

Abstracción de datos

Ricardo Pérez López

IES Doñana, curso 2020/2021

Generado el 17 de noviembre de 2020 a las 18:57:00

Índice general

| | |
|--|-----------|
| 1. Introducción | 1 |
| 1.1. Introducción | 1 |
| 2. Especificaciones | 5 |
| 2.1. Sintaxis | 5 |
| 2.2. Operaciones | 6 |
| 2.3. Ejemplos | 6 |
| 3. Implementaciones | 11 |
| 3.1. Implementaciones | 11 |
| 4. Niveles y barreras de abstracción | 13 |
| 4.1. Niveles de abstracción | 13 |
| 4.2. Barreras de abstracción | 14 |
| 4.3. Propiedades de los datos | 15 |
| 5. Las funciones como datos | 16 |
| 5.1. Clausuras | 16 |
| 5.2. Representación funcional | 21 |
| 5.3. Estado interno | 22 |
| 5.4. Paso de mensajes | 24 |
| 5.5. Especificación de datos abstractos con estado interno | 25 |
| 6. Abstracción de datos y modularidad | 27 |
| 6.1. El tipo abstracto como módulo | 27 |

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 *ignorán* 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 un producto: 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 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,
2. darle un nombre a todo el conjunto, y
3. 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.

Además de combinar operaciones, también podemos **combinar datos**.

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.

Es decir: los datos compuestos, así sin más, no funcionan como abstracciones.

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 un **producto**.

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

Por tanto, las abstracciones de datos son construcciones que acaban formando parte del programa, de la misma manera que ocurre con las abstracciones funcionales.

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 y sentencias simples | Literales y 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 cómo se representan los valores del tipo abstracto en la memoria del ordenador.
- **Protección:** sólo se pueden utilizar con sus valores aquellas operaciones previstas en la especificación.

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

1. **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.

2. **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.

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

- Barbara Liskov

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
```

$\langle \text{ecuación} \rangle^+$

Donde:

$\langle \text{decl_var} \rangle ::= \langle \text{variable} \rangle : \langle \text{tipo} \rangle$
 $\langle \text{ecuación} \rangle ::= \langle \text{izquierda} \rangle \doteq \langle \text{derecha} \rangle$

Las **operaciones** son **abstracciones funcionales**.

Las **ecuaciones** son las **propiedades** que deben cumplir las operaciones.

2.2. Operaciones

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

- **Constructoras**: operaciones que devuelven un valor de tipo T .

A su vez, las constructoras se dividen en:

- * **Generadoras**: son 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 y 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
    [] : → lista
    [_] : elemento → lista
    _ ++ _ : lista × lista → lista
    _ : _ : elemento × lista → lista
    len : lista → ℕ
  var
    x : elemento; l, l1, l2, l3 : lista
  ecuaciones
    x : l ⇐ [x] ++ l
    l ++ [] ⇐ l
    [] ++ l ⇐ l
    (l1 ++ l2) ++ l3 ⇐ l1 ++ (l2 ++ l3)
    len([]) ⇐ 0
    len([x]) ⇐ 1
```

```
len(x : l) ≐ 1 + len(l)
len(l1 ++ l2) ≐ len(l1) + len(l2)
```

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.

Ejercicio

1. ¿A qué categoría pertenecen 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.
- En lugar de cambiar el estado de un dato compuesto, se «simula» ese cambio creando un nuevo dato con los cambios aplicados.

Por ejemplo: 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 ≐ [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
    [] : → lista
    _:_ : elemento × lista → lista
    [_] : elemento → lista
```

```

_++_ : lista × lista → lista
len : lista → ℕ
var
  x : elemento; l, l1, l2 : lista
ecuaciones
  [x] ≐ x : []
  [] ++ l ≐ l
  (x : l1) ++ l2 ≐ x : (l1 ++ l2)
  len([]) ≐ 0
  len(x : l) ≐ 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.

Para ello, tenemos que recordar que una lista sólo puede tener dos formas posibles:

- Una lista vacía:

`[]`

- Un elemento seguido de otra lista:

`x : l1`

Por ejemplo, la lista `[1, 2, 3]` es `1:(2:(3:[]))`.

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

a. Caso $l = []$ (caso base):

| | |
|-------------------|---|
| $l ++ []$ | # por definición de <i>l</i> |
| $\doteq [] ++ []$ | # por el primer axioma de <code>++</code> |
| $\doteq []$ | # por definición de <i>l</i> |
| $\doteq l$ | |

b. Caso $l = x : l_1$ (caso inductivo):

Suponemos que la propiedad se cumple para *l₁* (hipótesis inductiva), es decir, que $l_1 ++ [] \doteq l_1$. En tal caso, tenemos:

| | |
|--------------------------|--|
| $l ++ []$ | # por definición de <i>l</i> |
| $\doteq (x : l_1) ++ []$ | # por el segundo axioma de <code>++</code> |

| | |
|--------------------------|---------------------------|
| $\doteq x : (l_1 ++ [])$ | # por hipótesis inductiva |
| $\doteq x : l_1$ | # por definición de l |
| $\doteq l$ | |

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  $\rightarrow$   $\mathcal{B}$ 
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
  desapilar(pvacia)  $\doteq$  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 racional :  $\mathbb{Z} \times \mathbb{Z} \rightarrow \text{rac}$ 
  numer : rac  $\rightarrow \mathbb{Z}$ 
  denom : rac  $\rightarrow \mathbb{Z}$ 
  suma : rac  $\times$  rac  $\rightarrow \text{rac}$ 
  mult : rac  $\times$  rac  $\rightarrow \text{rac}$ 
  iguales? : rac  $\times$  rac  $\rightarrow \mathcal{B}$ 
  imprimir : rac  $\rightarrow \emptyset$ 

var
  r : rac; n, d, n1, n2, d1, d2 :  $\mathbb{Z}$ 

ecuaciones
  numer(racional(n, d))  $\doteq n$ 
  denom(racional(n, d))  $\doteq d$ 
  suma(racional(n1, d1), racional(n2, d2))  $\doteq \text{racional}(n_1 \cdot d_2 + n_2 \cdot d_1, d_1 \cdot d_2)$ 
  mult(racional(n1, d1), racional(n2, d2))  $\doteq \text{racional}(n_1 \cdot n_2, d_1 \cdot d_2)$ 
  iguales?(racional(n1, d1), racional(n2, d2))  $\doteq n_1 \cdot d_2 = n_2 \cdot d_1$ 
  imprimir(r) { imprime el racional r }
  racional(n, 0)  $\doteq \text{error}$ 

```

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 *suponer* que ya las tenemos.

Así, podemos definir las funciones `suma`, `mult`, `imprimir` e `iguales?` a partir 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.

Además, si hoy decidimos calcular el m.c.d. en el constructor pero el día de mañana cambiamos de opinión, podemos ponerlo en los selectores sin que el resto del programa se vea afectado. Sólo habría que cambiar la implementación del tipo abstracto.

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.

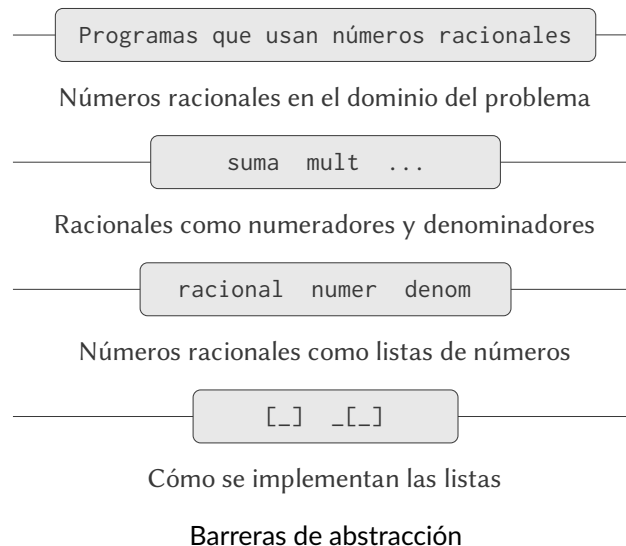
Para el caso de los números racionales, diferentes partes del programa manipulan números racionales usando diferentes operaciones, como se describe en esta tabla:

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

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.



Se produce una **violación de una barrera de abstracción** cada vez que una parte del programa que puede utilizar una función de un determinado nivel, utiliza una función de un nivel más bajo.

Por ejemplo, una función que calcula el cuadrado de un número racional se implementa mejor en términos de `mult`, que no necesita suponer 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 usamos el conocimiento de 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]]
```

Cuantas menos barreras de abstracción se crucen al escribir programas, mejor.

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 especificaciones 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 de qué forma 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 ciertos constructores y selectores».

Siguiendo con el mismo ejemplo: es evidente que cualquier grupo 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 `denom`, se cumple la siguiente propiedad:

Si $x = \text{racional}(n, d)$, entonces $\text{numer}(x)/\text{denom}(x) == n/d$.

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 decir que **los datos abstractos se definen mediante una colección de constructores y selectores junto con algunas propiedades que los datos abstractos deben cumplir**.

Mientras se cumplan dichas propiedades (como la anterior de la división), los constructores y selectores constituyen una representación válida de un tipo de datos abstracto.

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

Este punto de vista también se puede aplicar, por ejemplo, a las listas que hemos usado para implementar números racionales.

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

Si $p = \text{pareja}(x, y)$, entonces $\text{select}(p, 0) == x$
y $\text{select}(p, 1) == y$.

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

5. Las funciones como datos

5.1. Clausuras

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 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 $p = \text{pareja}(x, y)$, entonces $\text{select}(p, 0) == x$
y $\text{select}(p, 1) == y$.

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)
```

Igualmente, 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

Cuando se llama a la función `pareja`, ésta devuelve otra función llamada `get` (por tanto, `pareja` es una función de orden superior).

Además, `get` es una función local a la función `pareja`, ya que está definida en el ámbito de `pareja`.

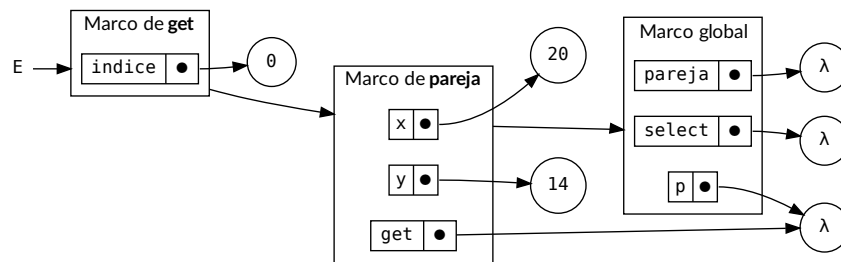
Por tanto, el marco de `get` apunta al de `pareja` en el entorno.

Las variables `x` e `y` son los parámetros de la función `pareja`, así que son locales a ella y, por tanto, sus ligaduras y referencias se almacenan en su marco.

Pero la función `get` necesita acceder al valor de esas variables.

Esas variables están en el entorno de `get`, ya que el marco de `get` apunta al de `pareja` en el entorno.

En consecuencia, la función `get` puede acceder a `x` e `y`.



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

El problema es que, cuando se va a ejecutar `get` a través de la llamada a `select(p, 0)`, el marco de `pareja` no estaría en la pila, ya que no hay ninguna llamada activa a la función `pareja`.

Y para que `get` pueda seguir accediendo a las variables `x` e `y` (que son locales a `pareja`), es necesario que el marco de `pareja` no se elimine de la memoria cuando finalice la ejecución de `pareja`.

Necesitamos un nuevo mecanismo que permita resolver este problema.

El **entorno restringido** de una función está formado por todos los marcos del entorno que afecta durante la ejecución de la función, **excepto el marco de la propia función y el marco global**.

Por tanto, el entorno restringido de `get` empieza en el marco de `pareja`, ya que `get` es local a `pareja`.

El entorno restringido de `get` es el entorno que la función `get` necesita recordar para poder funcionar.

La combinación de una función más su entorno restringido se denomina **clausura**.

Clausura = función + entorno restringido

Una clausura, por tanto, agrupa una función con la información que se necesita recordar para que la función funcione.

El motivo de no incluir el propio marco y el marco global, es que el propio marco se crea cuando se llama a la función, y el marco global está siempre disponible, así que no es necesario recordarlos.

Después de hacer:

```
p = pareja(20, 14)
```

tenemos que `p` contiene una clausura.

Esa clausura esta formada por:

- La **función** que hay en `p`, que es la función `get`.
- El **entorno restringido** de `p`, que empieza en el marco de `pareja` (ya que la función `get` se definió en el ámbito de la función `pareja`), el cual recuerda los valores que tenían las variables locales de `pareja` cuando se creó la clausura.

Pero ahora, el marco de `pareja` no puede estar en la pila de control, ya que esa función no tiene ninguna llamada activa en este momento.

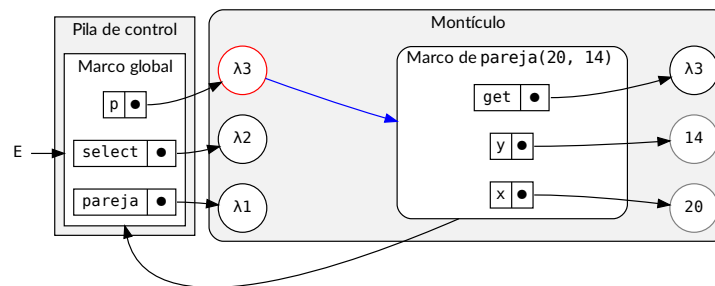
El problema es que la clausura `p` necesita ese marco para poder funcionar.

Para no perder el marco de la función `pareja` cuando termina de ejecutarse ésta, el intérprete saca su marco de la pila pero no lo destruye, sino que **lo guarda en el montículo** como si fuera un dato más.

Ahora la clausura hace referencia al marco dentro del montículo, es decir, que la clausura guarda una referencia al marco, que ahora es un dato más dentro del montículo.

El marco seguirá existiendo en el montículo mientras haya una referencia que apunte a él, como cualquier otro dato.

Las clausuras las representaremos gráficamente como una función de la que sale una flecha que va hasta el marco del ámbito donde se definió la función (el primer marco de su entorno restringido).



Pila de control después de hacer `p = pareja(20, 14)`

El **círculo rojo** representa la clausura, y la **flecha azul** apunta al entorno restringido de la clausura, que aquí contiene sólo al marco de `pareja`.

El marco de `pareja` apunta a su vez al marco global (aunque éste no forme parte del entorno restringido) para así marcar que `pareja` es el último marco del entorno restringido.

Por tanto, la clausura almacenada en `p` contiene la función y la parte del entorno que es necesario recordar para poder ejecutar la función.

Aquí, la clausura apunta al marco de `pareja`, ya que es local a `pareja` y necesita acceder a los datos definidos en el ámbito de `pareja`.

Cuando se llame a la función `p` se creará su marco en la pila, el cual apuntará al marco de `pareja` ya que es el marco a donde apunta el entorno restringido de la clausura.

El marco de `pareja`, a su vez, apuntará al global.

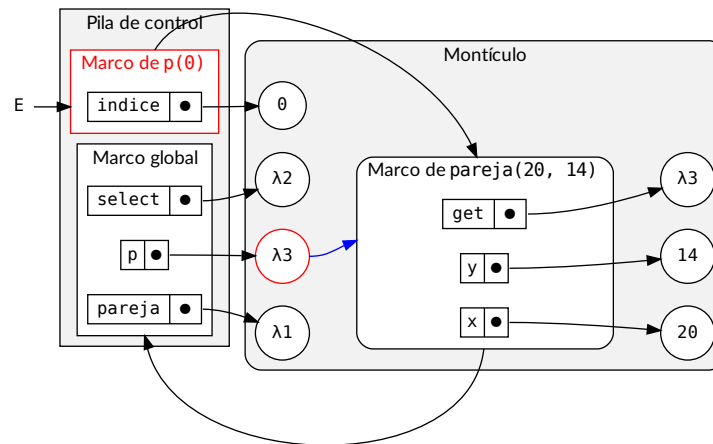
Por tanto, durante la ejecución de `p`, el entorno estaría formado por el marco de `p`, seguido del marco de `pareja` (almacenado en el montículo), seguido del marco global.

Naturalmente, el marco de `p` permanecerá en la pila mientras esté activa la llamada a `p`.

Por ejemplo, si hacemos:

```
p = pareja(20, 14)
x = p(0)
```

durante la ejecución de `p(0)` tendremos lo siguiente:

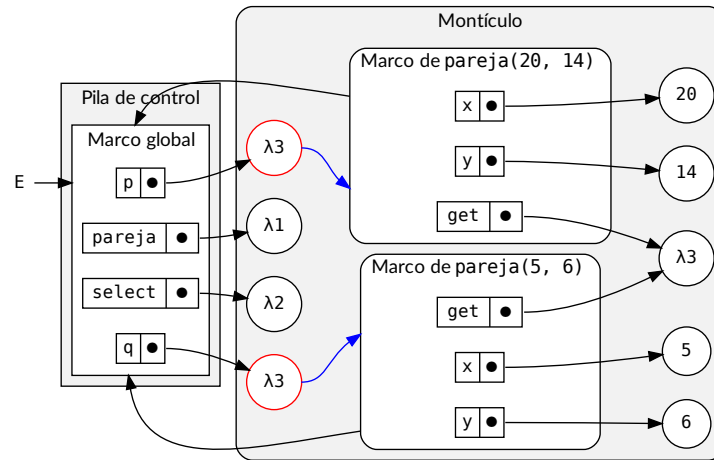


Estado del programa durante la ejecución de `x = p(0)`

Si tenemos dos clausuras `p` y `q` creadas de la siguiente manera:

```
p = pareja(20, 14)
q = pareja(5, 6)
```

en memoria tendremos lo siguiente:

Estado del programa tras crear `p` y `q`

Al llamar a `select(p, i)` se llama luego a `p(i)`, pero como `p` representa a la función `get`, realmente se llama a `get(i)`.

Lo mismo pasa al llamar a `select(q, i)`: se llama a `q(i)` pero, como `q` también representa a `get`, realmente se llama a `get(i)`.

Lo interesante es que `p` y `q` son referencias a la misma función `get`, pero cada una de ellas recuerda el valor que tenían las variables no locales (la `x` y la `y` de `pareja`) cuando se crearon `p` y `q`.

Es decir: `p` y `q` son funciones que recuerdan el contexto en el que fueron creadas.

Por tanto:

- `p` es la función `get` pero, para ella, `x` vale 20 e `y` vale 14.
- `q` es la función `get` pero, para ella, `x` vale 5 e `y` vale 6.

Así que `p` y `q` no son exactamente la misma cosa.

5.2. Representación funcional

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` (la función que devuelve `pareja` y que recibe `select`) **representa una pareja**, es decir, un **dato**.

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

Ese dato abstracto tiene un constructor (`pareja`) y un selector (`select`).

El uso de funciones de orden superior para representar datos no se corresponde con nuestra idea 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.

5.3. 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 en el tiempo, 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**.

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

Y, por supuesto, nos va a **impedir usar el modelo de sustitución** para razonar sobre nuestros datos, por lo que tendremos que usar el modelo de **máquina de estados**.

A cambio, ganaremos dos cosas:

- La posibilidad de modelar **sistemas modulares**, basados en partes independientes que actúan como elementos autónomos.
- 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
```

5.4. 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 dic:
            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.

5.5. Especificación de datos abstractos con estado interno

Ya hemos dicho que las especificaciones algebraicas no nos sirven para especificar un tipo abstracto que contenga estado interno y mutabilidad, porque las operaciones ya no conservan la transparencia referencial y sus propiedades ya no siempre se pueden describir con ecuaciones.

Lo que sí se puede hacer es usar el lenguaje natural para describir dichas propiedades.

El resultado es mucho menos elegante y formal, además de que favorece la aparición de ambigüedades en la interpretación, pero es lo mejor que podemos hacer, en general.

Por ejemplo, la especificación del tipo **Depósito** podría expresarse así si no hubiera mutabilidad ni estado interno:

```

espec depósito
operaciones
  depósito :  $\mathbb{R} \rightarrow \text{depósito}$ 
  parcial retirar :  $\text{depósito} \times \mathbb{R} \rightarrow \mathbb{R}$ 
  ingresar :  $\text{depósito} \times \mathbb{R} \rightarrow \mathbb{R}$ 
  saldo :  $\text{depósito} \rightarrow \mathbb{R}$ 
ecuaciones
   $f < c \Rightarrow \text{retirar}(\text{depósito}(f), c) \doteq \text{error}$ 
   $f \geq c \Rightarrow \text{retirar}(\text{depósito}(f), c) \doteq \text{depósito}(f - c)$ 
   $\text{ingresar}(\text{depósito}(f), c) \doteq \text{depósito}(f + c)$ 
   $\text{saldo}(\text{depósito}(f)) \doteq f$ 

```

depósito es la operación generadora.

retirar e **ingresar** son operaciones modificadoras.

saldo es una operación selectora.

Si el dato abstracto es mutable y recuerda su estado interno, las operaciones modificadoras ya no producen un nuevo dato **Depósito** a partir de otro, sino que cambian el estado interno del dato existente.

En tal caso, la especificación debe describir el **efecto** que producen las operaciones modificadoras sobre el dato abstracto.

```

espec depósito
operaciones
  depósito :  $\mathbb{R} \rightarrow \text{depósito}$ 
  parcial retirar :  $\text{depósito} \times \mathbb{R} \rightarrow$ 
  ingresar :  $\text{depósito} \times \mathbb{R} \rightarrow$ 
  saldo :  $\text{depósito} \rightarrow \mathbb{R}$ 
var
   $d : \text{depósito}; f, c : \mathbb{R}$ 
ecuaciones
  depósito( $f$ ) { Crea un depósito con un fondo (estado interno) inicial de  $f$ 
               y lo devuelve }
  saldo( $d$ )  $\doteq$  «los fondos actuales de  $d$ »
  saldo( $d$ )  $\geq c \Rightarrow \text{retirar}(d, c)$  { Reduce en  $c$  unidades los fondos del depósito  $d$  }
  saldo( $d$ )  $< c \Rightarrow \text{retirar}(d, c) \doteq \text{error}$ 
  ingresar( $d, c$ ) { Aumenta en  $c$  unidades los fondos del depósito  $d$  }

```

Los efectos que produce una operación se indican entre llaves { } al lado de la operación correspondiente.

Toda operación *impura* debe indicar sus efectos entre llaves.

La signatura de las operaciones **retirar** e **ingresar** indican mediante « \rightarrow » que son operaciones que no devuelven ningún valor, ya que su cometido es el de provocar un efecto lateral (en este caso,

modificar el estado interno del **Depósito**).

En concreto, los efectos que producen dichas operaciones son los de cambiar los fondos del depósito, restándole o sumándole una cantidad.

Por tanto, una vez ejecutada cualquiera de esas operaciones, se habrá cambiado el estado interno del dato abstracto.

Otra opción es que las operaciones modificadoras devolvieran un valor además de cambiar el estado interno del dato abstracto (por ejemplo, el saldo que queda en el depósito tras ingresar o retirar efectivo).

En tal caso, podría quedar algo así:

```

espec depósito
operaciones
  depósito :  $\mathbb{R} \rightarrow \text{depósito}$ 
  parcial retirar :  $\text{depósito} \times \mathbb{R} \rightarrow \mathbb{R}$ 
  ingresar :  $\text{depósito} \times \mathbb{R} \rightarrow \mathbb{R}$ 
  saldo :  $\text{depósito} \rightarrow \mathbb{R}$ 
var
  d : depósito; f, c :  $\mathbb{R}$ 
ecuaciones
  depósito(f) { Crea un depósito con un fondo (estado interno) inicial de f
                y lo devuelve }
  saldo(d)  $\doteq$  «los fondos actuales de d»
  saldo(d)  $\geq c \Rightarrow$  retirar(d, c) { Reduce en c unidades los fondos del depósito d
                                   y devuelve el saldo que queda }
  saldo(d)  $< c \Rightarrow$  retirar(d, c)  $\doteq$  error
  ingresar(d, c) { Aumenta en c unidades los fondos del depósito d
                  y devuelve el saldo que queda }

```

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.
- **Ocultá 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>.