Abstracción de datos

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Introducción

1.1. Introducción

Hemos visto que una buena modularidad se apoya en tres conceptos:

- **Abstracción**: los usuarios de un módulo necesitan saber qué hace pero no cómo lo hace.

- **Ocultación de información**: los módulos deben ocultar sus decisiones de diseño a sus usuarios.
- **Independencia funcional**: los módulos deben dedicarse a alcanzar un único objetivo, con una alta cohesión entre sus elementos y un bajo acoplamiento con el resto de los módulos.

Vamos a estudiar con más detalle el primero de ellos: la abstracción.

Hasta ahora hemos estudiado la abstracción como un proceso mental que ayuda a estudiar y manipular sistemas complejos *destacando* los detalles relevantes e *ignorando* momentáneamente los demás que ahora mismo no tienen importancia o no son necesarias.

Asimismo, hemos visto que la abstracción se define por niveles, es decir, que cuando estudiamos un sistema a un determinado nivel:

- Se *destacan* los detalles relevantes en ese nivel.
- Se *ignoran* los detalles irrelevantes en ese nivel. Si descendemos de nivel de abstracción, es probable que algunos de esos detalles pasen a ser relevantes.

Los programas son sistemas complejos, así que resulta importante que el lenguaje de programación nos permita estudiar y diseñar programas mediante sucesivos niveles de abstracción.

La abstracción es un proceso pero también es algo que puede formar parte de un programa.

Hasta ahora, las únicas abstracciones que hemos utilizado y creado son las **funciones**, también llamadas **abstracciones funcionales**.

Una función es una abstracción funcional porque el usuario de la función sólo necesita conocer la **especificación** de la abstracción (el *qué* hace) y puede ignorar el resto de los detalles de **implementación** que se encuentran en el cuerpo de la función (el *cómo* lo hace).

Por eso decimos que las funciones definen dos niveles de abstracción.

En otras palabras, al diseñar una función estamos creando una abstracción que separa la forma en la que se utiliza la función de la forma en como está implementada esa función.

Las abstracciones funcionales son un mecanismo que nos permite:

1. componer una **operación compleja** combinando otras operaciones más simples (dándole un nuevo nombre a todo el conjunto), y
2. poder usar esa nueva operación compleja sin necesidad de conocer cómo está hecha por dentro (es decir, sin necesidad de conocer cuáles son esas operaciones más simples que la forman, que son detalles que quedan ocultos al usuario).

Una vez que la función se ha diseñado y se está utilizando, se puede sustituir por cualquier otra que tenga el mismo comportamiento observable.

De forma similar, los datos compuestos o estructurados son un mecanismo que nos permite crear un **dato complejo** combinando otros datos más simples, formando una única unidad conceptual.

Pero, por desgracia, estos datos compuestos **no ocultan sus detalles de implementación al usuario**, sino que éste tiene que conocer cómo está construido.

Por ejemplo, podemos representar un número racional $\frac{a}{b}$ mediante una pareja de números enteros a y b (su numerador y su denominador).

Si almacenamos los dos números por separado no estaremos creando una sola unidad conceptual (no estaremos componiendo un nuevo dato a partir de otros datos más simples).

Eso no resulta conveniente. A nosotros, como programadores, nos interesa que el numerador y el denominador de un número racional estén juntos formando una sola cosa, un nuevo valor: un número racional.

Así que podríamos representar dicha pareja de números usando una lista como `[a, b]`, o una tupla `(a, b)`, o incluso un diccionario `{'numer': a, 'denom': b}`.

Pero estaríamos obligando al usuario de nuestros números racionales a tener que saber cómo representamos los racionales en función de otros tipos más primitivos, lo que nos impide cambiar luego esa representación sin afectar al resto del programa.

Es decir: les estamos obligando a conocer detalles de implementación de nuestros números racionales.

Por ejemplo, si representamos un racional como dos números enteros separados, no podríamos escribir una función que multiplique dos racionales $\frac{n_1}{d_1}$ y $\frac{n_2}{d_2}$ ya que dicha función tendría que devolver dos valores, el numerador y el denominador del resultado:

```
def mult_rac(n1, d1, n2, d2):
    return ????
```

Si representamos un racional $\frac{n}{d}$ con, por ejemplo, una tupla `(n, d)`, la función que multiplica dos racionales podría ser:

```
def mult_rac(r1, r2):
    return (r1[0] * r2[0], r1[1] * r2[1])
```

Es decir, que la función tendría que saber que el racional se representa internamente con una tupla, y que el numerador es el primer elemento y que el denominador es el segundo elemento.

Nos interesa que nuestro programa sea capaz de expresar el concepto de «número racional» y que pueda manipular números racionales como valores con entidad propia y definida, no simplemente como parejas de números enteros, independientemente de su representación interna.

Para todo esto, es importante que el programa que utilice los números racionales no necesite conocer los detalles internos de cómo está representado internamente un número racional.

Es decir: que los números racionales se pueden representar internamente como una lista de dos números, o como una tupla, o como un diccionario, o de cualquier otro modo, pero ese detalle interno debe quedar oculto para los usuarios de los números racionales.

La técnica general de aislar las partes de un programa que se ocupan de *cómo se representan* los datos de las partes que se ocupan de *cómo se usan* los datos es una poderosa metodología de diseño llamada **abstracción de datos**.

La **abstracción de datos** es una **técnica** pero también es algo que puede formar parte de un programa.

Diseñar programas usando abstracción de datos da como resultado la creación y utilización de **tipos abstractos de datos** (o **TAD**), a los que también se se les denomina **abstracciones de datos**.

La abstracción de datos se parece a la abstracción funcional:

- Cuando creamos una **abstracción funcional**, se ocultan los detalles de cómo se implementa una función, y esa función particular se puede sustituir luego por cualquier otra función que tenga el mismo comportamiento general sin que los usuarios de la función se vean afectados.

  En otras palabras, podemos hacer una abstracción que separe la forma en que se utiliza la función de los detalles de cómo se implementa la función.

- Igualmente, la **abstracción de datos** separa el uso de un dato compuesto de los detalles de cómo está construido ese dato compuesto, que quedan ocultos para los usuarios de la abstracción de datos.

  Para usar una abstracción de datos no necesitamos conocer sus detalles internos de implementación.

Elementos del lenguaje	Instrucciones	Datos
Primitivas	Definiciones, literales y sentencias simples	Datos simples (enteros, reales, booleanos…
Mecanismos de combinación	Expresiones y sentencias compuestas (estructuras de control)	Datos compuestos (listas, tuplas…)
Mecanismos de abstracción	Abstracciones funcionales (funciones)	Abstracciones de datos (tipos abstractos)

El concepto de **abstracción de datos** (o **tipo abstracto de datos**) fue propuesto por John Guttag en 1974 y dice que:

Tipo abstracto de datos

Un **tipo abstracto de datos** (o ***abstracción de datos***) es un conjunto de valores y de operaciones que se definen mediante una **especificación** que es independiente de cualquier representación.

Para ello, los tipos abstractos de datos se definen nombrando, no sus valores, sino sus **operaciones** y las propiedades que cumplen éstas.

Los **valores** de un tipo abstracto se definen también como operaciones.

Por ejemplo, `set` es un tipo primitivo en Python que actúa como un tipo abstracto de datos.

- Se nos proporcionan **operaciones primitivas** para crear conjuntos y manipular conjuntos (unión, intersección, etc.) y también un modo de visualizar sus valores.
- Pero no sabemos, ni necesitamos saber, cómo se representan internamente los conjuntos en la memoria del ordenador. Ese es un detalle interno del intérprete.

En general, el programador que usa un tipo abstracto puede no saber, e incluso se le impide saber, cómo se representan los elementos del tipo de datos.

Esas **barreras de abstracción** son muy útiles porque permiten cambiar la representación interna sin afectar a las demás partes del programa que utilizan dicho tipo abstracto.

En resumen, tenemos que un tipo abstracto debe cumplir:

- **Privacidad de la representación**: los usuarios no conocen la representación de los valores del tipo abstracto en la memoria del ordenador.
- **Protección**: sólo se pueden utilizar para el nuevo tipo las operaciones previstas en la especificación.

  «Son las especificaciones, y no los programas, los que realmente describen una abstracción; los programas simplemente la implementan.»

  – Barbara Liskov

El programador de un tipo abstracto debe crear, por tanto, dos partes bien diferenciadas:

- La **especificación** del tipo: única parte que conoce el usuario del mismo y que consiste en:
  * El **nombre** del tipo.
  * La especificación de las **operaciones** permitidas. Esta especificación tendrá:
    · Una parte **sintáctica**: la *signatura* de cada operación.
    · Otra parte **semántica**: que define las **propiedades** que deben cumplir dichas operaciones y que se pueden expresar mediante **ecuaciones** o directamente en lenguaje natural.
- La **implementación** del tipo: conocida sólo por el programador del mismo y que consiste en la *representación* del tipo por medio de otros tipos y en la implementación de las operaciones.

2. Especificaciones

2.1. Sintaxis

La sintaxis de una **especificación algebraica** es la siguiente:

```
espec ⟨tipo⟩
    [parámetros
        ⟨parámetro⟩+]
    operaciones
        (⟨operación⟩ : ⟨signatura⟩)+
    [var
        ⟨decl_var⟩ (; ⟨decl_var⟩)*]
    ecuaciones
        ⟨ecuación⟩+
```

Donde:

```
⟨decl_var⟩ ::= ⟨variable⟩ : ⟨tipo⟩
⟨ecuación⟩ ::= ⟨izquierda⟩ ≐ ⟨derecha⟩
```

2.2. Operaciones

Las operaciones que forman parte de la especificación de un tipo abstracto *T* pueden clasificarse en:

- **Constructoras**: operaciones que devuelven un valor de tipo *T*.
  - A su vez, las constructoras se dividen en:
    - **Generadoras**: el conjunto de operaciones generadoras está formado por aquellas operaciones constructoras que tienen la propiedad de que sólo con ellas es suficiente para generar cualquier valor del tipo, y excluyendo cualquiera de ellas hay valores que no pueden ser generados.
    - **Modificadoras**: son las operaciones constructoras que no forman parte del conjunto de las generadoras.
- **Selectoras**: operaciones que toman como argumento uno o más valores de tipo *T* que no devuelven un valor de tipo *T*.

2.3. Ejemplos

Un ejemplo de especificación de las **listas** como tipo abstracto sería:

espec *lista*
parámetros
elemento
operaciones
$[\,] : \rightarrow lista$
$[\_] : elemento \rightarrow lista$
$\_ \texttt{++} \_ : lista \times lista \rightarrow lista$
$\_ \texttt{:} \_ : elemento \times lista \rightarrow lista$
$\texttt{len} : lista \rightarrow \mathbb{N}$
var
$x : elemento; l, l_1, l_2, l_3 : lista$
ecuaciones
$x \texttt{ : } l \doteq [x] \texttt{ ++ } l$
$l \texttt{ ++ } [\,] \doteq l$
$[\,] \texttt{ ++ } l \doteq l$
$(l_1 \texttt{ ++ } l_2) \texttt{ ++ } l_3 \doteq l_1 \texttt{ ++ } (l_2 \texttt{ ++ } l_3)$
$\texttt{len}([\,]) \doteq 0$
$\texttt{len}([x]) \doteq 1$
$\texttt{len}(x \texttt{ : } l) \doteq 1 + \texttt{len}(l)$
$\texttt{len}(l_1 \texttt{ ++ } l_2) \doteq \texttt{len}(l_1) + \texttt{len}(l_2)$

La ecuación $t_1 \doteq t_2$ significa que el valor construido mediante t_1 es *el mismo* que el construido mediante t_2.

Este estilo de especificación se denomina **especificación algebraica**.

Su principal virtud es que permite definir un nuevo tipo de forma *totalmente independiente* de cualquier posible representación o implementación.

¿A qué categoría pertenencen cada una de esas operaciones?

Al definir y usar abstracciones de datos, seguiremos el enfoque propio de la programación funcional, donde:

- No existe mutabilidad ni estado interno ni cambios de estado.
- Se «simula» el cambio de estado de un dato compuesto creando un nuevo dato con los cambios aplicados.

Por ejemplo: en programación funcional, la operación `:` (que añade un elemento al principio de una lista) realmente no modifica dicha lista sino que crea una nueva lista con el elemento situado al principio, y la devuelve.

Esto lo podemos expresar en la especificación de la lista con la siguiente ecuación (siendo x un elemento y l una lista):

x `:` $l \doteq$ `[`x`]` `++` l

Se puede observar que no se modifica la lista l para añadir el elemento x al principio, sino que se crea una lista nueva por concatenación de otras dos.

¿Crees que hay ecuaciones que sobran? ¿Se podría crear una especificación equivalente pero más corta, de forma que tenga menos ecuaciones pero que se comporte exactamente igual que la anterior?

En realidad, en la especificación anterior hay ecuaciones que no son estrictamente necesarias ya que se pueden deducir a partir de otras que sí lo son:

- Las ecuaciones que son necesarias representan **axiomas** de nuestro sistema.
- Las ecuaciones que se pueden deducir de otras representan **teoremas** de nuestro sistema.

Bastaría con tener una especificación algebraica basada en axiomas y que dependan únicamente de operaciones generadoras.

espec *lista*
  parámetros
    elemento
  operaciones
    `[]` : $\rightarrow$ *lista*
    `_:_` : *elemento* $\times$ *lista* $\rightarrow$ *lista*
    `[_]` : *elemento* $\rightarrow$ *lista*
    `_++_` : *lista* $\times$ *lista* $\rightarrow$ *lista*
    `len` : *lista* $\rightarrow \mathbb{N}$
  var
    x : *elemento*; l, l_1, l_2 : *lista*
  ecuaciones
    `[`x`]` $\doteq$ x `:` `[]`
    `[]` `++` $l \doteq l$
    (x `:` l_1) `++` $l_2 \doteq x$ `:` (l_1 `++` l_2)
    `len`(`[]`) $\doteq 0$
    `len`(x `:` l) $\doteq 1 +$ `len`(l)

En el ejemplo de las listas, las **operaciones generadoras** son `[]` y `_:_`.

El resto de las operaciones se definen a partir de ellas.

Las ecuaciones que aparecen ahora en esta especificación son los **axiomas** de la misma, y las que hemos quitado son **teoremas** que se pueden deducir (*demostrar*) a partir de otras ecuaciones.

Por ejemplo, supongamos que queremos demostrar que se cumple la siguiente ecuación (que apareció antes en la primera especificación de *lista*):

l `++` `[]` $\doteq l$

siendo l una lista cualquiera.

Se puede demostrar por inducción sobre l, a partir de los axiomas de la especificación de *lista*:

- Caso l = `[]`:

  l `++` `[]` $\doteq$ {por definición de l}

  `[]` `++` `[]` $\doteq$ {por el primer axioma de `++`}

  `[]` $\doteq$ {por definición de l}

  l

- Caso $l = x$ `:` l_1:

  Suponemos que la propiedad se cumple para l_1 (hipótesis inductiva), es decir, que l_1 `++` `[]` $\doteq l_1$.

  l `++` `[]` $\doteq$ {por definición de l}

  (x `:` l_1) `++` `[]` $\doteq$ {por el segundo axioma de `++`}

  x `:` (l_1 `++` `[]`) $\doteq$ {por hipótesis inductiva}

  x `:` l_1 $\doteq$ {por definición de l}

  l

Como hemos demostrado que l `++` `[]` $\doteq l$ para cualquiera de las dos formas posibles que puede tener l, hemos logrado demostrar la propiedad para cualquier valor de l.

Las **pilas** como tipo abstracto se podrían especificar así:

espec *pila*
  parámetros
    elemento
  operaciones
    `pvacia` : $\rightarrow$ *pila*
    `apilar` : *pila* $\times$ *elemento* $\rightarrow$ *pila*
    parcial `cima` : *pila* $\rightarrow$ *elemento*
    parcial `desapilar` : *pila* $\rightarrow$ *pila*
    `vacia?` : *pila* $\rightarrow \mathfrak{B}$
  var
    p : *pila*; x : *elemento*
  ecuaciones
    `cima`(`apilar`(p, x)) $\doteq x$
    `desapilar`(`apilar`(p, x)) $\doteq p$
    `vacia?`(`pvacia`) $\doteq V$
    `vacia?`(`apilar`(p, x)) $\doteq F$
    `cima`(`pvacia`) $\doteq$ *error*

$\texttt{desapilar}(\texttt{pvacia}) \doteq \mathit{error}$

En esta especificación aparecen **operaciones parciales**, que son aquellas que no se pueden aplicar a cualquier operando.

Por ejemplo, no se puede (no tiene sentido) calcular la cima de una pila vacía o desapilar una pila vacía. En ambos casos obtenemos un error.

Por tanto, ambas operaciones son *parciales*, porque no se pueden aplicar sobre cualquier tipo de pila (sólo se puede sobre las *no vacías*).

Un programa que hiciera uso de las pilas, una vez implementado el tipo abstracto de datos, podría ser:

```
p = pvacia()        # crea una pila vacía
p = apilar(p, 4)    # apila el valor 4 en la pila p
p = apilar(p, 8)    # apila el valor 8 en la pila p
print(vacia(p))     # imprime False
print(cima(p))      # imprime 8
p = desapilar(p)    # desapila el valor 8 de la pila p
print(cima(p))      # imprime 4
p = desapilar(p)    # desapila el valor 4 de la pila p
print(vacia(p))     # imprime True
print(cima(pvacia)) # error
```

El programa usa la pila a través de las operaciones sin necesidad de conocer su representación interna (su implementación).

Es decir: no necesitamos saber cómo está hecha la pila por dentro, ni cómo están programadas las funciones `pvacia`, `apilar`, `cima` y `desapilar`.

Nos basta con saber las propiedades que deben cumplir esas funciones, y eso viene definido en la especificación.

Los **números racionales** se podrían especificar así:

espec *rac*

operaciones

parcial $\texttt{racional} : \mathbb{Z} \times \mathbb{Z} \to \mathit{rac}$
$\texttt{numer} : \mathit{rac} \to \mathbb{Z}$
$\texttt{denom} : \mathit{rac} \to \mathbb{Z}$
$\texttt{suma} : \mathit{rac} \times \mathit{rac} \to \mathit{rac}$
$\texttt{mult} : \mathit{rac} \times \mathit{rac} \to \mathit{rac}$
$\texttt{iguales?} : \mathit{rac} \times \mathit{rac} \to \mathfrak{B}$
$\texttt{imprimir} : \mathit{rac} \to \emptyset$

var

$r : \mathit{rac}; n, d, n_1, n_2, d_1, d_2 : \mathbb{Z}$

ecuaciones

$\texttt{numer}(\texttt{racional}(n, d)) \doteq n$
$\texttt{denom}(\texttt{racional}(n, d)) \doteq d$
$\texttt{suma}(\texttt{racional}(n_1, d_1), \texttt{racional}(n_2, d_2)) \doteq \texttt{racional}(n_1 \cdot d_2 + n_2 \cdot d_1, d_1 \cdot d_2)$
$\texttt{mult}(\texttt{racional}(n_1, d_1), \texttt{racional}(n_2, d_2)) \doteq \texttt{racional}(n_1 \cdot n_2, d_1 \cdot d_2)$
$\texttt{iguales?}(\texttt{racional}(n_1, d_1), \texttt{racional}(n_2, d_2)) \doteq n_1 \cdot d_2 = n_2 \cdot d_1$
$\texttt{imprimir}(r)$ { imprime el racional r }

`racional`$(n, 0) \doteq$ *error*

La operación `imprimir` es un caso especial, ya que no es una operación *pura*, sino que su finalidad es la de provocar el **efecto lateral** de imprimir por la salida un número racional de forma «bonita».

Por eso no devuelve ningún valor, cosa que se refleja en su signatura usando $\emptyset$ como tipo de retorno de la operación:

`imprimir` : *rac* $\rightarrow \emptyset$

Y el efecto que produce se indica entre llaves en el apartado de ecuaciones:

`imprimir`(*r*) { imprime el racional *r* }

Introducir operaciones *impuras* amplía la funcionalidad de nuestro tipo abstracto, pero hay que tener cuidado porque se pierde la transparencia referencial y, con ello, nuestra capacidad para razonar matemáticamente sobre las especificaciones.

Según la especificación anterior, podemos suponer que disponemos de un constructor y dos selectores a través de las siguientes funciones:

- `racional(`*n*`,`*d*`)`: devuelve el número racional con numerador *n* y denominador *d*.
- `numer(`*x*`)`: devuelve el numerador del número racional *x*.
- `denom(`*x*`)`: devuelve el denominador del número racional *x*.

Todas las demás operaciones se podrían definir como funciones a partir de éstas tres.

Estamos usando una estrategia poderosa para diseñar programas: el **pensamiento optimista**, ya que todavía no hemos dicho cómo se representa un número racional, o cómo se deben implementar las funciones `numer`, `denom` y `racional`.

Nos basta con saber *qué* hacen y con *suponer* que ya las tenemos.

Así, podemos definir las funciones `suma`, `mult`, `imprimir` e `iguales?` en función de `racional`, `numer` y `denom` sin necesidad de saber cómo están implementadas esas tres funciones ni cómo se representa internamente un número racional.

Esos detalles de implementación quedan ocultos y son innecesarios para definir las funciones `suma`, `mult`, `imprimir` e `iguales?`, ya que bastaría con saber *qué* hacen las funciones `racional`, `numer` y `denom` y no *cómo* lo hacen.

Hasta el punto de que ni siquiera hace falta tener implementadas aún las funciones `racional`, `numer` y `denom` para poder definir las demás. **Suponemos** que las tenemos (*pensamiento optimista*).

Una posible implementación en Python de las operaciones `suma`, `mult`, `imprimir` e `iguales?` a partir de `racional`, `numer` y `denom` podría ser la siguiente:

```
def suma(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return racional(nx * dy + ny * dx, dx * dy)

def mult(x, y):
    return racional(numer(x) * numer(y), denom(x) * denom(y))
```

```
def iguales(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)

def imprimir(x):
    print(numer(x), '/', denom(x), sep='')
```

Esta implementación es correcta porque se ha obtenido a partir de la especificación de los racionales, y en cuanto se tengan implementadas las funciones `racional`, `numer` y `denom`, funcionará perfectamente.

3. Implementaciones

3.1. Implementaciones

Ahora tenemos las operaciones sobre números racionales implementadas sobre las funciones selectoras `numer` y `denom` y la función constructora `racional`, pero aún no hemos implementado estas tres funciones.

Lo que necesitamos es alguna forma de unir un numerador y un denominador en un valor compuesto (una pareja de números).

Podemos usar cualquier representación que nos permita combinar ambos valores (numerador y denominador) en una sola unidad y que también nos permita manipular cada valor por separado cuando sea necesario.

Por ejemplo, podemos usar una lista de dos números enteros para representar un racional mediante su numerador y su denominador:

```
def racional(n, d):
    """Un racional es una lista que contiene el numerador y el denominador."""
    return [n, d]

def numer(x):
    """El numerador es el primer elemento de la lista."""
    return x[0]

def denom(x):
    """El denominador es el segundo elemento de la lista."""
    return x[1]
```

Junto con las operaciones aritméticas que definimos anteriormente, podemos manipular números racionales con las funciones que hemos definido y sin tener que conocer su representación interna:

```
>>> medio = racional(1, 2)
>>> imprimir(medio)
1/2
>>> tercio = racional(1, 3)
>>> imprimir(mult(medio, tercio))
1/6
>>> imprimir(suma(tercio, tercio))
6/9
```

Como muestra el ejemplo anterior, nuestra implementación de números racionales no simplifica las fracciones resultantes.

Podemos corregir ese defecto cambiando únicamente la implementación de `racional`.

Usando el máximo común divisor podemos reducir el numerador y el denominador para obtener un número racional equivalente:

```
from math import gcd

def racional(n, d):
    g = gcd(n, d)
    return [n // g, d // g]
```

Con esta implementación revisada de `racional` nos aseguramos de que los racionales se expresan de la forma más simplificada posible:

```
>>> imprimir(suma(tercio, tercio))
2/3
```

Lo interesante es que este cambio sólo ha afectado al constructor `racional`, y las demás operaciones no se han visto afectadas por ese cambio.

Esto es así porque el resto de las operaciones no conocen ni necesitan conocer la representación interna de un número racional (es decir, la implementación del constructor `racional`). Sólo necesitan conocer la **especificación** de `racional`, la cual no ha cambiado.

Otra posible implementación sería simplificar el racional no cuando se *construya*, sino cuando se *acceda* a alguna de sus partes:

```
def racional(n, d):
    return [n, d]

def numer(x):
    g = gcd(x[0], x[1])
    return x[0] // g

def denom(x):
    g = gcd(x[0], x[1])
    return x[1] // g
```

La diferencia entre esta implementación y la anterior está en cuándo se calcula el máximo común divisor.

- Si en los programas que normalmente usan los números racionales accedemos muchas veces a sus numeradores y denominadores, será preferible calcular el m.c.d. en el constructor.
- En caso contrario, puede que sea mejor esperar a acceder al numerador o al denominador para calcular el m.c.d.
- En cualquier caso, cuando se cambia una representación por otra, las demás funciones (`suma`, `mult`, etc.) no necesitan ser modificadas.

Hacer que la representación dependa sólo de unas cuantas funciones de la interfaz nos ayuda a

diseñar programas, así como a modificarlos, porque nos permite cambiar la implementación por otras distintas cuando sea necesario.

Por ejemplo, si estamos diseñando un módulo de números racionales y aún no hemos decidido si calcular el m.c.d. en el constructor o en los selectores, la metodología de la abstracción de datos nos permite retrasar esa decisión sin perder la capacidad de seguir programando el resto del programa.

4. Niveles y barreras de abstracción

4.1. Niveles de abstracción

Parémonos ahora a considerar algunos de las cuestiones planteadas en el ejemplo de los números racionales.

Hemos definido todas las operaciones de *rac* en términos de un constructor `racional` y dos selectores `numer` y `denom`.

En general, la idea que hay detrás de la **abstracción de datos** es la de:

1. definir un **nuevo tipo de datos** (abstracto),
2. identificar un **conjunto básico de operaciones** sobre las cuales se expresarán todas las operaciones que manipulen los valores de ese tipo, y luego
3. **obligar a usar sólo esas operaciones** para manipular los datos.

Al obligar a usar los datos únicamente a través de sus operaciones, es mucho más fácil cambiar luego la representación interna de los datos abstractos o la implementación de las operaciones básicas sin tener que cambiar el resto del programa.

Para el caso de los números racionales, diferentes partes del programa manipulan números racionales usando diferentes operaciones, como se describe en esta tabla:

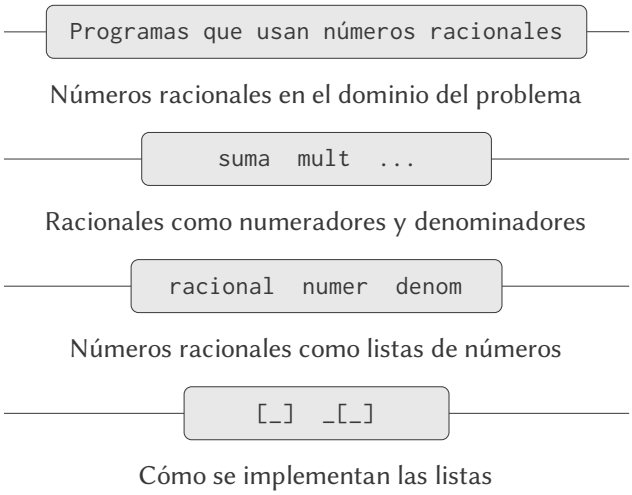
Las partes del programa que…	Tratan a los racionales como…	Usando sólo…
Usan números racionales para realizar cálculos	Valores de datos completos, un todo	`suma`, `mult`, `iguales`, `imprimir`
Crean racionales o implementan operaciones sobre racionales	Numeradores y denominadores	`racional`, `numer`, `denom`
Implementan selectores y constructores de racionales	Parejas de números representadas como listas de dos elementos	Literales de tipo lista `[_]` e indexación `_[_]`

4.2. Barreras de abstracción

Cada una de las filas de la tabla anterior representa un nivel de abstracción, de forma que cada nivel usa las operaciones y las facilidades ofrecidas por el nivel inmediatamente inferior.

Dicho de otra forma: en cada nivel, las funciones que aparecen en la última columna imponen una

barrera de abstracción. Estas funciones son usadas desde un nivel más alto de abstracción e implementadas usando un nivel más bajo de abstracción.



Barreras de abstracción

Se produce una violación de una barrera de abstracción cada vez que una parte del programa que puede utilizar una función de nivel superior utiliza una función de un nivel inferior.

Por ejemplo, una función que calcula el cuadrado de un número racional se implementa mejor en términos de `mult`, que no necesita supone nada sobre cómo se implementa un número racional:

```
def cuadrado(x):
    return mult(x,x)
```

Si hiciéramos referencia directa a los numeradores y los denominadores estaríamos violando una barrera de abstracción:

```
def cuadrado_viola_una_barrera(x):
    return racional(numer(x) * numer(x), denom(x) * denom(x))
```

Y si suponemos que los racionales se representan como listas estaríamos violando dos barreras de abstracción:

```
def cuadrado_viola_dos_barreras(x):
    return [x[0] * x[0], x[1] * x[1]]
```

Las barreras de abstracción hacen que los programas sean más fáciles de mantener y modificar.

Cuantas menos funciones dependan de una representación particular, menos cambios se necesitarán cuando se quiera cambiar esa representación.

Todas las implementaciones de `cuadrado` que acabamos de ver se comportan correctamente, pero

sólo la primera es lo bastante robusta como para soportar bien los futuros cambios de los niveles inferiores.

La función `cuadrado` no tendrá que cambiarse incluso aunque cambiemos la representación interna de los números racionales.

Por el contrario, `cuadrado_viola_una_barrera` tendrá que cambiarse cada vez que cambien las signaturas del constructor o los selectores, y `cuadrado_viola_dos_barreras` tendrá que cambiarse cada vez que cambie la representación interna de los números racionales.

4.3. Propiedades de los datos

Las barreras de abstracción definen la forma en como pensamos sobre los datos.

Por ejemplo, podemos pensar que las operaciones `suma`, `mult`, etc. están definidas sobre *datos* (numeradores, denominadores y racionales) cuyo comportamiento está definido por las funciones `racional`, `numer` y `denom`.

Pero... ¿qué es un *dato*? No basta con decir que es «cualquier cosa implementada mediante determinados constructores y selectores».

Por ejemplo: es evidente que cualquier conjunto arbitrario de tres funciones (un constructor y dos selectores) no sirve para representar adecuadamente a los números racionales.

Además, se tiene que garantizar que, entre el constructor `racional` y los selectores `numer` y `denum`, se cumple la siguiente propiedad:

Si x = `racional(`n, d`)`, entonces `numer(`x`)/denom(`x`) ==` n/d .

De hecho, esta es la única condición que deben cumplir las tres funciones para poder representar adecuadamente a los números racionales.

Una representación válida de un número racional no está limitada a ninguna implementación particular (como, por ejemplo, una lista de dos elementos), sino que nos sirve cualquier implementación que satisfaga la propiedad anterior.

En general, podemos pensar que **los datos abstractos se definen mediante una colección de selectores y constructores junto con algunas propiedades que los datos abstractos deben cumplir**. Mientras se cumplan dichas propiedades (como la anterior de la división), los selectores y constructores constituyen una representación válida de un tipo de datos.

Los detalles de implementación debajo de una barrera de abstracción **pueden cambiar**, pero si no cambia su comportamiento, entonces la abstracción de datos sigue siendo válida y cualquier programa escrito utilizando esta abstracción de datos seguirá siendo correcto.

Este punto de vista tambíen se puede aplicar, por ejemplo, a las parejas con forma de lista que hemos usado para implementar números racionales.

En realidad, tampoco hace falta que sea una lista. Nos basta con cualquier representación que agrupe dos valores juntos y que nos permita acceder a cada valor por separado. Es decir, la propiedad que tienen que cumplir las parejas es que:

Si `p = pareja(x, y)`, entonces `select(p, 0) == x`
y `select(p, 1) == y`.

Tales propiedades se describen como **ecuaciones** en la **especificación algebraica** del tipo abstracto.

5. Las funciones como datos

5.1. Representación funcional

Anteriormente, hemos dicho que los datos se caracterizan por las **operaciones** (constructoras y selectoras) con las que se manipulan y por las **propiedades** que cumplen dichas operaciones, de manera que **podemos cambiar los detalles de implementación** bajo una barrera de abstracción siempre y cuando no cambiemos su comportamiento.

Por tanto, bajo esa barrera de abstracción podemos usar cualquier implementación siempre y cuando logren que las operaciones que definen dicha barrera de abstracción satisfagan las propiedades que deben cumplir.

Por ejemplo, para representar a los números racionales usamos parejas de números, pero esas parejas se pueden representar de muchas maneras. Podemos usar listas, pero en general nos sirve cualquier dato definido por un constructor `pareja` y un selector `select` que cumpla:

Si `p = pareja(x, y)`, entonces `select(p, 0) == x`
y `select(p, 1) == y`.

Esas dos operaciones, `pareja` y `select`, que deben cumplir la condición antes indicada, formarían otra barrera de abstracción sobre la cual se podrían implementar los números racionales:

```
def racional(x, y):
    return pareja(x, y)

def numer(r):
    return select(r, 0)

def denom(r):
    return select(r, 1)
```

A su vez, para implementar las parejas, nos valdría cualquier implementación que satisfaga la propiedad que deben cumplir las parejas. Por ejemplo, cualquier estructura de datos o tipo compuesto que permita almacenar dos elementos juntos y seleccionar cada elemento por separado, como una lista, una tupla o algo similar:

```
def pareja(x, y):
    return [x, y]

def select(p, i):
    return p[i]
```

Pero, de hecho, ni siquiera necesitamos estructuras de datos para representar parejas de números.

Podemos implementar dos funciones `pareja` y `select` que cumplan con la propiedad anterior tan bien como una lista de dos elementos:

```
def pareja(x, y):
    """Devuelve una función que representa una pareja."""
    def get(indice):
        if indice == 0:
            return x
        elif indice == 1:
            return y

    return get

def select(p, i):
    """Devuelve el elemento situado en el índice i de la pareja p."""
    return p(i)
```

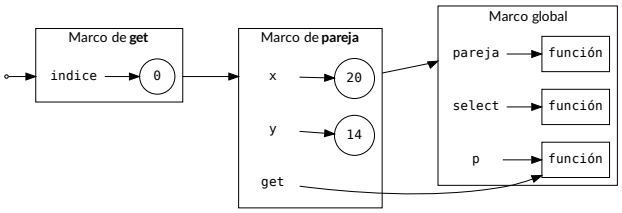
Con esta implementación, podemos crear y manipular parejas:

```
>>> p = pareja(20, 14)
>>> select(p, 0)
20
>>> select(p, 1)
14
```

Ver en Pythontutor

Las variables `x` e `y` son los parámetros de la función `pareja`, es decir, que son locales a ella. Por tanto, las ligaduras entre sus identificadores y las variables que contienen su valores se almacenan en el marco de `pareja`.

La función `get` puede acceder a `x` e `y` ya que se encuentran en su entorno.



Entorno dentro de la función `get` al llamar a `select(p, 0)`

Lo interesante es que el marco de la función `pareja` no se elimina de la memoria al salir de la función con `return get`, ya que la función `get` necesita seguir accediendo a valores (las variables `x` e `y`) cuyas ligaduras se almacenan en el marco de `pareja` y no en el suyo.

Es decir: el intérprete conserva todo el entorno que la función `get` necesita para poder funcionar, incluyendo sus variables no locales, como es el caso aquí de de los parámetros `x` e `y` de la función `pareja`.

La combinación de una función más el entorno necesario para su ejecución se denomina **clausura**.

Las funciones `pareja` y `select`, así definidas, son **funciones de orden superior**: la primera porque devuelve una función y la segunda porque recibe una función como argumento.

La función `get` que devuelve `pareja` y que recibe `select` **representa una pareja**, es decir, un **dato**.

A esto se le denomina **representación funcional**.

El uso de funciones de orden superior para representar datos no se corresponde en absoluto con nuestra noción intuitiva de lo que deben ser los datos. Sin embargo, **las funciones son perfectamente capaces de representar datos compuestos**. En nuestro caso, estas funciones son suficientes para representar parejas en nuestros programas.

Esto no quiere decir que Python realmente implemente las listas mediante funciones. En realidad las implementa de otra forma por razones de eficiencia, pero podría implementarlas con funciones sin ningún problema.

La práctica de la abstracción de datos nos permite cambiar fácilmente unas representaciones por otras, y una de esas representaciones puede ser mediante funciones.

La representación funcional, aunque pueda parecer extraña, es una forma perfectamente adecuada de representar parejas, ya que cumple las propiedades que deben cumplir las parejas.

Este ejemplo también demuestra que la capacidad de manipular funciones como valores (mediante funciones de orden superior) proporciona la capacidad de manipular datos compuestos.

5.2. Estado interno

Ciertos datos pueden tener un **estado interno** consistente en información que el dato almacena (o *recuerda*) y que puede cambiar durante la ejecución del programa.

Por ejemplo:

- Una **lista** posee un estado interno que se corresponde con su **contenido**, es decir, con los **elementos que contiene** en un momento dado.
- Esos elementos pueden **cambiar** durante la ejecución del programa: podemos añadir elementos a la lista, eliminar elementos de la lista o cambiar un elemento de la lista por otro distinto.
- Cada vez que efectuamos alguna de estas operaciones sobre una lista estamos cambiando su estado interno.

Por tanto, la palabra «estado» implica un proceso evolutivo durante el cual ese estado puede ir cambiando.

Al introducir el concepto de «estado interno» en nuestros datos, estamos introduciendo también la capacidad de cambiar dicho estado, es decir, que los datos ahora son mutables, y la mutabilidad es un concepto propio de la programación imperativa.

Además, esto nos va a impedir representar un dato abstracto mutable usando las especificaciones algebraicas que hemos usado hasta ahora, ya que, a partir de ahora, el resultado de una operación puede depender no sólo de lo que dicten las ecuaciones de la especificación sino también de la historia previa que haya tenido el dato abstracto (es decir, de su estado interno).

A cambio, ganaremos la posibilidad de modelar de forma fácil y natural el comportamiento de sistemas y procesos que se dan en el mundo real y que son inherentemente dinámicos, es decir, que cambian con el tiempo y que van pasando por distintos estados a medida que se opera con ellos.

Para ello, aprovecharemos una característica aún no explorada hasta ahora: **las funciones también pueden tener estado interno**.

Por ejemplo, definamos una función que simule el proceso de retirar dinero de una cuenta bancaria.

Crearemos una función llamada `retirar`, que tome como argumento una cantidad a retirar. Si hay suficiente dinero en la cuenta para permitir la retirada, `retirar` devolverá el saldo restante después de la retirada. De lo contrario, `retirar` producirá el error `'Fondos insuficientes'`.

Por ejemplo, si empezamos con 100 € en la cuenta, nos gustaría obtener la siguiente secuencia de valores de retorno al ir llamando a `retirar`:

```
>>> retirar(25)
75
>>> retirar(25)
50
>>> retirar(60)
'Fondos insuficientes'
>>> retirar(15)
35
```

La expresión `retirar(25)`, evaluada dos veces, produce valores diferentes.

Por lo tanto, la función `retirar` **no es pura**.

Llamar a la función no sólo devuelve un valor, sino que también tiene el **efecto lateral** de cambiar la función de alguna manera, de modo que la siguiente llamada con el mismo argumento devolverá un resultado diferente.

Este efecto lateral es el resultado de retirar dinero de los fondos disponibles **provocando un cambio en el estado de una variable** que almacena dichos fondos y que se encuentra **fuera del marco actual**.

Para que todo funcione, debe empezarse con un saldo inicial.

La función `deposito` es una función de orden superior que recibe como argumento un saldo inicial y devuelve la propia función `retirar`, pero de forma que esa función **recuerda** el saldo inicial.

```
>>> retirar = deposito(100)
```

La implementación de `deposito` requiere un **acceso no local** al valor de los fondos iniciales y una **función local** que actualiza y devuelve dicho valor:

```
def deposito(fondos):
    """Devuelve una función que reduce los fondos en cada llamada."""
    def deposito_local(cantidad):
        nonlocal fondos
        if cantidad > fondos:
            return 'Fondos insuficientes'
        fondos -= cantidad  # Asignación a variable no local
        return fondos

    return deposito_local
```

5.3. Paso de mensajes

Al poder asignar valores a una variable no local, hemos podido conservar un estado que es interno para una función pero que evoluciona y cambia con el tiempo a través de llamadas sucesivas a esa función.

Supongamos que queremos ampliar la idea anterior definiendo más operaciones sobre los fondos de la cuenta corriente.

Por ejemplo, además de poder retirar una cantidad, queremos también poder ingresar cantidades en la cuenta, así como consultar en todo momento su saldo actual.

En ese caso, podemos usar una técnica que consiste en usar una función que **despacha** las operaciones en función del *mensaje* recibido.

```
def deposito(fondos):
    def retirar(cantidad):
        nonlocal fondos
        if cantidad > fondos:
            return 'Fondos insuficientes'
        fondos -= cantidad
        return fondos

    def ingresar(cantidad):
        nonlocal fondos
        fondos += cantidad
        return fondos

    def saldo():
        return fondos

    def despacho(mensaje):
        if mensaje == 'retirar':
            return retirar
        elif mensaje == 'ingresar':
            return ingresar
        elif mensaje == 'saldo':
            return saldo
        else:
            raise ValueError('Mensaje incorrecto')

    return despacho
```

Ahora, un depósito se representa internamente como una función que recibe mensajes y los despacha a la función correspondiente:

```
>>> dep = deposito(100)
>>> dep
<function deposito.<locals>.despacho at 0x7f0de1300e18>
>>> dep('retirar')(25)
75
>>> dep('ingresar')(200)
275
>>> dep('saldo')()
275
```

También podemos hacer:

```
>>> dep = deposito(100)
>>> retirar = dep('retirar')
>>> ingresar = dep('ingresar')
>>> saldo = dep('saldo')
>>> retirar(25)
75
>>> ingresar(200)
275
>>> saldo()
275
```

Una variante de esta técnica es la de usar un diccionario para asociar a cada mensaje con cada operación:

```
def deposito(fondos):
    def retirar(cantidad):
        nonlocal fondos
        if cantidad > fondos:
            return 'Fondos insuficientes'
        fondos -= cantidad
        return fondos

    def ingresar(cantidad):
        nonlocal fondos
        fondos += cantidad
        return fondos

    def saldo():
        return fondos

    dic = {'retirar': retirar, 'ingresar': ingresar, 'saldo': saldo}

    def despacho(mensaje):
        if mensaje in d:
            return dic[mensaje]
        else:
            raise ValueError('Mensaje incorrecto')

    return despacho
```

Se denomina **paso de mensajes** a este estilo de programación que consiste en agrupar, dentro de una función que responde a diferentes mensajes, las operaciones que actúan sobre un dato.

El paso de mensajes combina dos técnicas de programación:

- Las funciones de orden superior que devuelven otras funciones.
- El uso de una función que *despacha* a otras funciones dependiendo del mensaje recibido.

6. Abstracción de datos y modularidad

6.1. El tipo abstracto como módulo

Claramente, un **tipo abstracto** representa una **abstracción**:

- Se **destacan** los detalles (normalmente pocos) de la **especificación**, es decir, el *comportamiento*

observable del tipo. Es de esperar que este aspecto sea bastante estable y cambie poco durante la vida útil del programa.

- Se **ocultan** los detalles (normalmente numerosos) de la **implementación**. Este aspecto es, además, propenso a cambios.

Y estas propiedades anteriores hacen que el tipo abstracto sea el concepto ideal alrededor del cual basar la descomposición en módulos de un programa grande.

Recordemos que para que haya una buena modularidad:

- Las **conexiones** del módulo con el resto del programa han de ser pocas y simples. De este modo se espera lograr una relativa independencia en el desarrollo de cada módulo con respecto a los otros.
- La **descomposición** en módulos ha de ser tal que la mayor parte de los cambios y mejoras al programa impliquen modificar sólo un módulo o un número muy pequeño de ellos.
- El **tamaño** de un módulo ha de ser el adecuado: si es demasiado grande, será difícil realizar cambios en él; si es demasiado pequeño, los costes de integración con otros módulos aumenta.

La parte del código fuente de un programa dedicada a la definición de un tipo abstracto de datos es un **candidato a módulo** que cumple los siguientes requisitos:

- La **interfaz** del tipo abstracto con sus usuarios es un ejemplo de pocas y simples conexiones con el resto del programa: los usuarios simplemente invocan sus operaciones permitidas. Otras conexiones más peligrosas, como compartir variables entre módulos o compartir el conocimiento acerca de la estructura interna, son imposibles.
- La **implementación** puede cambiarse libremente sin afectar al funcionamiento de los módulos usuarios. Es de esperar, por tanto, que muchos cambios al programa queden localizados en el interior de un sólo módulo.
- El **tamaño** de una sola función que implementa una abstracción funcional es demasiado pequeño para ser útil como unidad modular. En cambio, la definición de un tipo abstracto consta, en general, de una colección de funciones más una representación, lo que proporciona un tamaño más adecuado.

Un tipo abstracto de datos conecta perfectamente con los cuatro conceptos más importantes relacionados con la modularidad:

- Es una **abstracción**: se usa sin necesidad de saber cómo se implementa.
- **Oculta información**: los detalles y las decisiones de diseño quedan en la implementación del tipo abstracto, y son innecesarios para poder usarlo.
- Es **funcionalmente independiente**: es un módulo destinado a una sola tarea, con alta cohesión y bajo acoplamiento.
- Es **reutilizable**: su comportamiento puede resultar tan general que puede usarse en diferentes programas con ninguna o muy poca modificación.

Bibliografía

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass. : New York: MIT Press ; McGraw-Hill.

DeNero, John. n.d. “Composing Programs.” http://www.composingprograms.com.

Peña Marí, Ricardo. 2003. *Diseño de Programas: Formalismo Y Abstracción*. Madrid: Prentice Hall.

Python Software Foundation. n.d. “Sitio Web de Documentación de Python.” https://docs.python.org/3.