
def escribir(x,y):
    assert type(x)==str
    assert esFraccion(y)
    print(x, y.numerador, "/", y.denominador)
#escribir("f:",fraccion(1/2))
```

El comentario final sugiere que la invocación a la función escribir fue utilizada sólo en la etapa de prueba del módulo, y sin utilizar assert porque la función no devuelve un resultado.

Uso del módulo para fracciones

Las funciones del módulo se pueden utilizar en un programa que lee dos fracciones y escribe los resultados simplificados de acuerdo al siguiente diálogo:

```
operaciones con fracciones a/b y c/d
a?1
b?2
c?3
d?4
suma: 5 / 4
resta: -1 / 4
producto: 3 / 8
división: 2 / 3
mayor: 3 / 4
```

El programa que implementa el diálogo anterior contiene las siguientes instrucciones:

```
from fraccion import *
print("operaciones con fracciones a/b y c/d")
f1=leer("a?","b?")
f2=leer("c?","d?")
escribir("suma:", simplificar(sumar(f1,f2)))
escribir("resta:", simplificar(restar(f1,f2)))
escribir("producto:", simplificar(multiplicar(f1,f2)))
escribir("división:", simplificar(dividir(f1,f2)))
if comparar(f1,f2)>0:
   escribir("mayor:", f1)
else:
   escribir("mayor:", f2)
```

El programa consiste casi exclusivamente de invocaciones a funciones del módulo fraccion. Esta situación caracteriza al paradigma de programación funcional en que los problemas se resuelven en base al uso de funciones, por ejemplo sumar(f1,f2), y de composición de funciones, por ejemplo, escribir(…,simplificar(sumar(f1,f2))). Esta característica será más evidente al procesar listas de valores.

Los nombres de las funciones son todos infinitivos de verbos, lo que resulta fácil de recordar y deducir. Los nombres de las funciones podrían ser sustantivos, puesto que representan valores. En ese caso sería más difícil elegir nombres apropiados e intuitivos: suma o adición; resta, sustracción o diferencia; multiplicación o producto; división o cuociente; fraccionSimple o fraccionSimplificada; comparación y lectura. En el caso de la función escribir, el nombre escritura sería cuestionable, puesto que no representa un valor, dado que la función no devuelve un resultado.