

Programación modular

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2026/02/20 a las 21:16:00

Índice

1. Introducción	2
1.1. Modularidad	2
1.2. Beneficios de la programación modular	7
2. Diseño modular	7
2.1. Creadores y usuarios	7
2.2. Partes de un módulo	9
2.2.1. Interfaz	9
2.2.2. Implementación	11
2.3. Diagramas de estructura	12
3. Programación modular en Python	12
3.1. Scripts como módulos	12
3.1.1. Docstrings	14
3.2. Importación de módulos	15
3.3. Módulos como scripts	25
3.4. La librería estándar	26
3.5. Paquetes	26
4. Criterios de descomposición modular	32
4.1. Introducción	32
4.2. Tamaño y número	32
4.3. Abstracción	33
4.4. Ocultación de información	34
4.5. Independencia funcional	38
4.5.1. Cohesión	39
4.5.2. Acoplamiento	40
4.6. Reusabilidad	41
4.6.1. Diseño ascendente	42
4.6.2. Estrategia sándwich	43
4.7. Principios SOLID	44

1. Introducción

1.1. Modularidad

La **programación modular** es una técnica de programación que consiste en descomponer nuestro programa en partes llamadas **módulos** y escribir cada módulo por separado.

- El concepto de *módulo* hay que entenderlo en sentido amplio: cualquier parte de un programa se podría considerar «módulo» si cumple unas mínimas condiciones.

Equivale a la técnica clásica de resolución de problemas basada en:

1. Descomponer un problema en subproblemas.
2. Resolver cada subproblema por separado.
3. Combinar las soluciones para así obtener la solución al problema original.

La **modularidad** es la propiedad que tienen los programas escritos de forma modular, es decir, como una colección de módulos que se utilizan entre sí.

Las técnicas de modularidad y descomposición de problemas se aplican en la mayoría de las disciplinas científicas e industriales.

Por ejemplo, el diseño y fabricación de un coche resulta complicado si vemos a éste como un todo (un conglomerado de piezas todas juntas).

Además, desde ese punto de vista sería difícil poder reparar una avería, ya que la sustitución de cualquier pieza podría afectar a cualquier otra parte del coche, a menudo de formas poco evidentes.

En cambio, el coche resulta mucho más fácil de entender si lo descomponemos en partes, estudiamos cada parte por separado y definimos claramente cómo interactúa cada parte con las demás.

Esas partes (que aquí llamamos módulos) son componentes más o menos independientes que interactúan con otros componentes de una manera bien definida. Por ejemplo:

- Frenos
- Luces
- Climatización
- Dirección
- Motor
- Carrocería

El objetivo es convertir la programación es una tarea que consista en ir fabricando y ensamblando bloques constructivos que tengan sentido por sí mismos, que lleven a cabo una función concreta y que, al combinarlos adecuadamente, nos dé como resultado el programa que queremos desarrollar.

Para alcanzar ese objetivo, 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, como hace también la *abstracción* (que, de hecho, puede servir como guía para la modularización).

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**.

Ambos conceptos son combinables, como veremos luego.

A **nivel práctico**, la modularidad nos ofrece una herramienta que nos permite partir el código del 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 es intelectualmente más difícil de entender.

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

Esto, además, nos va a permitir que un programa pueda ser desarrollado conjuntamente por varias personas al mismo tiempo, cada una de ellas responsabilizándose de escribir uno o varios módulos del mismo programa.

Esos módulos se podrán escribir de forma más o menos independiente aunque, por supuesto, estarán relacionados entre sí, ya que unos consumirán los servicios que proporcionan otros.

Los módulos contienen **definiciones internas** llamadas **miembros**, entre las que pueden estar:

- Subprogramas (funciones y procedimientos).
- Definiciones de tipos de datos.
- Variables.
- Constantes.
- Otros módulos (submódulos).

Todos esos elementos internos del módulo son manipulables desde dentro del mismo y también pueden estar accesibles desde fuera, disponibles para su uso por parte de otros módulos del programa.

Gracias a sus variables locales, un módulo almacena y recuerda su propio **estado interno**, el cual puede cambiar durante la ejecución del programa.

Para que todo esto funcione, los módulos deben introducir su propio **ámbito léxico**, deben estar **encapsulados** y deben tener su propio **espacio de nombres**:

- Los módulos introducen su propio **ámbito léxico**, por lo que las definiciones que se ejecuten dentro del módulo serán **locales** a éste.

Además, el ámbito del módulo representa el **ámbito global** para todas las definiciones que se ejecuten dentro del módulo.

- Los módulos están **encapsulados**, lo que hace que sea aún más independiente de los demás módulos que forman el programa, ya que un elemento local al módulo sólo será visible directamente dentro de éste, y sólo se podrá ver desde fuera si se **exporta**.
- Finalmente, los módulos tienen su propio **espacio de nombres** separado del resto, donde se almacenan sus ligaduras locales, es decir, las ligaduras creadas dentro de ese módulo.

Así, los elementos locales al módulo pertenecen al propio módulo, y puede haber dos elementos distintos con el mismo nombre en diferentes módulos de un mismo programa, lo que evita el *name clash*.

El hecho de que los módulos estén encapsulados facilita la escritura de cada módulo por separado y su integración posterior en el mismo programa, ya que el programador de un módulo no se tiene que preocupar por si casualmente usó el mismo nombre que ha usado otro programador al escribir su módulo.

Los **módulos** son *cápsulas* que:

- Permiten seleccionar qué se **exporta** a otros módulos (lo que constituye su **interfaz**) y qué se **importa** de otros módulos (sus **dependencias**).
- Permiten la **descomposición física del código en archivos separados**, para que se puedan escribir de forma más o menos independiente unos de otros.
- Representan una **descomposición lógica** del programa.

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.

Según esa definición, podría pensarse que **un subprograma es un ejemplo de módulo**.

Sin embargo, un subprograma es un ejemplo bastante pobre de módulo, ya que, según lo que hemos visto hasta ahora:

- Un subprograma no exporta nada al exterior.
- Un subprograma no puede usar elementos locales de otros subprogramas.
- Un subprograma no tiene estado interno que se conserve y recuerde durante la ejecución del programa.

(Si bien en temas posteriores veremos que algunas de estas limitaciones se pueden superar con el uso de *clausuras*.)

Además, descomponer un programa en partes usando únicamente como criterio la descomposición en subprogramas **no resultaría adecuado en general**, ya que:

- No habría *descomposición física*, salvo que se coloque cada función en un archivo separado, lo que resulta poco práctico.
- En la mayoría de los casos, nos encontramos con varios subprogramas que no actúan por separado, sino de forma conjunta entre ellas formando un todo interrelacionado.

Por ejemplo, existen lenguajes procedimentales (como C) donde podemos usar y crear **bibliotecas de funciones**, que son colecciones agrupadas de funciones reutilizables (y, normalmente, ya compiladas) que se pueden invocar desde el código de nuestro programa.

Esas bibliotecas se distribuyen apropiadamente empaquetadas de forma que luego se puedan enlazar en tiempo de compilación con el resto de nuestro código para formar el programa ejecutable final.

Pero aunque podamos encapsular varias funciones juntas en una biblioteca, en general eso tampoco resulta suficiente para llamar «módulo» a ese paquete de funciones.

Al final, lo que es y no es un módulo depende mucho también de las construcciones que el lenguaje de programación nos proporciona.

Por ejemplo, lenguajes como Haskell, OCaml, Turbo Pascal o Modula-2 tienen sus módulos, mientras que las «clases» de los lenguajes orientados a objetos también son muy buenos ejemplos de módulos.

La programación estructurada y la programación procedural son suficientes para la denominada **programación a pequeña escala (Programming In The Small, PITs)**.

Se trata de programas que un solo programador puede programar y comprender fácilmente en su totalidad.

Sin embargo, no son técnicas o paradigmas lo suficientemente generales ni apropiadas como para escribir programas muy grandes.

Estos programas, escritos por muchas personas, deben constar de módulos que puedan desarrollarse y probarse independientemente de otros módulos, en paralelo, por varios programadores que se reparten simultáneamente el trabajo.

Este tipo de programación se denomina **programación a gran escala (Programming In The Large, PITL)**.

La programación a gran escala se centra en programas que no son comprensibles para una sola persona y que son desarrollados por equipos de programadores.

En este nivel, los programas deben constar de módulos que puedan escribirse, compilarse y probarse independientemente de otros módulos (o lo más posible, al menos).

Para ello, un módulo debe tener un único propósito y una interfaz estrecha con otros módulos.

Además, es conveniente diseñar módulos que sean *reutilizables* (es decir, que puedan incorporarse a muchos programas) y modificables sin forzar cambios en otros módulos.

Todo es un camino hacia la obtención de componentes constructivos cada vez más grandes y abstractos.

Por ejemplo, en Python tendríamos (en orden de menor a mayor nivel de abstracción y complejidad):

1. Expresiones y abstracciones lambda.
2. Sentencias.
3. Estructuras de control.
4. Funciones imperativas.
5. Clases.
6. Módulos.
7. Paquetes.

Cada componente constructivo puede contener, recursivamente, elementos del mismo nivel o inferior, pero no de un nivel superior.

Estos componentes constructivos ayudan a organizar el programa en varios niveles de abstracción y complejidad, y nos permiten escribir programas cada vez más grandes y complejos porque nos permite aislar y estudiar cada parte y cada nivel por separado.

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

Tendríamos funciones para crear racionales, para sumarlos, para multiplicarlos, para simplificarlos, etc.

Todas esas funciones trabajarían conjuntamente, incluso usándose unas a otras, y actuarían sobre colecciones de datos que representarían números racionales de una forma apropiada.

Por consiguiente, aunque resulta muy apropiado considerar cada función anterior por separado, es más conveniente considerarlas como un todo: el *módulo de manipulación de números racionales*.

Otro ejemplo: supongamos un módulo encargado de realizar operaciones trigonométricas sobre ángulos.

Esos ángulos serían números reales que podrían representar grados o radianes.

Ese módulo contendría las funciones encargadas de calcular el seno, coseno, tangente, etc. de un ángulo pasado como parámetro a cada función.

Para ello, el módulo podría contener un dato (almacenado en una variable dentro del módulo) que indicará si las funciones anteriores deben interpretar los ángulos como grados o como radianes.

Ese dato constituiría el estado interno del módulo, al ser un valor que se almacena dentro del mismo módulo (pertenece a éste) y puede cambiar su comportamiento dependiendo del valor que tenga el dato.

Además, probablemente también necesite conocer el valor de π , que podría almacenar el módulo internamente como una constante.

En un hipotético programa que usara los dos módulos anteriores (de números racionales y de ángulos), ambos serían independientes y estarían separados en archivos distintos, lo que tendría estas ventajas, entre otras:

- Es más fácil localizar y corregir un fallo del programa. Por ejemplo, si se encuentra un fallo en las operaciones con números racionales, sería más fácil localizarlo, ya que el principal sospechoso sería el módulo de los números racionales.
- Cada módulo se podría programar y poner a punto por separado, incluso al mismo tiempo si ponemos a trabajar en cada uno a equipos distintos de programadores.
- El desarrollo del resto del programa también podría ir en paralelo al de estos dos módulos, por las mismas razones, lo que reduce el tiempo de desarrollo.

De lo dicho hasta ahora se deducen varias **conclusiones importantes**:

- Un módulo es una parte de un programa.
- Los módulos nos permiten descomponer el programa en partes más o menos independientes y manejables por separado.

- Una **función** cumple con la definición de «módulo» pero, en general, **en la práctica no es una buena candidata** para ser considerada un módulo por sí sola.
- Los módulos, contienen colecciones de **funciones interrelacionadas** y otros posibles elementos, como **datos** en forma de **constants y variables** manipulables desde el interior del módulo y (posiblemente) también desde el exterior del mismo.
- Las variables del módulo constituyen el **estado interno** del módulo.
- Se puede controlar qué elementos se **exportan** al exterior del módulo, y qué elementos se **importan** de otros módulos.
- En la práctica, 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 distinto al mismo tiempo con poca necesidad de comunicación entre ellos.

Se **mejora la fiabilidad y robustez** del producto resultante, porque los cambios 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.

Reusabilidad, porque facilita el que un módulo de un programa pueda ser aprovechado (con poca o ninguna alteración) en otros programas.

En lenguajes compilados, la descomposición física del código del programa en varios archivos permite la **compilación por separado**, lo que acelera significativamente la compilación de programas grandes cuando hay cambios en pocos módulos (los módulos que no han cambiado y que ya están compilados no necesitan una recompilación).

2. Diseño modular

2.1. Creadores y usuarios

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 del módulo, necesarios para que éste funcione. Esos detalles internos constituyen la **implementación** del módulo.
- Los **usuarios** del módulo son los programadores que desean usar ese módulo en sus programas. También se les llama así a los módulos que usan a ese módulo (lo necesitan para funcionar) en un programa.

A la parte del módulo que es accesible directamente desde fuera del mismo se le denomina la **interfaz** del módulo.

Decimos que los módulos **exportan su interfaz** al resto de los módulos.

Por otra parte, un módulo puede hacer uso de otros módulos (que se denominan **sus dependencias**) y para ello debe **importarlos** previamente.

En terminología de programación modular, hablamos entonces de la existencia de dos tipos de módulos:

- El que usa a otro módulo es el módulo «**usuario**», «**cliente**», «**consumidor**» o «**importador**».
- El que es usado por otro módulo es el módulo «**usado**», «**servidor**», «**consumido**» o «**importado**».

Cuando el *módulo usuario* importa al *módulo importado*, tiene acceso a los elementos exportados por este último a través de su interfaz, lo que le permite consumir los servicios que ese módulo importado proporciona.

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

Esos usuarios consumirán los servicios del módulo a través de su interfaz, la cual oculta a los usuarios los detalles internos de funcionamiento que no es necesario conocer para usar el módulo.

Esos detalles internos forman la *implementación* del módulo y sólo son interesantes para (y sólo deben ser conocidos por) el **creador** del módulo.

Por tanto, nos interesa que los módulos actúen como **cajas negras**.

Por supuesto, un módulo puede usar a otros módulos y ser usado por otros módulos, todo al mismo tiempo. Por tanto, el mismo módulo puede ser importador e importado simultáneamente.

Los **miembros** o *elementos* de un módulo son aquellos elementos que forman parte del módulo.

En general, a la hora de describir las partes de un módulo, distinguimos entre **miembros públicos** y **miembros privados**:

- Los **miembros públicos** o **exportados** de un módulo son aquellos miembros que están diseñados para ser usados fuera del módulo por los usuarios del mismo.
Estos miembros (al menos, las especificaciones de los mismos, es decir, lo necesario para poder usarlos) **deben formar parte de la interfaz del módulo**.
- Los **miembros privados** o **internos** de un módulo son aquellos miembros que están diseñados para ser usados únicamente por el propio módulo y, por tanto, no deberían ser usados (ni siquiera conocidos) fuera del módulo.
Estos miembros **NO deben formar parte de la interfaz del módulo**, sino que deben ir en la *implementación* del módulo.

2.2. Partes de un módulo

Concretando lo dicho anteriormente, un módulo tendrá:

- Una **interfaz**, formada por:
 - El **nombre** del módulo.
 - Las **signaturas de sus funciones exportadas o públicas** que permiten a los usuarios consumir sus servicios, así como manipular y acceder al estado interno del módulo desde fuera del mismo.
 - Es posible que la interfaz también incluya otros elementos, como **constants públicas**.
- Una **implementación**, formada por:
 - Las **implementaciones** de sus **funciones públicas**.
 - Posibles **variables locales** al módulo (el *estado interno* de éste).
 - Posibles **funciones internas o privadas**, que no aparecen en la interfaz porque están pensadas para ser usadas sólo por el propio módulo internamente, no por otras partes del programa.
 - Otros miembros públicos o privados, como **constants**, etc.

Lo anterior es una simplificación, ya que, dependiendo de las características del lenguaje de programación utilizado, un módulo puede tener miembros pertenecientes a muchas otras categorías sintácticas, como por ejemplo:

- Tipos.
- Clases.
- Otros módulos.

Todos esos miembros pueden formar parte de la interfaz o de la implementación del módulo, dependiendo de las necesidades del mismo, y ser, por tanto, miembros públicos o privados de éste.

También se admiten otras combinaciones más complicadas. Por ejemplo, si alguno de esos miembros es público, su especificación podría formar parte de la interfaz del módulo, mientras que su implementación podría formar parte de la implementación del módulo.

2.2.1. Interfaz

La **interfaz** es la parte del módulo que es accesible directamente desde fuera del mismo.

Por tanto, es la parte a través de la cual el módulo ofrece sus servicios a sus usuarios, que sólo podrán acceder al mismo a través de su interfaz.

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

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

La interfaz es la parte del módulo que éste **exporta** a los demás módulos del programa, que son sus posibles usuarios.

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

No todas las funciones definidas en un módulo tienen por qué formar parte de la interfaz del mismo. Las que sí lo hacen son las denominadas **funciones públicas o exportadas**.

2.2.1.1. Especificación

La interfaz es un concepto puramente **sintáctico**.

Describe la sintaxis (nombre y tipos) de los elementos del módulo a los que se puede acceder desde fuera del mismo.

Pero sólo con eso, el usuario no tiene toda la información que necesita para poder usar el módulo.

Para ello, también necesita saber qué hace el módulo, y qué hacen las funciones que forman la interfaz del módulo.

La **especificación** del módulo representa toda la información que el usuario del mismo necesita conocer para poder utilizarlo.

La especificación de un módulo está formada por:

- La **interfaz** del módulo.
- La **especificación** de todas las funciones que aparecen en la interfaz.
- **Documentación** adicional que describa lo que hace el módulo y que aporte al usuario cualquier información extra que necesite saber para poder usar el módulo sin tener que conocer o acceder a partes internas del mismo.

Aquí podría incluirse documentación sobre el módulo en sí, así como sobre las funciones y las constantes públicas que aparecen en la interfaz, escrita en lenguaje natural y de fácil comprensión para el lector humano, añadiendo posibles ejemplos de uso.

Desde el punto de vista de los usuarios del módulo, las funciones son **abstracciones funcionales**, de forma que, para poder usarlas, sólo se necesita conocer las **especificaciones** de esas funciones y no sus *implementaciones* concretas (sus definiciones completas).

Recordemos que la **especificación de una función** está formada por tres partes:

- **Precondición**: la condición que se debe cumplir al llamar a la función.
- **Signatura**: los aspectos meramente sintácticos como el nombre de la función, el nombre y tipo de sus parámetros y el tipo de su valor de retorno.
- **Postcondición**: la condición que se cumplirá al finalizar su ejecución.

Por tanto, la especificación del módulo contendrá las especificaciones de las funciones públicas, pero no sus implementaciones.

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

En teoría, la interfaz podría incluir también algunas o todas las **variables locales al módulo**, de forma que el módulo usuario podría acceder y modificar directamente una variable del módulo usado.

Sin embargo, en la práctica eso no resulta apropiado, ya que:

- Cambiar el valor de una variable local a un módulo equivale a modificar una variable global (ya que existen en el ámbito global del módulo) y, por tanto, se considera un efecto lateral.
- 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.

Las constantes sí se admiten ya que nunca cambian su valor y, por tanto, están exentas de posibles efectos laterales.

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

2.2.2. Implementación

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

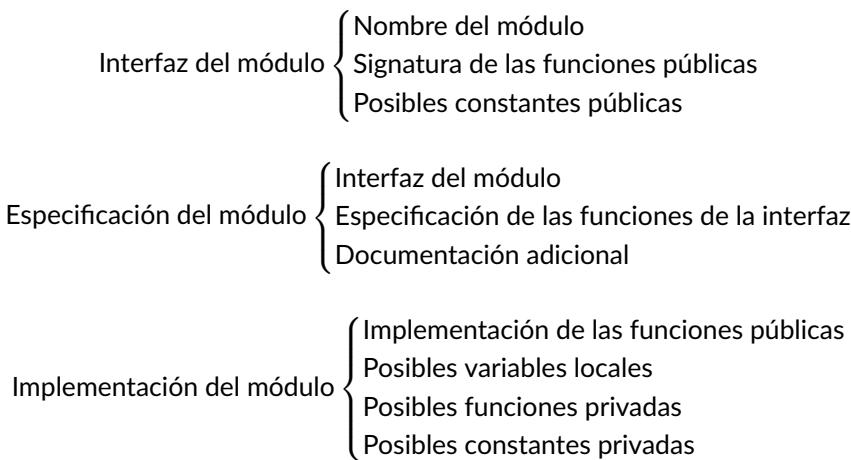
Por tanto, es la parte que los usuarios del módulo no necesitan (ni deben) conocer para poder usarlo adecuadamente.

Está formada por:

- La **implementación de las funciones** que forman la interfaz.
- Todas las **variables locales al módulo** que almacenan su estado interno, si las hay.
- Las funciones que utiliza el propio módulo internamente y que no forman parte de su interfaz (**funciones internas o privadas**), si las hay.
- Posibles **constants privadas**, usadas sólo por el propio módulo.

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

2.2.2.1. Resumen

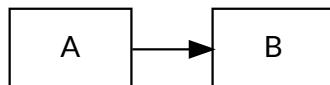


2.3. Diagramas de estructura

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 *consume*) al módulo B, o también se puede decir que el módulo A *depende* del módulo B.



A depende de B

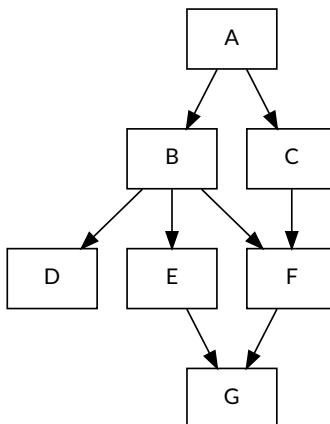


Diagrama de estructura

Al módulo A se le denomina **módulo principal**, ya que es el módulo por el que comienza la ejecución del programa.

El módulo principal depende, directa o indirectamente, de los demás módulos del programa.

No hay ningún módulo que dependa del módulo principal.

3. Programación modular en Python

3.1. Scripts como módulos

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

Los módulos contienen instrucciones: definiciones y otras sentencias.

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

Eso quiere decir que **un módulo se puede ejecutar de dos maneras**:

- **Como un script independiente**, llamándolo desde la línea de órdenes del sistema operativo (sería el **módulo principal**):

```
$ python modulo.py
```

- **Importándolo dentro de otros módulos** que quieran usar los servicios que proporciona, usando la sentencia **import** (o **from ... import**):

```
import modulo
```

Las sentencias que contiene un módulo **se ejecutan sólo la primera vez** que se encuentra el nombre de ese módulo en una sentencia **import**, o bien cuando el archivo se ejecuta como un *script*.

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

Cada módulo determina su propio **ámbito local**, que es usado como el **ámbito global** de todas las instrucciones que componen el módulo.

Además, los módulos son **espacios de nombres**.

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

Esas variables y funciones serán los **atributos** del **objeto módulo** que se creará si se importa el módulo con la sentencia **import**.

Al entrar en un módulo para ejecutar sus sentencias, se entra en un nuevo ámbito que constituirá **el ámbito global del módulo**.

Además, se creará un nuevo marco encima de la pila, que constituirá **el nuevo marco global durante la ejecución del módulo**.

Los demás marcos que pudieran existir antes de ejecutar el módulo (por ejemplo, el marco global del *script* que ha importado el módulo) seguirán más abajo en la pila, pero no serán accesibles desde el módulo actual.

Esos marcos quedan fuera del entorno y se recuperarán al finalizar la ejecución del módulo importado.

Igualmente, los ámbitos que existieran antes de importar un módulo (incluyendo el ámbito global del *script* que ha importado al módulo) quedan temporalmente invalidados al encontrarse en archivos fuente distintos, y se recuperarán al finalizar la ejecución del módulo y volver al archivo fuente anterior.

3.1.1. Docstrings

Al igual que ocurre con las funciones imperativas, los módulos también pueden llevar su **cadena de documentación (docstring)**.

La cadena de documentación es un literal de tipo cadena que aparece como primera sentencia del módulo, y que tiene la finalidad de documentar el mismo.

Recordemos que, por convenio, las *docstrings* siempre se delimitan mediante triples comillas ("").

La función `help` muestran la *docstring* del objeto para el que se solicita la ayuda.

Internamente, la *docstring* se almacena en el atributo `__doc__` del módulo.

Ejemplo

```
"""Módulo de ejemplo (ejemplo.py)."""

def saluda(nombre):
    """Devuelve un saludo.

    Args:
        nombre (str): El nombre de la persona a la que saluda.

    Returns:
        str: El saludo.
    """
    return "¡Hola, " + nombre + "!"
```

Existen dos formas distintas de *docstrings*:

- **De una sola línea (one-line):** para casos muy obvios que necesiten poca explicación.
- **De varias líneas (multi-line):** para casos donde se necesita una explicación más detallada.

```
>>> import ejemplo
>>> help(ejemplo)
Help on module ejemplo:

NAME
    ejemplo - Módulo de ejemplo (ejemplo.py).

FUNCTIONS
    saluda(nombre)
        Devuelve un saludo.

        Args:
            nombre (str): El nombre de la persona a la que saluda.

        Returns:
            str: El saludo.

FILE
    /home/ricardo/python/ejemplo.py

>>> help(ejemplo.saluda)
Help on function saluda in module ejemplo:
```

```
saluda(nombre)
    Devuelve un saludo.

Args:
    nombre (str): El nombre de la persona a la que saluda.

Returns:
    str: El saludo.
```

Lo que hace básicamente la función `help(objeto)` es acceder al contenido del atributo `__doc__` del objeto y mostrarlo de forma legible.

Siempre podemos acceder directamente al atributo `__doc__` para recuperar la *docstring* original usando `objeto.__doc__`:

```
>>> import ejemplo
>>> print(ejemplo.__doc__)
Módulo de ejemplo (ejemplo.py).
>>> print(ejemplo.saluda.__doc__)
Devuelve un saludo.

Args:
    nombre (str): El nombre de la persona a la que saluda.

Returns:
    str: El saludo.
```

Esta información también es usada por otras herramientas de documentación externa, como `pydoc`.

3.2. 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 ámbito actual:

```
import math
```

Al importar un módulo de esta forma, se incorpora al espacio de nombres actual (que suele ser el marco del *script* o función que se está ejecutando actualmente) la ligadura entre el nombre del módulo importado y el propio módulo, el cual es **es un objeto de tipo module**.

De esta forma, lo que se importa dentro del espacio de nombres actual es una **referencia** al objeto módulo.

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

Por ejemplo, si tenemos los siguientes *scripts* (o módulos, que es lo mismo en Python) `uno.py` y `dos.py`, y empezamos a ejecutar `uno.py`:

```
1 # uno.py
2
3 x = 7
4
```

```

5 import dos
6
7 a = 4
8 b = 3

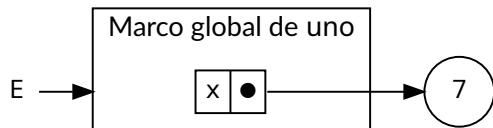
```

```

# dos.py
x = 9
y = 5

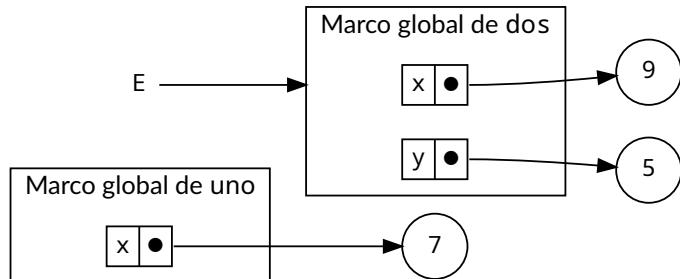
```

Al ejecutar la línea 3 tendremos el siguiente entorno:



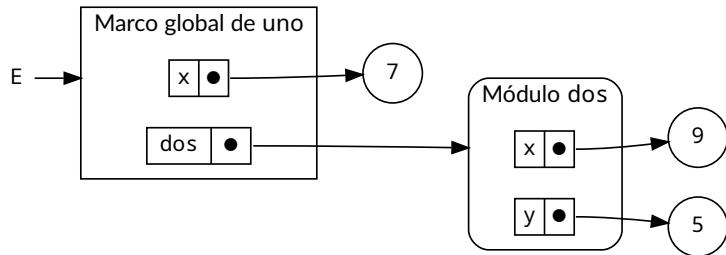
Entorno en la línea 3 del módulo `uno`

Al ejecutar `import dos` en la línea 5 de `uno.py`, pasaremos a ejecutar el script `dos.py` y el marco global del entorno será ahora el marco global de `dos`. Los marcos creados hasta ahora durante la ejecución de `uno` (incluyendo el marco global de `uno`) no estarán en el entorno:

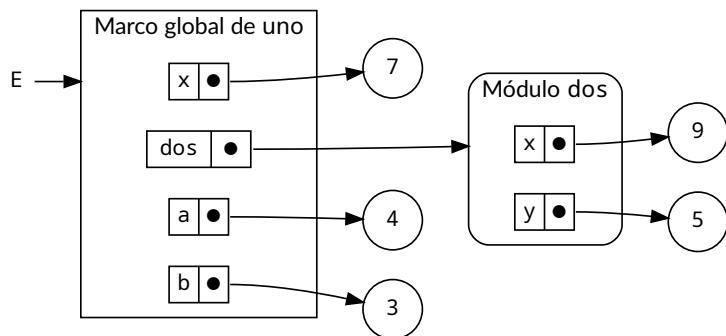


Ejecución de `dos` durante su importación en `uno`

Al finalizar la ejecución del script `dos.py`, el marco global de `dos` se convierte en un objeto de tipo `module` en el montículo, y sus ligaduras se convierten en atributos del objeto. Ese objeto se ligará al identificador `dos` en el marco global de `uno`, que volverá a estar en el entorno y pasará a ser de nuevo el marco global del entorno:

Entorno tras ejecutar `import dos`

Finalmente, tras ejecutar todo el script `uno.py` tendríamos el siguiente entorno:

Entorno justo al final de la ejecución de `uno`

Recordemos que, siempre que tengamos una referencia a un objeto, podemos acceder a los atributos que contiene usando el operador punto (`.`), indicando el objeto y el nombre del atributo al que queramos acceder:

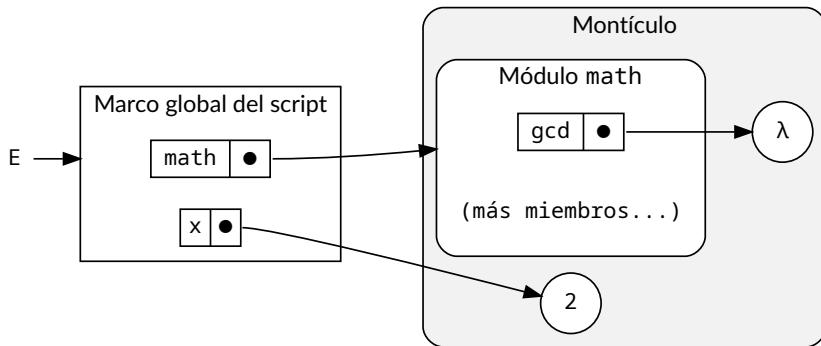
objeto . atributo

Por tanto, para acceder al contenido del módulo importado, indicaremos el nombre de ese módulo seguido de un punto (`.`) y el nombre del contenido al que queramos acceder:

módulo . contenido

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

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



Entorno tras ejecutar las dos sentencias anteriores

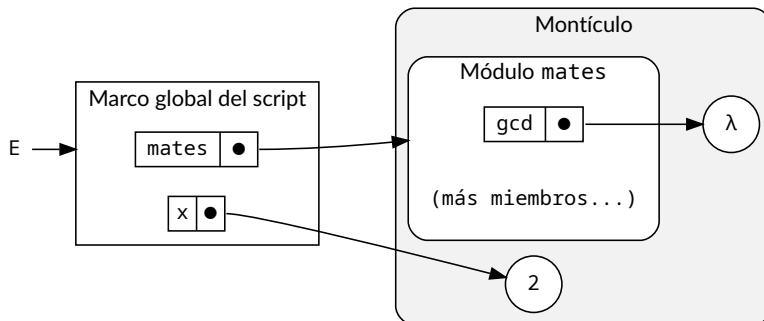
Se puede importar un módulo dándole otro nombre dentro del espacio de nombres actual, usando la sentencia **import ... as**.

Por ejemplo:

```
import math as mates
```

La sentencia anterior importa el módulo **math** dentro del espacio de nombres actual pero con el nombre **mates** en lugar del **math** original. Por tanto, para usar la función **gcd** del ejemplo anterior, haremos:

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



Resultado de ejecutar las dos líneas anteriores

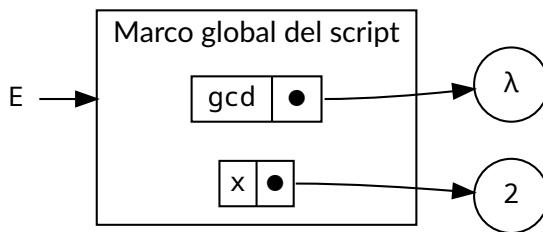
Existe una variante de la sentencia **import** que nos permite importar directamente los miembros de un módulo en lugar del propio módulo. Para ello, se usa la orden **from**.

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, sin necesidad de indicar el nombre del módulo importado:

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



Resultado de ejecutar las dos líneas anteriores

De hecho, ahora el módulo importado no está definido en el módulo importador.

Ese decir: ahora 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 y, por tanto, ese nombre no está definido en el ámbito del módulo importador.

En consecuencia, si hacemos:

```
x = math # error
```

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

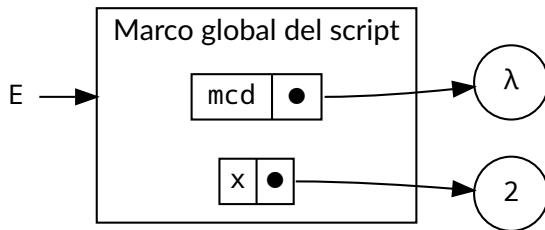
También podemos usar la palabra clave `as` con la orden `from`:

```
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 líneas anteriores

Existe incluso una variante para importar todos los miembros de un módulo:

```
from math import *
```

Con esta sintaxis importaremos todos los miembros del módulo excepto aquellos cuyo nombre comience por un guion bajo (`_`).

Los miembros con nombres que comienzan por `_` son considerados **privadas** o internos al módulo, lo que significa que no están concebidos para ser usadas por los usuarios del módulo y que, por tanto, no forman parte de su **interfaz** (no deben usarse fuera del módulo).

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 «machaque» sin control definiciones ya existentes.

Para saber qué miembros (públicos o privados) contiene un módulo, se puede usar la función `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']
```

La función `dir` puede usarse con cualquier objeto al que podamos acceder a través de una referencia, aunque también podemos llamarla sin argumentos, y en ese caso mostrará los miembros del módulo actual:

```
>>> import math
>>> dir()
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__', 'math']
```

Todos los objetos tienen un atributo llamado `__dict__` que contiene un diccionario que almacena los atributos del objeto.

Los módulos son objetos y, por tanto, también tienen el atributo `__dict__`, que en este caso representa, a todos los efectos, al **espacio de nombres** del módulo:

```
>>> import math
>>> math.__dict__
{'__name__': 'math', '__doc__': "This module provides access to the
mathematical functions undefined by the C standard.", '__package__': '',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__':
ModuleSpec(name='math', loader=<class '_frozen_importlib.BuiltinImporter'>,
origin='built-in'), 'acos': <built-in function acos>, 'acosh': <built-in
function acosh>, 'asin': <built-in function asin>, 'asinh': <built-in
function asinh>, 'atan': <built-in function atan>, 'atan2': <built-in
function atan2>, 'atanh': <built-in function atanh>, 'cbrt': <built-in
function cbrt>, 'ceil': <built-in function ceil>, 'copysign': <built-in
function copysign>, 'cos': <built-in function cos>, 'cosh': <built-in
function cosh>, 'degrees': <built-in function degrees>, 'dist': <built-in
function dist>, 'erf': <built-in function erf>, 'erfc': <built-in function
erfc>, 'exp': <built-in function exp>, ...}
```

`dir(math)` equivale a `sorted(math.__dict__.keys())`.

También disponemos de la función `globals`, la cual devuelve un diccionario que contiene las definiciones globales del ámbito actual:

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': '_pyrepl',
'__loader__': None, '__spec__': None, '__annotations__': {}, '__builtins__':
<module 'builtins' (built-in)>, '__file__':
'/usr/lib/python3.13/_pyrepl/__main__.py', '__cached__':
'/usr/lib/python3.13/_pyrepl/_pycache_/_main_.cpython-313.pyc', 'math':
<module 'math' (built-in)>}
```

`dir()` equivale a `sorted(globals().keys())`.

El ámbito del *script* donde se almacena un módulo constituye el ámbito global de ese módulo y, al importar un módulo, su marco global debe permanecer en la memoria para que los miembros del módulo puedan acceder a otros elementos del mismo.

En concreto, cuando el módulo se importa usando `import <módulo>`:

1. Se mete su marco en la pila.
2. Se ejecutan sus sentencias, y sus definiciones crean ligaduras y variables que se almacenan en ese marco.
3. Cuando se acaba, se saca el módulo de la pila y se convierte en un objeto en el montículo que representa al módulo.

El módulo representa el marco global para todas las funciones definidas dentro de ese módulo.

Esto es debido a que las funciones definidas dentro del módulo recuerdan las ligaduras globales del módulo en el que se han definido por medio del atributo `__globals__` de la función.

Para ello, lo que hace el intérprete básicamente es lo siguiente:

$$\langle \text{func} \rangle.\texttt{__globals__} = \langle \text{módulo} \rangle.\texttt{__dict__}$$

para todas las funciones `\langle func \rangle` definidas en el módulo `\langle módulo \rangle`.

El marco del módulo importador NO está dentro del entorno de esas funciones.

Por ejemplo:

```
# uno.py:  
x = 1  
def f():  
    return x
```

```
# dos.py:  
import uno  
print(uno.f()) # imprime 1
```

Al hacer **import uno**:

1. Se crea un marco para **uno** en la pila.
2. Se ejecutan sus sentencias, que almacenan las ligaduras y variables de **x** y **f** en ese marco.
3. Se hace:

```
f.__globals__ = uno.__dict__
```

4. Cuando se termina de ejecutar el módulo, se saca su marco de la pila y se almacena como un objeto en el montículo.
5. Se crea en el marco de **dos** una ligadura que hace referencia a ese objeto.

En la llamada **uno.f()**, el entorno de **f** está formado por su propio marco y por el marco global de **uno**, por lo que **f** tiene toda la información que necesita para funcionar.

El entorno de **f** NO contiene el marco global de **dos**.

Por ejemplo:

```
# uno.py:  
x = 1  
def f():  
    return x
```

```
# dos.py:  
import uno  
  
x = 35      # crea una nueva x en dos  
print(uno.f()) # sigue imprimiendo 1
```

Aquí, la llamada **uno.f()** sigue devolviendo **1** y no **35**, ya que **f** sigue recordando que **x** vale **1**.

Recordemos que el entorno de `f` está formado por su marco y por el marco global de `uno` (el cual recuerda a través de su atributo `__globals__`).

Y en este caso:

```
# uno.py:  
  
def f():  
    return x
```

```
# dos.py:  
  
import uno  
  
x = 35          # crea una nueva x en uno  
  
print(uno.f()) # da error
```

Aquí, la llamada `uno.f()` da un error porque el entorno de `f` no contiene ninguna `x`, ya que ese entorno está formado por su marco y por el marco global de `uno` (NO contiene al marco global de `dos`).

Si usamos `from ... import ...`:

```
# uno.py:  
  
x = 1  
  
def f():  
    return x
```

```
# dos.py:  
  
from uno import f  
  
print(f()) # imprime 1
```

Al hacer `from uno import f`, estamos importando sólo la función `f`, no el módulo `uno` como tal, pero al hacerlo ya estaríamos perdiendo el marco global del módulo `uno`, que contiene información que necesita la función `f` (en este caso, la variable `x`).

Para que la llamada `f()` funcione correctamente en `dos`, el intérprete debe conservar el marco del módulo `uno` y enlazar su `__dict__` con `f` mediante el atributo `__globals__` de `f`.

En detalle, cuando el intérprete ejecuta `from uno import f`:

1. Crea el marco de `uno` en la pila y ejecuta sus sentencias.
2. En el marco de `uno` se guardan las ligaduras y variables de `x` y `f`.
3. Se hace:

```
f.__globals__ = uno.__dict__
```

4. Al terminar de ejecutarse el módulo `uno`, se saca su marco de la pila pero no se elimina, sino que se almacena en el montículo.

5. En el marco de `dos`, se crea la ligadura entre `f` y la función.

Si ahora hacemos `x = 99` en el módulo `dos`:

```
# uno.py:  
  
x = 1      # variable global de uno  
  
def f():  
    return x
```

```
# dos.py:  
  
from uno import f  
  
x = 99      # crea una nueva x en dos  
  
print(f())  # sigue imprimiendo 1
```

El resultado sigue siendo el mismo que antes, porque:

- La asignación `x = 99` que se ejecuta dentro del módulo `dos` crea una nueva variable `x` en el marco de `dos`, y no modifica la `x` de `uno`.
- La `f` importada en `dos` recuerda sus variables globales a través de su atributo `__globals__`, por lo que recuerda que `x` valía `1` cuando se definió al ejecutarse la sentencia `from uno import f`.
- Cuando se ejecuta `f`, su entorno está formado por su marco y el marco global de `uno`, en ese orden.
- Por tanto, `f` accede a la `x` de `uno`, no la de `dos`.

Es importante también tener en cuenta que el entorno de `f` NO incluye el marco global de `dos`.

Por tanto, lo siguiente daría error:

```
# uno.py:  
  
def f():  
    return x      # Una variable que no está en uno
```

```
# dos.py:  
  
from uno import f  
  
x = 99      # crea una nueva x en dos  
  
print(f())  # da error porque la x de dos no está en el entorno de f
```

3.3. Módulos como scripts

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

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

se ejecutará el módulo como un *script*, 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 indica que ese *script* se considera el **módulo principal** (en inglés, *main module*) del programa.

Por eso, si se añade este código al final del módulo:

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

esas sentencias se ejecutarán únicamente cuando el módulo se ejecute como el módulo principal del programa.

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 puede importar dentro de otro.

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

```
$ python fact.py 4
24
```

Y si importamos el módulo dentro de otro, la condición del último **if** no se cumplirá, por lo que su único efecto será el de incorporar la definición de la función `fac` dentro del módulo importador.

3.4. La librería estándar

La **librería estándar** (o **biblioteca estándar**) de Python contiene módulos predefinidos que proporcionan:

- Acceso a funcionalidades del sistema, como operaciones de E/S sobre archivos.
- Soluciones estandarizadas a muchos de los problemas que los programadores pueden encontrarse en su día a día.

Algunos módulos están diseñados explícitamente para promover y mejorar la portabilidad de los programas Python abstrayendo los aspectos específicos de cada plataforma a través de un API independiente de la plataforma.

La documentación contiene la información más completa sobre el contenido de la librería estándar, a la que podemos acceder a través de la siguiente dirección:

<https://docs.python.org/3/library/index.html>

En esa dirección, además, se incluye información sobre:

- Funciones, constantes, excepciones y tipos predefinidos, que también forman parte de la librería estándar.
- Componentes opcionales que habitualmente podemos encontrar en cualquier instalación de Python.

3.5. Paquetes

Los **paquetes** son una forma de estructurar el espacio de nombres de módulos de Python usando *nombres de módulo con puntos*.

Por ejemplo, el nombre del módulo `A.B` representa el submódulo `B` en un paquete llamado `A`.

Así como el uso de módulos evita el *name clash* entre los miembros de diferentes módulos escritos por diferentes creadores, los nombres de módulos con puntos evita el *name clash* entre los nombres de los propios módulos.

Es decir, nos permite tener varios módulos con el mismo nombre, siempre que estén en paquetes distintos.

Esto evita que los creadores de módulos se tengan que preocupar por los nombres de los módulos creados por otros creadores.

En Python, un **paquete** es una unidad de organización del código que agrupa módulos relacionados bajo un mismo espacio de nombres.

Dicho de forma resumida, un paquete es un directorio que contiene módulos Python y que puede importarse.

Más concretamente, un paquete es un directorio que Python reconoce como importable y que:

- Contiene uno o más módulos (archivos `.py`).
- Puede contener *subpaquetes*.
- Define un espacio de nombres jerárquico.

- Puede contener un archivo de inicialización y control del paquete llamado `__init__.py`.

Ideas clave:

Concepto	Qué es
Módulo	Un archivo <code>.py</code>
Paquete	Un directorio de módulos
Subpaquete	Un paquete dentro de otro

Ejemplo conceptual:

```
xml           # paquete
xml.etree    # subpaquete
xml.etree.ElementTree # módulo
```

Por ejemplo, supongamos la siguiente estructura de archivos y directorios:

```
sonido/
    __init__.py          <-- Paquete de nivel superior
    utilidades.py        <-- Inicializa el paquete sonido
    formatos/
        __init__.py      <-- Módulo utilidades del paquete sonido
        wavread.py        <-- Subpaquete de conversión entre formatos
        wavwrite.py       <-- Inicializa el subpaquete formatos
        aiffread.py       <-- Módulo wavread del subpaquete formatos
        aifffwrite.py     <-- Módulo wavwrite del subpaquete formatos
        auread.py         <-- Módulo aiffread del subpaquete formatos
        auwrite.py        <-- Módulo aifffwrite del subpaquete formatos
        ...
    efectos/
        __init__.py      <-- Módulo auread del subpaquete formatos
        echo.py           <-- Subpaquete de efectos de sonido
        surround.py      <-- Inicializa el subpaquete efectos
        reverse.py        <-- Módulo surround del subpaquete efectos
        ...
    filtros/
        __init__.py      <-- Módulo reverse del subpaquete efectos
        equalizer.py     <-- Subpaquete de filtros
        vocoder.py       <-- Inicializa el subpaquete filtros
        karaoke.py        <-- Módulo equalizer del subpaquete filtros
        ...
    ...
```

Los usuarios del paquete pueden importar módulos individuales del mismo.

Por ejemplo, la siguiente sentencia:

```
import sonido.efectos.echo
```

importa el módulo `sonido.efectos.echo`, es decir, el módulo `echo` del subpaquete `efectos` del paquete `sonido`.

Para usarlo, debe hacerse referencia al mismo con el nombre completo (también llamado *nombre totalmente cualificado*):

```
sonido.efectos.echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra alternativa para importar el módulo es:

```
from sonido.efectos import echo
```

Esto también carga el módulo `echo` y lo deja disponible sin su prefijo de paquete, por lo que puede usarse así:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra variante más es importar directamente el miembro deseado (función, variable, etc.) del módulo:

```
from sonido.efectos.echo import echofilter
```

De nuevo, esto carga el módulo `echo`, pero deja directamente disponible la función `echofilter`, por lo que se puede usar sin necesidad de poner el nombre del módulo como prefijo:

```
echofilter(input, output, delay=0.7, atten=4)
```

Nótese que al usar:

```
from paquete import elemento
```

el elemento puede ser tanto un módulo (o subpaquete) del paquete, como algún otro nombre definido en el paquete, como por ejemplo una función, una variable o una clase.

La sentencia `import` primero verifica si el elemento está definido en el paquete; si no, asume que es un módulo y trata de cargarlo. Si no lo puede encontrar, se genera una excepción `ImportError`.

Por otro lado, cuando se usa la sintaxis:

```
import elemento.subelemento.subsubelemento
```

cada elemento excepto el último debe ser un paquete; ese último elemento puede ser un módulo o un paquete pero no puede ser una función, variable o clase definida en el elemento previo.

El archivo `__init__.py` sirve para inicializar un paquete y controlar su comportamiento cuando se importa.

Tradicionalmente, un directorio sólo era considerado un paquete si contenía un archivo llamado `__init__.py`.

Hoy en día ya no es necesario, pero aún sigue siendo válido, habitual y recomendado.

Además, el uso de `__init__.py` permite:

- Ejecutar código al importar el paquete.
- Controlar qué se expone al usar `*` en una importación, como en una sentencia `import` así:

```
from paquete import *
```

Cuando se importa un paquete en sí:

```
import sonido
```

Python hace lo siguiente, en este orden:

1. Localiza el paquete `sonido`.
2. Ejecuta el contenido del archivo `sonido/__init__.py`.
3. Crea el objeto llamado `sonido`, que representa al paquete.

Es decir:

- `__init__.py` es código Python normal.
- Importar un paquete supone, en esencia, importar el `__init__.py` del paquete como si fuera un módulo.
- De hecho, el objeto que representa al paquete es de tipo *módulo*.

El archivo `__init__.py` permite decidir qué «sale hacia fuera» cuando se importa el paquete en sí.

Por ejemplo, supongamos que el archivo `sonido/__init__.py` contiene la siguiente línea:

```
from sonido.utilidades import suma, resta
```

Ahora, se puede hacer directamente:

```
from sonido import suma
```

sin necesidad de dar dos pasos:

```
from sonido.utilidades import suma
```

Cuando se intenta importar todos los módulos de un paquete:

```
from sonido.efectos import *
```

lo que ocurre depende de lo que haya en el `__init__.py` del paquete.

Los módulos que se importen serán los que el creador del paquete haya establecido previamente.

Para ello, la sentencia `import` sigue el siguiente convenio: si el `__init__.py` del paquete contiene una lista llamada `__all__`, ésta se tomará como la lista de nombres de módulos que se importarán al ejecutar la sentencia `from <paquete> import *`.

El creador del paquete es responsable de mantener actualizada esta lista cada vez que se publique una nueva versión del paquete.

Por ejemplo, si el archivo `sonido/efectos/__init__.py` contiene el siguiente código:

```
__all__ = ["echo", "surround", "reverse"]
```

entonces la siguiente sentencia importaría esos tres módulos del paquete `sonido.efectos`:

```
from sonido.efectos import *
```

Hay que tener en cuenta que los módulos pueden quedar sombreados por nombres definidos localmente.

Por ejemplo, si se añade una función llamada `reverse` al archivo `sonido/efectos/__init__.py`:

```
__all__ = [
    "echo",      # se refiere al archivo 'echo.py'
    "surround",  # se refiere al archivo 'surround.py'
    "reverse",   # ahora se refiere a la función 'reverse' (!)
]

def reverse(msg: str):  # hace sombra al módulo 'reverse.py'
    return msg[::-1]    # si se hace 'from sonido.efectos import *'
```

entonces la orden

```
from sonido.efectos import *
```

solo importaría los dos módulos `echo` y `surround`, pero no el módulo `reverse`, ya que este último queda sombreado por la función `reverse` definida localmente.

Si no se define `__all__`, la sentencia:

```
from sonido.efectos import *
```

no importa todos los módulos del paquete `sonido.efectos` al espacio de nombres actual.

Lo que hace es:

1. Asegurarse que se haya importado el paquete `sonido.efectos`, posiblemente ejecutando algún código de inicialización que haya en `__init__.py`.
2. Importa los nombres que estén definidos en el paquete.

Esto incluye:

- Cualquier nombre definido (y módulos explícitamente cargados) por `__init__.py`.
- Cualquier módulo del paquete que pudiera haber sido explícitamente cargado por sentencias `import` anteriores.

En el siguiente ejemplo:

```
import sonido.efectos.echo
import sonido.efectos.surround
from sonido.efectos import *
```

los módulos `echo` y `surround` se importan en el espacio de nombre actual porque están definidos en el paquete `sonido.efectos` cuando se ejecuta la sentencia `from ... import`.

Esto también funciona cuando se define `__all__`.

A pesar de que ciertos módulos están diseñados para exportar sólo nombres que siguen ciertos patrones cuando se usa `from <paquete> import *`, se considera una mala práctica en código de producción.

No hay nada malo en usar `from <paquete> import <submodulo>`. De hecho, esta es la notación recomendada a menos que el módulo que queremos importar necesite usar módulos con el mismo nombre desde un paquete diferente.

Cuando se estructuran paquetes en subpaquetes (como en el ejemplo del paquete `sonido`) los módulos pueden hacer **importaciones absolutas** para referirse a módulos de paquetes «hermanos».

Las importaciones absolutas consisten en usar el nombre completo del módulo, incluyendo todo el camino desde el paquete principal hasta el módulo a importar.

Por ejemplo, si el módulo `sonido.filtros.vocoder` necesita usar el módulo `echo` del paquete `sonido.efectos`, puede hacer:

```
from sonido.efectos import echo
```

También se pueden hacer **importaciones relativas** usando sólo la forma de importación `from <módulo> import <nOMBRE>`.

Estas importaciones usan:

- Un punto (`.`) para indicar el paquete actual.
- Dos puntos (`..`) para indicar el paquete «padre».
- Tres puntos (`....`) para indicar el paquete «abuelo».

y así sucesivamente, en la cadena de paquetes involucrados en la importación relativa.

Por ejemplo, desde el módulo `surround` se puede hacer:

```
from . import echo          # importa módulo echo del paquete actual (surround)
from .. import formatos      # importa módulo formatos del paquete padre (sonido)
from ...filtros import equalizer  # importa módulo equalizer del paquete filtros
```

Hay que tener en cuenta que las importaciones relativas se basan en el nombre del paquete del módulo actual. Como el módulo principal de un programa Python no pertenece a ningún paquete, debe usar siempre importaciones absolutas.

4. Criterios de descomposición modular

4.1. Introducción

No existe una única forma de descomponer un programa en módulos (entendiendo 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?

4.2. Tamaño y número

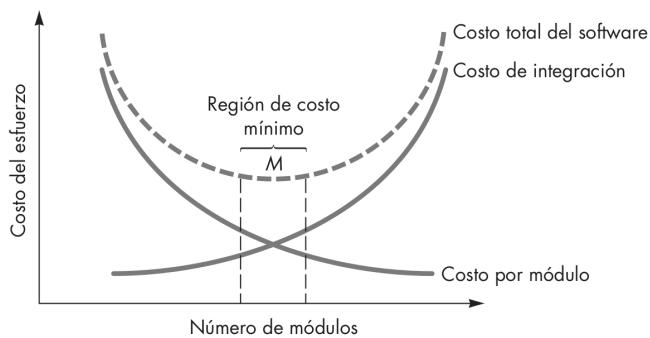
Se supone que, si seguimos al pie de la letra la estrategia de diseño basada en la división de problemas, sería posible concluir que si el programa se dividiera indefinidamente, cada vez se necesitaría menos esfuerzo hasta llegar a cero.

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.
- Por otra parte, 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**: abstracción, ocultación de información, independencia funcional, reusabilidad y principios SOLID.

4.3. 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.

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

Es decir, se puede estudiar y usar el subsistema sabiendo *qué* hace sin tener que saber *cómo* lo hace.

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 (el que está justo debajo) 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** a partir de 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 (las denominadas **abstracciones modulares**) son más fáciles de usar, diseñar y mantener.

4.4. 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.
- 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 decisiones de diseño al exterior.

- Por tanto, B debe **ocultar** al exterior sus decisiones de **implementación** y exponer sólo lo necesario para que otros lo puedan utilizar por medio de una interfaz estable.

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

Es decir: cada módulo debe ser una cápsula recelosa de su privacidad que tiene «aversión» por exponer sus decisiones internas a los demás.

Antes de Parnas, los programas se dividían en módulos pensando en:

- qué funciones hacían, o
- cómo fluían los datos.

Él propuso otro criterio mucho más potente:

Dividir el programa según lo que pueda cambiar, no según lo que hace.

O, dicho de otra forma:

No diseñes módulos por lo que hacen, sino por lo que quieras proteger del cambio.

Según este principio, un módulo es una **decisión de diseño aislada**.

«Ocultar información» no significa esconder datos por seguridad.

Significa:

- Ocultar decisiones internas que puedan cambiar en el futuro, como:
 - Estructuras de datos.
 - Algoritmos.
 - Formatos de almacenamiento.
 - Detalles técnicos.

- Y exponer sólo:
 - Operaciones públicas claras.

El principio de ocultación de información tiene las siguientes **consecuencias prácticas**:

- Reduce dependencias entre módulos.
- Facilita el mantenimiento del programa.
- Evita que los cambios y errores en un módulo se propaguen al resto del programa.
- Permite la evolución del sistema.

La **abstracción**, la **encapsulación** y la **ocultación de información** se complementan.

Es importante no confundir *encapsulación* con *ocultación de información*:

- La **ocultación de información** es un principio de diseño, es un criterio que guía la descomposición del programa en módulos.
- La **encapsulación** es un mecanismo de agrupamiento y protección que crea cápsulas que aíslan elementos del exterior.

En ese sentido:

- La ocultación de información es el proceso mental.
- La encapsulación es la herramienta para implementar la ocultación.

La **abstracción** y la **ocultación de información** sirven de guía para:

- Descomponer un programa en módulos.
- Determinar qué elementos van en cada módulo.
- Decidir qué elementos exporta y qué elementos oculta cada módulo.

La **abstracción** puede usarse como una **técnica de diseño** que **nos da un método para descomponer el programa en módulos**.

La idea clave no es esconder cambios, sino **simplificar la complejidad**.

Se basa en preguntarse:

- Qué conceptos principales forman el sistema.
- Cómo se ven esos conceptos desde un punto de vista abstracto, es decir, desde el punto de vista del *qué* hacen y no del *cómo*.
- De qué manera se pueden descomponer esos conceptos por niveles de abstracción para ir bajando el nivel de detalle.
- Qué es esencial y qué puede ignorarse en un determinado nivel.

Por tanto, nos ayuda a decidir:

- Qué módulos debe tener el programa.
- Qué información debe exponer y qué debe ocultar cada módulo, porque afirma que:

- La *interfaz* debe estar formada, precisamente, por los elementos que responden a «qué» hace el módulo.
- La *implementación* debe estar formada por los que responden a «cómo» lo hace.

La **ocultación de información** es un **principio de diseño** que, al igual que la abstracción, nos proporciona una guía para la descomposición de un programa en módulos.

La idea clave es **proteger contra el cambio**.

Se basa en preguntarse:

- Qué decisiones de diseño pueden cambiar en el futuro.
- Cómo afectarían esos cambios a los elementos de que forman el programa.
- Qué partes del programa necesitan realmente conocer esas decisiones de diseño.

Por tanto, nos ayuda a decidir:

- Qué módulos debe tener un programa:

Aquí, los módulos que salgan serán aquellos que se obtengan al preguntarnos qué decisiones de diseño pueden cambiar.

- Qué detalles debe ocultar un módulo y qué debe exponer a los demás, es decir, qué detalles internos deben conocerse de un módulo para poder usarlo:

Aquí, las decisiones de diseño deberán ir ocultas y hay que crear interfaces limpias que no expongan ninguna decisión de diseño.

- En qué módulo va cada elemento del programa, es decir, qué elementos deben formar parte de un módulo y cuáles deben formar parte de otros módulos:

Aquí, los elementos que necesiten conocer las mismas decisiones de diseño irán en el mismo módulo.

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, menos expuesto estará a verse afectado por posibles cambios futuros en el funcionamiento interno del mismo.

Al **creador** de un módulo...

- ... le interesa la **abstracción** y la **ocultación de información** porque le ayudan a determinar qué módulos deben formar parte del programa, qué elementos deben formar parte de su módulo y qué elementos deben exportarse u ocultarse al exterior.

Además, también le interesa la **ocultación de información** porque le permite tomar decisiones de diseño *ahora* sobre el funcionamiento interno de su módulo y cambiar de opinión sobre las mismas *más adelante* sin afectar a los usuarios de su módulo.

La clave está en combinar adecuadamente las dos técnicas.

Para ello, se suele seguir una regla de tres pasos que pueden aplicarse recursivamente en caso necesario:

1. Se deducen las abstracciones más generales a partir del modelo del dominio.
2. Se preven las decisiones de diseño que habrá que tomar para implementar dichas abstracciones y en qué medida pueden cambiar.
3. Se rompen esas abstracciones en módulos, según los posibles cambios en las decisiones de diseño.

Así se empieza por abstracciones, pero luego se modulariza por decisiones que pueden cambiar.

Modularizar por abstracciones tiene la ventaja de que produce un código más comprensible, pero suele generar módulos que cambian mucho.

En cambio, modularizar por decisiones cambiantes lleva a un mantenimiento más fácil aunque resulta menos intuitivo al principio.

4.5. Independencia funcional

La **independencia funcional** es otro criterio a seguir para obtener una modularidad efectiva.

Se obtiene una modularidad efectiva cuando los módulos son **funcionalmente independientes**.

Esto se logra desarrollando módulos de manera que:

- Cada módulo resuelva **una funcionalidad específica**, es decir, que se dedique a una cosa cosa y que tenga una finalidad concreta.
- Tenga una **interfaz sencilla** cuando se vea desde otras partes del programa (idealmente, mediante paso de parámetros) y que esté **destinada únicamente a cumplir con esa funcionalidad**.

Al limitar su objetivo a una sola tarea o finalidad, el módulo necesita menos ayuda de otros módulos.

Por ello, el módulo debe ser tan **independiente** como sea posible del resto de los módulos del programa.

Es decir: debe depender lo menos posible de lo que hagan otros módulos, y también debe depender 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», tener «aversión» a relacionarse con otros módulos y **depender lo menos posible de 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**.

4.5.1. Cohesión

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

Cuanto más cohesivo sean nuestros módulos, 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 pertenecen 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*) al resto.

En pocas palabras, un módulo cohesivo debe tener un único objetivo, y todos los elementos que lo componen deben contribuir 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 pertenecen 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»).

4.5.2. 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.
- La cantidad y complejidad de 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 todos los módulos de los que depende éste, que además suelen ser muchos.

Los módulos se puede reutilizar o probar por separado ya que todos los módulos que dependen entre sí forman un *pack* y sólo funcionan cuando van todos juntos.

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 acoplamiento tenemos. Lo importante es intentar conseguir un acoplamiento bajo y reconocer cuándo hay mucho acoplamiento 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 *comutador 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.

4.6. Reusabilidad

La **reusabilidad** es un factor de calidad del software que se puede aplicar también 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 reusables (que en este contexto también se denominan **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 que consiste principalmente en ir combinando módulos 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.

4.6.1. Diseño ascendente

Cuando hablamos de la programación procedural, dijimos que el diseño descendente tiene ventajas pero también inconvenientes, entre los cuales teníamos:

- Hasta que no se ha llegado al nivel más bajo, no se tiene algo que pueda probarse, ensamblarse y ejecutarse.
- Por tanto, es un enfoque más difícil de aplicar si no se entienden bien los detalles prácticos ni se conocen perfectamente desde el principio los requisitos del programa que hay que constituir.
- Puede ignorar la reutilización de componentes ya existentes.
- Resulta fácil obtener subprogramas que sólo sean útiles para el programa actual y no se puedan generalizar para su reutilización en otras situaciones.

Por tanto, con el diseño descendente a veces cuesta conseguir la programación de subprogramas o módulos reutilizables.

Frente al diseño descendente, tenemos también el llamado **diseño ascendente** o **bottom-up**, que es una técnica que se puede aplicar tanto en programación procedural (con subprogramas) como en programación modular (con módulos).

Por eso, a partir de ahora hablaremos de *componentes* para referirnos a módulos o subprogramas.

Como su nombre indica, consiste en ir de abajo arriba, construyendo componentes básicos que se van combinando y ensamblando poco a poco hasta obtener el programa completo.

Además, en todo momento se busca que esos componentes sean reutilizables, es decir, aprovechables en otros programas.

Cómo se aplica el diseño ascendente:

1. Se diseñan ladrillos básicos de construcción en forma de pequeños componentes independientes y genéricos.
2. Se combinan esos componentes para crear otros cada vez más complejos, integrándolos unos con otros.
En la práctica, consiste en hacer que el componente que estamos programando en un momento dado, utilice a otros que ya hemos programado (nosotros u otros programadores).
3. Finalmente, se integran todos los componentes para formar el sistema completo.

El diseño ascendente tiene **ventajas**:

- Favorece la reutilización de código ya existente, en forma de módulos o librerías de subprogramas.

- Permite programar y verificar partes funcionales del programa *antes de tener el programa completo*, ya que se pueden ir ensamblando esas partes unas con otras a medida que se van programando y sin tener que esperar a tenerlas todas.

Esas partes se hacen cada vez más complejas conforme se van ensamblando entre sí los distintos componentes que las forman y, con ello, se va construyendo el programa *de abajo arriba*.

- El diseño global del programa puede cambiar a medida que se van conectando los componentes unos con otros, por lo que el diseño ascendente es más flexible y es menos sensible a cambios en la arquitectura del programa.

Pero también tiene **inconvenientes**:

- Puede ser difícil garantizar que al final todos los componentes encajarán juntos perfectamente en un sistema coherente.
- Se necesita experiencia para construir componentes lo suficientemente genéricos como para resultar reutilizables.

La opción que generalmente resulta más conveniente es la de combinar ambos enfoques en una única estrategia.

4.6.2. Estrategia sándwich

En la práctica, el diseño descendente y el ascendente se combinan en un único enfoque llamado habitualmente **estrategia sándwich**.

Este enfoque consiste en lo siguiente:

1. **Diseño top-down:** Definir la arquitectura general y los grandes módulos según los requisitos del sistema, empezando por una visión a vista de pájaro de la solución que se debe construir.
2. **Diseño bottom-up:** Diseñar y programar componentes básicos (módulos y procedimientos) que se sabe que se necesitarán (por experiencia previa o por requisitos funcionales claros) y que se van a tener que combinar entre sí.
3. **Integración en el medio (sándwich):** Los componentes de más alto nivel (*top-down*) se implementan usando los de más bajo nivel (*bottom-up*), asegurando coherencia entre la visión global que ofrece el *top-down* y los aspectos prácticos que ofrece el *bottom-up*.

En resumen, se diseña la arquitectura desde arriba y se construyen componentes reutilizables desde abajo, encontrándose ambos en el desarrollo de los componentes intermedios.

En la práctica, podemos decir que:

1. El enfoque *top-down* se puede usar para identificar qué módulos principales van a formar parte del programa.
2. También se puede usar el enfoque *top-down* para programar los subprogramas y los detalles algorítmicos que formen cada módulo, aplicando el refinamiento sucesivo pero sin bajar a un nivel de detalle muy alto (o sea, dejándolos a medio terminar).
3. Se puede aplicar el enfoque *bottom-up* para construir componentes que se puedan reutilizar en varias partes del programa y que no requieran de otras partes, de forma que se puedan escribir y probar ya, sin esperar a tener más partes funcionando.

4. Estos componentes se usan para terminar de programar los algoritmos que se concretaron a medias en el paso 2 anterior, de forma que los del paso 2 llaman o usan a los del paso 3.

La estrategia sándwich tiene las siguientes **ventajas**:

- Combina lo mejor de ambos enfoques.
- Garantiza visión global y cohesión (*top-down*).
- Permite reutilización y validación temprana de componentes (*bottom-up*).
- Facilita la integración progresiva del sistema.

4.7. Principios SOLID

Los **principios SOLID** son un conjunto de cinco principios de diseño modular cuyo objetivo es hacer el software más mantenible, extensible, comprensible y robusto.

Inicialmente fueron propuestos para el paradigma de la *Programación orientada a objetos*, pero pueden aplicarse a cualquier sistema modular.

SOLID es un acrónimo que fue popularizado por Robert C. Martin («Uncle Bob»), y proviene del nombre de los cinco principios en inglés:

- Single Responsibility Principle (SRP).
- Open/Closed Principle (OCP).
- Liskov Substitution Principle (LSP).
- Interface Segregation Principle (ISP).
- Dependency Inversion Principle (DIP).

Los cinco principios son:

- **Principio de Responsabilidad Única (SRP)**: los módulos deben tener un único propósito y una única responsabilidad clara.
- **Principio Abierto/Cerrado (OCP)**: los módulos deben estar abiertos para su extensión pero cerrados para su modificación.
- **Principio de Sustitución de Liskov (LSP)**: los módulos deben poderse intercambiar unos por otros siempre que respeten la misma especificación.
- **Principio de Segregación de la Interfaz (ISP)**: los módulos sólo deben exponer la mínima funcionalidad que sea relevante para sus usuarios, y los usuarios no deben depender de interfaces que no usan (es decir: la interfaz debe ser mínima).
- **Principio de Inversión de Dependencias (DIP)**: los módulos deben depender de abstracciones y no de concreciones.

Bibliografía

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

Python Software Foundation. n.d. *Sitio Web de Documentación de Python*. <https://docs.python.org/3>.