Programación modular I

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Introducción

1.1. Modularidad

La **programación modular** es una técnica de programación que consiste en descomponer y programar nuestro programa en partes llamadas **módulos**.

- El concepto de *módulo* hay que entenderlo en sentido amplio: cualquier parte de un programa se puede considerar «módulo».

Equivale a la técnica clásica de resolución de problemas basada en 1) descomponer un problema en subproblemas; 2) resolver cada subproblema por separado; y 3) combinar las soluciones para así obtener la solución al problema original.

La **modularidad** es la propiedad que tienen los programas escritos siguiendo los principios de la programación modular.

El concepto de modularidad se puede estudiar a nivel *metodológico* y a nivel *práctico*.

A nivel metodológico, la modularidad nos proporciona una herramienta más para controlar la complejidad de forma similar a como lo hace la abstracción.

Todos los mecanismos de control de la complejidad actúan de la misma forma: la mente humana es incapaz de mantener la atención en muchas cosas a la vez, así que lo que hacemos es centrarnos en una parte del problema y dejamos a un lado el resto momentáneamente.

- La **abstracción** nos permite controlar la complejidad permitiéndonos estudiar un problema o su solución por niveles, centrándonos en lo esencial e ignorando los detalles que a ese nivel resultan innecesarios.
- Con la **modularidad** buscamos descomponer conceptualmente el programa en partes que se puedan estudiar y programar por separado, de forma más o menos independiente, lo que se denomina **descomposición lógica**.

A nivel práctico, la modularidad nos ofrece una herramienta que nos permite partir el programa en partes más manejables.

A medida que los programas se hacen más y más grandes, el esfuerzo de mantener todo el código dentro de un único *script* se hace mayor.

No sólo resulta incómodo mantener todo el código en un mismo archivo, sino que además resulta intelectualmente más difícil de entender.

Lo más práctico es descomponer físicamente nuestro programa en una colección de archivos fuente que se puedan trabajar por separado, lo que se denomina **descomposición física**.

Un módulo es, pues, una parte de un programa que se puede estudiar, entender y programar por separado con relativa independencia del resto del programa.

Por tanto, podríamos considerar que una función es un ejemplo de módulo, ya que se ajusta a esa definición (salvo quizás que no habría *descomposición física*, aunque se podría colocar cada función en un archivo separado y entonces sí).

Sin embargo, descomponer un programa en partes usando únicamente como criterio la descomposición funcional no resulta adecuado en general, ya que muchas veces nos encontramos con funciones que no actúan por separado, sino de forma conjunta formando un todo interrelacionado.

Además, un módulo no tiene por qué ser simplemente una abstracción funcional, sino que también puede tener su propio estado interno en forma de datos (variables) manipulables desde dentro del módulo pero también desde fuera.

Por ejemplo, supongamos un conjunto de funciones que manipulan números racionales.

Tendríamos funciones para crear racionales, para sumarlos, para multiplicarlos, para simplificarlos... Y todas esas funciones trabajarían conjuntamente, actuando sobre la misma colección de datos (la representación interna que usa el módulo para implementar los números racionales).

Esos datos (es decir, esa representación interna) también formarían parte del módulo y constituirían su estado interno.

Por consiguiente, aunque resulta muy apropiado considerar cada función anterior por separado, también resulta evidente que debemos considerarlas como formando un todo conjunto con los datos que manipulan: el *módulo de manipulación de números racionales*.

De lo dicho hasta ahora se deducen varias conclusiones importantes:

- Un módulo es una parte del programa.
- Los módulos nos permiten descomponer el programa en **partes independientes y manejables por separado**.
- Una función, en general, no es una candidata con suficiente entidad como para ser considerada un módulo.
- Los módulos, en general, agrupan colecciones de **funciones interrelacionadas**.
- Los módulos, en general, también poseen un **estado interno** en forma de estructuras de datos, manipulable desde el interior del módulo así como desde el exterior del mismo usando las funciones que forman el módulo.
- A nivel práctico, los módulos se programan físicamente en **archivos separados** del resto del programa.

1.2. Beneficios de la programación modular

El tiempo de desarrollo se reduce porque grupos separados de programadores pueden trabajar cada uno en un módulo con poca necesidad de comunicación entre ellos.

Se mejora la productividad del producto resultante, porque los cambios (pequeños o grandes) realizados en un módulo no afectarían demasiado a los demás.

Comprensibilidad, porque se puede entender mejor el sistema completo cuando se puede estudiar módulo a módulo en lugar de tener que estudiarlo todo a la vez.

2. Diseño modular

2.1. Partes de un módulo

Desde la perspectiva de la programación modular, un programa está formado por una colección de módulos que interactúan entre sí.

Puede decirse que un módulo proporciona una serie de *servicios* que son *usados* o *consumidos* por otros módulos del programa.

Así que podemos estudiar el diseño de un módulo desde **dos puntos de vista complementarios**:

- El **creador** o **implementador** del módulo es la persona encargada de la programación del mismo y, por tanto, debe conocer todos los detalles internos al módulo, necesarios para que éste funcione (es decir, su **implementación**).
- Los **usuarios** del módulo son los programadores que desean usar ese módulo en sus programas. También se les llama así a los módulos de un programa que usan a ese módulo (lo necesitan para funcionar).

  A la parte que un usuario necesita conocer para poder usar el módulo se le denomina la **interfaz** del módulo.

Los **usuarios** están interesados en usar al módulo como una **entidad abstracta** sin necesidad de conocer los *detalles internos* del mismo, sino sólo lo necesario para poder consumir los servicios que proporciona (su **interfaz**).

Concretando, un módulo tendrá:

- Un **nombre** (que generalmente coincidirá con el nombre del archivo en el que reside).
- Una **interfaz**, formada por un conjunto de **especificaciones de funciones** que permiten al usuario consumir sus servicios, así como manipular y acceder al estado interno desde fuera del módulo.

  Es posible que la interfaz también incluya **constantes**.
- Una **implementación**, formada por:
  * Su posible estado interno en forma de **variables** locales al módulo.
  * Un conjunto de **funciones auxiliares** pensadas para ser usadas exclusivamente por el propio módulo de manera interna, pero no por otras partes del programa.

2.1.1. Interfaz

La **interfaz** es la parte del módulo que el **usuario** del mismo necesita conocer para poder utilizarlo.

Es la parte **expuesta, pública o visible** del mismo.

También se la denomina su **API** (*Application Program Interface*).

Debería estar perfectamente **documentada** para que cualquier potencial usuario tenga toda la información necesaria para poder usar el módulo sin tener que conocer o acceder a partes internas del mismo.

En general **debería estar formada únicamente por funciones** (y, tal vez, **constantes**) que el usuario del módulo pueda llamar para consumir los servicios que ofrece el módulo.

Esas funciones deben usarse como *abstracciones funcionales*, de forma que el usuario sólo necesite conocer las **especificaciones** de las funciones y no sus *implementaciones* concretas (el *cuerpo* o código de las funciones).

Acabamos de decir que la interfaz de un módulo debería estar formada únicamente por **funciones** (y, tal vez, *constantes*).

En teoría, la interfaz podría estar formada también por (algunas o todas las) **variables locales al módulo**, pero en la práctica eso no resulta apropiado, ya que cualquier cambio posterior en la representación interna de los datos almacenados en esas variables afectaría al resto de los módulos que acceden a dichas variables.

Más adelante estudiaremos este aspecto en profundidad cuando hablemos del **principio de ocultación de información**.

2.1.2. Implementación

La **implementación** es la parte del módulo que queda **oculta a los usuarios** del mismo.

Es decir: es la parte que los usuarios del módulo no necesitan (ni deben) conocer para poder usarlo adecuadamente.

Está formada por todas las **variables locales al módulo** que almacenan su estado interno, junto con las funciones que utiliza el propio módulo para gestionarse a sí mismo y que no forman parte de su interfaz (**funciones *auxiliares***).

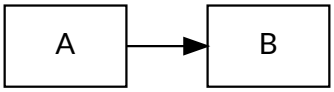
La implementación debe poder cambiarse tantas veces como sea necesario sin que por ello se tenga que cambiar el resto del programa.

2.2. Diagramas de estructura

Los diferentes módulos que forman un programa y la relación que hay entre ellos se puede representar gráficamente mediante un *diagrama de descomposición modular* o **diagrama de estructura**.

En el diagrama de estructura, cada módulo se representa mediante un rectángulo y las relaciones entre cada par de módulos se dibujan como una línea con punta de flecha entre los dos módulos.

Una flecha dirigida del módulo *A* al módulo *B* representa que el módulo *A* *utiliza* o *llama* o *depende* del módulo *B*.



A depende de *B*

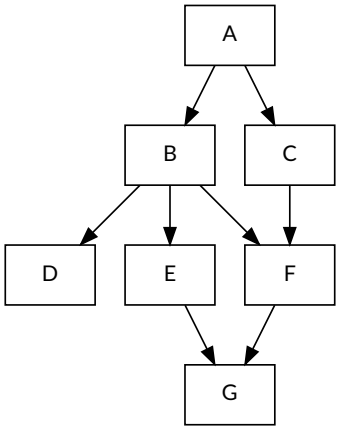


Diagrama de estructura

2.3. Programación modular en Python

En Python, un módulo es otra forma de llamar a un *script*. Es decir: «módulo» y «*script*» son sinónimos en Python.

Los módulos contienen definiciones y sentencias.

El nombre del archivo es el nombre del módulo con extensión `.py`.

Dentro de un módulo, el nombre del módulo (como cadena) se encuentra almacenado en la variable global `__name__`.

Cada módulo tiene su propio ámbito local, que es usado como el ámbito global de todas las funciones definidas en el módulo.

Por tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse de posibles colisiones accidentales con las variables globales de otros módulos.

2.3.1. Importación de módulos

Para que un módulo pueda usar a otros módulos tiene que **importarlos** usando la orden `import`. Por ejemplo, la siguiente sentencia importa el módulo `math` dentro del módulo actual:

```
import math
```

Al importar un módulo de esta forma, lo que se hace es incorporar la definición del propio módulo (el módulo *importado*) en el ámbito actual del módulo o *script* que ejecuta el `import` (el módulo *importador*).

O dicho de otra forma: se incorpora al marco actual (es decir, el marco del ámbito donde se ejecuta el `import`) la ligadura entre el nombre del módulo importado y el propio módulo, por lo que el módulo importador puede acceder al módulo importado a través de su nombre.

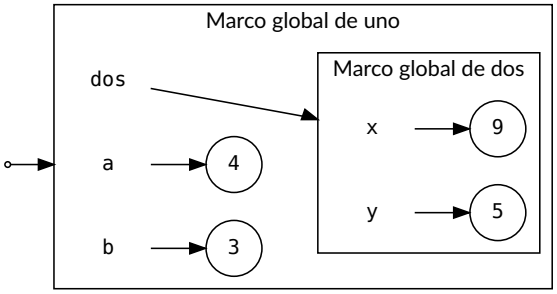
De esta forma, lo que se importa dentro del marco actual no es el contenido del módulo importado, sino el módulo en sí.

Por ejemplo, si tenemos:

```
# uno.py

import dos

a = 4
b = 3
```

```
# dos.py

x = 9
y = 5
```

al final de la ejecución del script `uno.py` tendremos:



Importación del módulo `dos` en `uno`

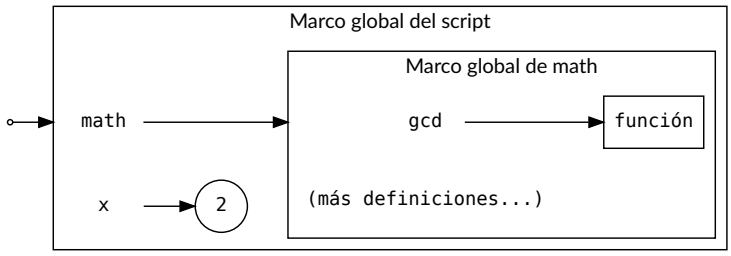
Eso significa que los módulos en Python son internamente un dato más, al igual que las listas o las funciones: se pueden asignar a variables, se pueden borrar de la memoria con `del`, etc.

Y significa, además, que los módulos tienen su propio ámbito y, por tanto, crean su propio marco donde se almacenan sus definiciones, aunque eso es algo que ya sabíamos.

Para acceder al contenido del módulo importado, indicamos el nombre de ese módulo seguido de un punto (`.`) y el nombre del contenido al que queramos acceder.

Por ejemplo, para acceder a la función `gcd` definida en el módulo `math` haremos:

```
import math

x = math.gcd(16, 6)
```



Entorno en la última línea del script anterior

Se recomienda (aunque no es obligatorio) colocar todas las sentencias `import` al principio del módulo importador.

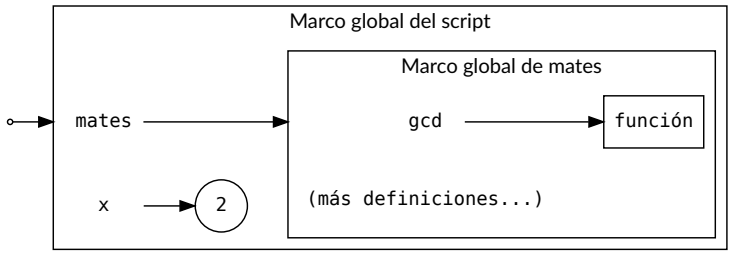
Se puede importar un módulo dándole al mismo tiempo otro nombre dentro del marco actual, usando la sentencia `import` con la palabra clave `as`.

Por ejemplo:

```
import math as mates
```

La sentencia anterior importa el módulo `math` dentro del módulo actual pero con el nombre `mates` en lugar del `math` original. Por tanto, para usar la función `gcd` como en el ejemplo anterior usaremos:

```
x = mates.gcd(16, 6)
```



Resultado de ejecutar las dos últimas líneas anteriores

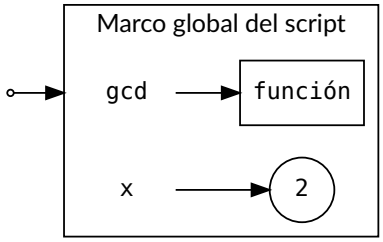
Existe una variante de la sentencia `import` que nos permite importar directamente las definiciones de un módulo en lugar del propio módulo. Para ello, se usa la orden `from`.

Por ejemplo, para importar sólo la función `gcd` del módulo `math`, y no el módulo en sí, haremos:

```
from math import gcd
```

Por lo que ahora podemos usar la función `gcd` directamente dentro del módulo importador, sin necesidad de indicar el nombre del módulo importado:

```
x = gcd(16, 6)
```



Resultado de ejecutar las dos últimas líneas anteriores

De hecho, ahora el módulo importado no está definido en el módulo importador (es decir, que en el marco global del módulo importador no hay ninguna ligadura con el nombre del módulo importado).

En nuestro ejemplo, eso significa que el módulo `math` no existe ahora como tal en el módulo importador, es decir, que ese nombre no está definido en el ámbito del módulo importador.

Por tanto, si hacemos:

```
x = math  # error
```

da error porque no hemos importado el módulo como tal, sino sólo una de sus funciones.

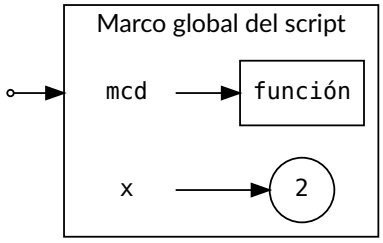
También podemos usar la palabra clave `as` con la orden `from`:

```
from math import gcd as mcd
```

De esta forma, se importa en el módulo actual la función `gcd` del módulo `math` pero llamándola `mcd`.

Por tanto, para usarla la invocaremos con su nuevo nombre:

```
x = mcd(16, 6)
```



Resultado de ejecutar las dos últimas líneas anteriores

Existe incluso una variante para importar todas las definiciones de un módulo:

```
from math import *
```

Con esta sintaxis importaremos todas las definiciones del módulo excepto aquellas cuyo nombre comience por un guión bajo (_).

Las definiciones con nombres que comienzan por _ son consideradas **privadas** o internas al módulo, lo que significa que no están concebidas para ser usadas por los usuarios del módulo y que, por tanto, no forman parte de su **interfaz**.

En general, los programadores no suelen usar esta funcionalidad ya que puede introducir todo un conjunto de definiciones desconocidas dentro del módulo importador, lo que incluso puede provocar que se «*machaquen*» definiciones ya existentes.

Para saber qué definiciones contiene un módulo, se puede usar la funcion `dir()`:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

2.3.2. Módulos como *scripts*

Un módulo puede contener sentencias ejecutables además de definiciones.

Generalmente, esas sentencias existen para inicializar el módulo.

Las sentencias de un módulo se ejecutan sólo la primera vez que se encuentra el nombre de ese módulo en una sentencia `import`.

También se ejecutan si el archivo se ejecuta como un *script*.

Cuando se ejecuta un módulo Python desde la línea de órdenes como:

```
$ python3 fact.py <argumentos>
```

se ejecutará el código del módulo como si fuera un *script* más, igual que si se hubiera importado con un `import` dentro de otro módulo, pero con la diferencia de que la variable global `__name__` contendrá el valor `"__main__"`.

Eso significa que si se añade este código al final del módulo:

```
if __name__ == "__main__":
    <sentencias>
```

el módulo podrá funcionar como un *script* independiente.

Por ejemplo, supongamos el siguiente módulo `fact.py`:

```
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n - 1)

if __name__ == "__main__":
    import sys
    print(fac(int(sys.argv[1])))
```

Este módulo se podrá usar como un *script* separado o como un módulo que se pueda importar dentro de otro.

Si se usa como *script*, podremos llamarlo desde la línea de órdenes del sistema operativo:

```
$ python3 fac.py 4
24
```

Y si importamos el módulo dentro de otro, el código del último `if` no se ejecutará, por lo que sólo se incorporará la definición de la función `fac`.

3. Criterios de descomposición modular

3.1. Introducción

No existe una única forma de descomponer un programa en módulos (entiendo aquí por *módulo* cualquier parte del programa en sentido amplio, incluyendo una simple función).

Las diferentes formas de dividir el sistema en módulos traen consigo diferentes requisitos de comunicación y coordinación entre las personas (o equipos) que trabajan en esos módulos, y ayudan a obtener los beneficios descritos anteriormente en mayor o menor medida.

Nos interesa responder a las siguientes preguntas:

- ¿Qué criterios se deben seguir para dividir el programa en módulos?
- ¿Qué módulos debe tener nuestro programa?
- ¿Cuántos módulos debe tener nuestro programa?

- ¿De qué tamaño deben ser los módulos?

3.2. Tamaño y número

Se supone que, si seguimos al pie de la letra la estrategia de diseño basada en la división de problemas, sería posible concluir que si el programa se dividiera indefinidamente, cada vez se necesitaría menos esfuerzo hasta llegar a cero.

Evidentemente, esto no es así, ya que hay otras fuerzas que entran en juego.

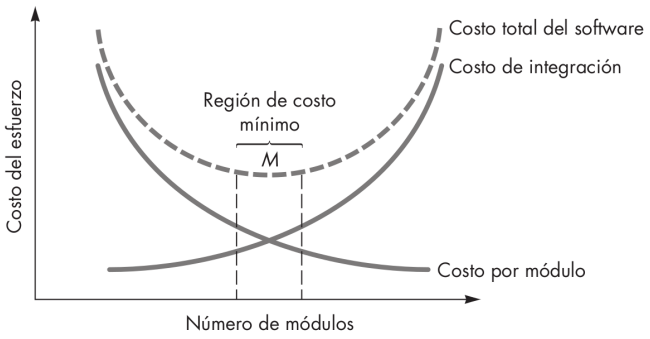
El coste de desarrollar un módulo individual disminuye conforme aumenta el número de módulos.

Dado el mismo conjunto de requisitos funcionales, cuantos más módulos hay, más pequeños son.

Sin embargo, cuantos más módulos hay, más cuesta integrarlos.

El tamaño de cada módulo debe ser el adecuado: si es demasiado grande, será difícil hacer cambios en él; si es demasiado pequeño, no merecerá la pena tratarlo como un módulo, sino más bien como parte de otros módulos.

El valor *M* es el número de módulos ideal, ya que reduce el coste total del desarrollo.



Esfuerzo frente al número de módulos

Las curvas de la figura anterior constituyen una guía útil al considerar la modularidad.

Debemos evitar hacer pocos o muchos módulos para así permanecer en la cercanía de M.

Pero, ¿cómo saber cuál es la cercanía de M? ¿Cómo de modular debe hacerse el programa?

Debe hacerse un diseño con módulos, de manera que el desarrollo pueda planearse con más facilidad, que los cambios se realicen con más facilidad, que las pruebas y la depuración se efectúen con mayor eficiencia y que el mantenimiento a largo plazo se lleve a cabo sin efectos indeseados de importancia.

Para ello nos basaremos en los siguientes **criterios**.

3.3. Abstracción

La **abstracción** es un proceso mental que se basa en estudiar un aspecto del problema a un determinado nivel centrándose en lo esencial e ignorando momentáneamente los detalles que no son importantes en este nivel.

Igualmente, nos permite comprender la esencia de un subsistema sin tener que conocer detalles innecesarios del mismo.

La utilización de la abstracción también permite trabajar con conceptos y términos que son familiares en el entorno del problema sin tener que transformarlos en una estructura no familiar.

La abstracción se usa principalmente como una técnica de **manejo y control de la complejidad**.

Cuando se considera una solución modular a cualquier problema, se pueden definir varios **niveles de abstracción**:

- En niveles **más altos** de abstracción, se enuncia una solución en términos más generales usando el lenguaje del entorno del problema.

  A estos niveles hay menos elementos de información, pero más grandes e importantes.

- En niveles **más bajos** de abstracción se da una descripción más detallada de la solución.

  A estos niveles se revelan más detalles, aparecen más elementos y se aumenta la cantidad de información con la que tenemos que trabajar.

La barrera de separación entre un nivel de abstracción y su inmediatamente inferior es la diferencia entre el *qué* y el *cómo*:

- Cuando estudiamos un concepto a un determinado nivel de abstracción, estudiamos *qué* hace.
- Cuando bajamos al nivel inmediatamente inferior, pasamos a estudiar *cómo* lo hace.

Esta división o separación puede continuar en niveles inferiores, de forma que siempre puede considerarse que cualquier nivel responde al *qué* y el nivel siguiente responde al *cómo*.

Recordemos que un módulo tiene siempre un doble punto de vista:

- El punto de vista del *creador* o implementador del módulo.
- El punto de vista del *usuario* del módulo.

> La abstracción nos ayuda a **definir qué módulos constituyen nuestro programa** considerando la relación que se establece entre los *creadores* y los *usuarios* de los módulos.

Esto es así porque **los usuarios de un módulo quieren usar a éste como una abstracción**: sabiendo *qué* hace (su función) pero sin necesidad de saber *cómo* lo hace (sus detalles internos).

El responsable del *cómo* es únicamente el **creador** del módulo.

Los módulos definidos como abstracciones son más fáciles de usar, diseñar y mantener.

3.4. Ocultación de información

David Parnas introdujo el **principio de ocultación de información** en 1972.

Afirmó que el criterio principal para la modularización de un sistema debe ser la **ocultación de *decisiones de diseño* complejas o que puedan cambiar en el futuro**, es decir, que los módulos se deben caracterizar por ocultar *decisiones de diseño* a los demás módulos.

> Por tanto: todos los elementos que necesiten conocer las mismas *decisiones de diseño*, deben pertenecer al mismo módulo.

Al ocultar la información de esa manera se evita que los usuarios de un módulo necesiten de un conocimiento íntimo del diseño interno del mismo para poder usarlo, y los aísla de los posibles efectos de cambiar esas decisiones posteriormente.

Implica que la modularidad efectiva se logra **definiendo un conjunto de módulos independientes que intercambien sólo aquella información estrictamente necesaria para que el programa funcione**.

Dicho de otra forma:

- **Para que un módulo *A* pueda usar a otro *B*, *A* tiene que conocer de *B* lo menos posible**, lo imprescindible.

  > El uso del módulo debe realizarse únicamente por medio de **interfaces** bien definidas que no cambien (o cambien poco) con el tiempo y que no expongan detalles internos al exterior.

- Por tanto, *B* debe **ocultar** al exterior sus detalles internos de **implementación** y exponer sólo lo necesario para que otros lo puedan utilizar.

  Ésto aísla a los usuarios de los posibles cambios internos que pueda haber posteriormente en *B*.

> Es decir: cada módulo debe ser una **caja negra** recelosa de su privacidad que tiene «aversión» por exponer sus interioridades a los demás.

La **abstracción** y la **ocultación de información** se complementan:

- La **ocultación de información** es un **principio de diseño** que se basa en que los módulos deben ocultar a los demás módulos sus decisiones de diseño y exponer sólo la información estrictamente necesaria para que los demás puedan usarlos.
- La **abstracción** puede usarse como una **técnica de diseño** que nos ayuda a cumplir con el principio de ocultación de información, porque nos permite descomponer el programa en módulos y **nos ayuda a identificar qué detalles hay que ocultar** y qué información hay que exponer a los demás.

Al **usuario** de un módulo…

- … le interesa la **abstracción** porque le permite usar el módulo sabiendo únicamente *qué* hace sin tener que saber *cómo* lo hace.
- … le interesa la **ocultación de información** porque cuanta menos información necesite conocer para usar el módulo, más fácil y cómodo le resultará usarlo.

Al **creador** de un módulo…

- … le interesa la **abstracción** como técnica porque le ayuda a determinar qué información debe ocultar su módulo al exterior.

- … le interesa la **ocultación de información** porque cuantos más detalles necesiten conocer los usuarios para poder usar su módulo, menos libertad dentrá de poder cambiar esos detalles en el futuro (cuando lo necesite o cuando lo desee) sin afectar a los usuarios de su módulo.

3.5. Independencia funcional

La independencia funcional se logra desarrollando módulos de manera que cada módulo resuelva una funcionalidad específica y tenga una interfaz sencilla cuando se vea desde otras partes de del programa (idealmente, mediante paso de parámetros).

- De hecho, la interfaz del módulo debe estar destinada únicamente a cumplir con esa funcionalidad.

Al limitar su objetivo, el módulo necesita menos ayuda de otros módulos.

Y por eso el módulo debe ser tan independiente como sea posible del resto de los módulos del programa, es decir, que dependa lo menos posible de lo que hagan otros módulos, y también que dependa lo menos posible de los datos que puedan facilitarle otros módulos.

> Dicho de otra forma: los módulos deben centrarse en resolver un problema concreto (ser «monotemáticos»), deben ser «antipáticos» y tener «aversión» a relacionarse con otros módulos.

Los módulos independientes son más fáciles de desarrollar porque la función del programa se subdivide y las interfaces se simplifican, por lo que se pueden desarrollar por separado.

Los módulos independientes son más fáciles de mantener y probar porque los efectos secundarios causados por el diseño o por la modificación del código son más limitados, se reduce la propagación de errores y es posible obtener módulos reutilizables.

De esta forma, la mayor parte de los cambios y mejoras que haya que hacer al programa implicarán modificar sólo un módulo o un número muy pequeño de ellos.

Es un objetivo a alcanzar para obtener una modularidad efectiva.

La independencia funcional se mide usando dos métricas: la **cohesión** y el **acoplamiento**.

El objetivo de la independencia funcional es **maximizar la cohesión y minimizar el acoplamiento**.

3.5.1. Cohesión

La **cohesión** mide la fuerza con la que se relacionan los componentes de un módulo.

Cuanto más cohesivo sea un módulo, mejor será nuestro diseño modular.

Un módulo cohesivo realiza una sola función, por lo que requiere interactuar poco con otros componentes en otras partes del programa.

En un módulo cohesivo, sus componentes están fuertemente relacionados entre sí y pertenencen al módulo por una razón lógica (no están ahí por casualidad), es decir, todos cooperan para alcanzar un objetivo común que es la función del módulo.

Un módulo cohesivo mantiene unidos (*atrae*) los componentes que están relacionados entre ellos y mantiene fuera (*repele*) el resto.

En pocas palabras, un módulo cohesivo debe tener un único objetivo, y todos los elementos que lo componen deben contrubuir a alcanzar dicho objetivo.

Aunque siempre debe tratarse de lograr mucha cohesión (por ejemplo, una sola tarea), con frecuencia es necesario y aconsejable hacer que un módulo realice varias tareas, siempre que contribuyan a una misma finalidad lógica.

Sin embargo, para lograr un buen diseño hay que evitar módulos que llevan a cabo funciones no relacionadas.

La siguiente es una escala de grados de cohesión, ordenada de mayor a menor:

- Cohesión funcional
- Cohesión secuencial
- Cohesión de comunicación

  [Hasta aquí, los módulos se consideran **cajas negras**.]
- Cohesión procedimental
- Cohesión temporal
- Cohesión lógica
- Cohesión coincidental

No hace falta determinar con precisión qué cohesión tenemos. Lo importante es intentar conseguir una cohesión alta y reconocer cuándo hay poca cohesión para modificar el diseño y conseguir una mayor independencia funcional.

Cohesión funcional: se da cuando los componentes del módulo pertenecen al mismo porque todos contribuyen a una tarea única y bien definida del módulo.

Cohesión secuencial: se da cuando los componentes del módulo pertenecen al mismo porque la salida de uno es la entrada del otro, como en una cadena de montaje (por ejemplo, una función que lee datos de un archivo y los procesa).

Cohesión de comunicación: se da cuando los componentes del módulo pertenecen al mismo porque trabajan sobre los mismos datos (por ejemplo, un módulo que procesa números racionales).

Cohesión procedimental: se da cuando los componentes del módulo pertenecen al mismo porque siguen una cierta secuencia de ejecución (por ejemplo, una función que comprueba los permisos de un archivo y después lo abre).

Cohesión temporal: se da cuando los componentes del módulo pertenecen al mismo porque se ejecutan en el mismo momento (por ejemplo, una función que se dispara cuando se produce un error, abriría un archivo, crearía un registro de error y notificaría al usuario).

Cohesión lógica: se da cuando los componentes del módulo pertenecen al mismo porque pertenencen a la misma categoría lógica aunque son esencialmente distintos (por ejemplo, un módulo que agrupe las funciones de manejo del teclado o el ratón).

Cohesión coincidental: se da cuando los componentes del módulo pertenecen al mismo por casualidad o por razones arbitrarias, es decir, que la única razón por la que se encuentran en el mismo módulo es porque se han agrupado juntos (por ejemplo, un módulo de «utilidades»).

3.5.2. Acoplamiento

El **acoplamiento** es una medida del grado de interdependencia entre los módulos de un programa.

Dicho de otra forma, es la fuerza con la que se relacionan los módulos de un programa.

El acoplamiento depende de:

- La complejidad de la interfaz entre los módulos
- El punto en el que se entra o se hace referencia a un módulo
- Los datos que se pasan a través de la interfaz

Lo deseable es tener módulos con poco acoplamiento, es decir, módulos que dependan poco unos de otros.

De esta forma obtenemos programas más fáciles de entender y menos propensos al *efecto ola*, que ocurre cuando se dan errores en un sitio y se propagan por todo el programa.

Los programas con alto acoplamiento tienden a presentar los siguientes problemas:

- Un cambio en un módulo normalmente obliga a cambiar otros módulos (consecuencia del *efecto ola*).
- Requiere más esfuerzo integrar los módulos del programa ya que dependen mucho unos de otros.
- Un módulo particular resulta más difícil de reutilizar o probar debido a que hay que incluir en el lote a los módulos de los que depende éste (no se puede reutilizar o probar por separado).

La siguiente es una escala de grados de acoplamiento, ordenada de mayor a menor:

- Acoplamiento por contenido
- Acoplamiento común
- Acoplamiento externo
- Acoplamiento de control
- Acoplamiento por estampado
- Acoplamiento de datos
- Sin acoplamiento

No hace falta determinar con precisión qué cohesión tenemos. Lo importante es intentar conseguir una cohesión alta y reconocer cuándo hay poca cohesión para modificar el diseño y conseguir una mayor independencia funcional.

Acoplamiento por contenido: ocurre cuando un módulo modifica o se apoya en el funcionamiento interno de otro módulo (por ejemplo, accediendo a datos locales del otro módulo). Cambiar la forma en que el segundo módulo produce los datos obligará a cambiar el módulo dependiente.

Acoplamiento común: ocurre cuando dos módulos comparten los mismos datos globales. Cambiar el recurso compartido obligará a cambiar todos los módulos que lo usen.

Acoplamiento externo: ocurre cuando dos módulos comparten un formato de datos impuesto externamente, un protocolo de comunicación o una interfaz de dispositivo de entrada/salida.

Acoplamiento de control: ocurre cuando un módulo controla el flujo de ejecución del otro (por ejemplo, pasándole un *conmutador* booleano).

Acoplamiento por estampado: ocurre cuando los módulos comparten una estructura de datos compuesta y usan sólo una parte de ella, posiblemente una parte diferente. Esto podría llevar a cambiar la forma en la que un módulo lee un dato compuesto debido a que un elemento que el módulo no necesita ha sido modificado.

Acoplamiento de datos: ocurre cuando los módulos comparten datos entre ellos (por ejemplo, parámetros). Cada dato es una pieza elemental y dicho parámetro es la única información compartida (por ejemplo, pasando un entero a una función que calcula una raíz cuadrada).

Sin acoplamiento: ocurre cuando los módulos no se comunican para nada uno con otro.

3.6. Reusabilidad

La **reusabilidad** es un factor de calidad del software que se puede aplicar tambien a sus componentes o módulos.

Un módulo es **reusable** cuando puede aprovecharse para ser utilizado (tal cual o con muy poca modificación) en varios programas.

A la hora de diseñar módulos (o de descomponer un programa en módulos) nos interesa que los módulos resultantes sean cuanto más reusables mejor.

Para ello, el módulo en cuestión debe ser lo suficientemente general y resolver un problema patrón que sea suficientemente común y se pueda encontrar en varios contextos y programas diferentes.

Además, para aumentar la reusabilidad, es conveniente que el módulo tenga un bajo acoplamiento y que, por tanto, no dependa de otros módulos del programa.

Esos módulos incluso podrían luego formar parte de una *biblioteca* o *repositorio* de módulos y ponerlos a disposición de los programadores para que puedan usarlos en sus programas.

A día de hoy, el desarrollo de programas se basa en gran medida en seleccionar y utilizar módulos (o bibliotecas, *librerías* o *paquetes*) desarrollados por terceros o reutilizados de otros programas elaborados por nosotros mismos anteriormente.

Es decir: la programación se ha convertido en una actividad consistente principalmente en ir combinando componentes intercambiables.

Eso nos permite acortar el tiempo de desarrollo porque podemos construir un programa a base de ir ensamblando módulos reusables como si fueran las piezas del engranaje de una máquina.

4. Abstracción de datos

4.1. 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.

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.

4.2. Especificaciones

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⟩
```

4.2.1. 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*.

4.2.2. Ejemplos

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

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

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* → *pila*
`vacia?` : *pila* → $\mathfrak{B}$
var
p : *pila*; *x* : *elemento*
ecuaciones
`cima`(`apilar`(*p*, *x*)) ≐ *x*
`desapilar`(`apilar`(*p*, *x*)) ≐ *p*
`vacia?`(`pvacia`) ≐ *V*
`vacia?`(`apilar`(*p*, *x*)) ≐ *F*
`cima`(`pvacia`) ≐ *error*
`desapilar`(`pvacia`) ≐ *error*

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

```
p = pvacia()        # crea vacía
p = apilar(p, 4)    # apila valor 4
p = apilar(p, 8)    # apila valor 8
print(vacia(p))     # imprime False
print(cima(p))      # imprime 8
print(desapilar(p)) # imprime 8
print(desapilar(p)) # imprime 4
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).

Y los **números racionales** se podrían especificar así:

espec *rac*
operaciones
parcial `racional` : $\mathbb{Z} \times \mathbb{Z} \to$ *rac*
`numer` : *rac* → $\mathbb{Z}$
`denom` : *rac* → $\mathbb{Z}$
`suma` : *rac* × *rac* → *rac*
`mult` : *rac* × *rac* → *rac*
`iguales?` : *rac* × *rac* → $\mathfrak{B}$
`imprimir` : *rac* → $\emptyset$
var
r : *rac*; $n, d, n_1, n_2, d_1, d_2 : \mathbb{Z}$
ecuaciones
`numer`(`racional`(n, d)) ≐ n
`denom`(`racional`(n, d)) ≐ d
`suma`(`racional`(n_1, d_1), `racional`(n_2, d_2)) ≐ `racional`($n_1 \cdot d_2 + n_2 \cdot d_1, d_1 \cdot d_2$)
`mult`(`racional`(n_1, d_1), `racional`(n_2, d_2)) ≐ `racional`($n_1 \cdot n_2, d_1 \cdot d_2$)
`iguales?`(`racional`(n_1, d_1), `racional`(n_2, d_2)) ≐ $n_1 \cdot d_2 = n_2 \cdot d_1$
`imprimir`(*r*) { imprime el racional *r* }
`racional`(n, 0) ≐ *error*

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*.

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`.

Aun así, si definimos estas tres funciones, podríamos sumar, multiplicar, imprimir y comprobar la igualdad de números racionales, con lo que podemos definir las funciones `suma`, `mult`, `imprimir` e `iguales?` en función de `racional`, `numer` y `denom`.

```
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='')
```

4.3. 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.

4.4. Barreras 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.

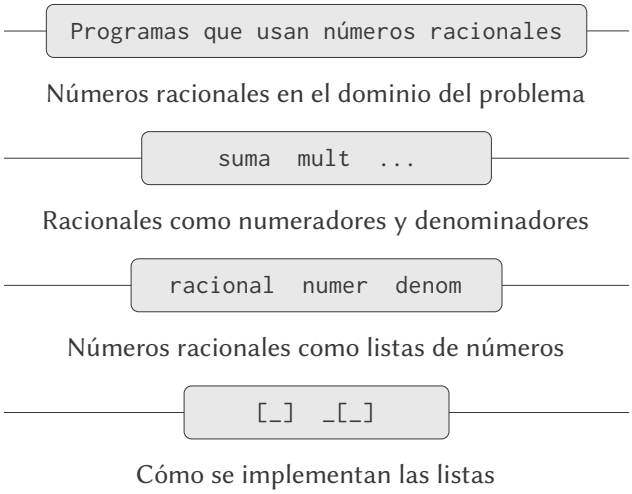
Al obligar a usar los datos únicamente a través de sus operaciones, es mucho más fácil cambiar luego la representación interna de los datos abstractos o la implementación de las operaciones básicas sin tener que cambiar el resto del programa.

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 `_[_]`

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.

4.5. Las propiedades de los datos

Las barreras de abstracción determinan la forma en la que pensamos sobre los datos.

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 sirven para representar adecuadamente a los números racionales. Además, tenemos 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 .

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 también 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.

4.5.1. Representación funcional

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)
```

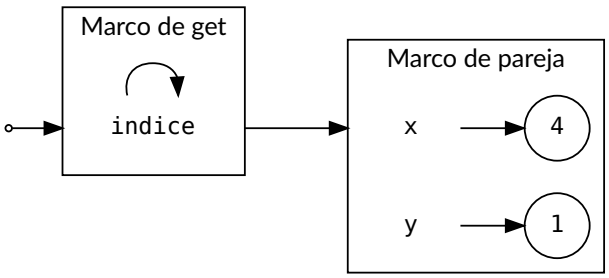
Con esta implementación, podemos crear y manipular parejas:

```
>>> 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 en la función `get` al llamar a `pareja(4, 1)`

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 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.

El hecho de ver aquí la **representación funcional** de una pareja no es porque Python realmente trabaje de esta manera (las listas en Python se implementan internamente de otra forma, por razones de eficiencia), sino porque podría trabajar de esta manera.

La práctica de la abstracción de datos nos permite cambiar fácilmente unas representaciones por otras.

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.

4.5.2. Estado interno

Ciertos datos pueden tener **estado interno**, lo que significa que su contenido 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.

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
```

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.

4.6. La metáfora del objeto

Al principio, distinguíamos entre funciones y datos: las funciones realizan operaciones sobre los datos y éstos esperan pasivamente a que se opere con ellos.

Cuando empezamos a representar a los datos con funciones, vimos que los datos también pueden encapsular **comportamiento**.

Esos datos ahora representan información, pero también **se comportan** como las cosas que representan.

Por tanto, los datos ahora saben cómo reaccionar ante los mensajes que reciben cuando las demás partes del programa les envían mensajes.

Esta forma de ver a los datos como objetos activos que se relacionan entre sí y que son capaces de reaccionar y cambiar su estado interno en función de los mensajes que reciben, da lugar a todo un nuevo paradigma de programación llamado **programación orientada a objetos**.

4.7. 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

Pressman, Roger S, Darrel Ince, Rafael Ojeda Martín, and Luis Joyanes Aguilar. 2004. *Ingeniería Del Software: Un Enfoque Práctico*. Madrid: McGraw-Hill.

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