Tabla de contenidos

1. ¿Qué es modularización?
2. Importación en Python
3. Sentencias de importación
4. ¿Y por qué usar módulos?
5. Abstracción de componentes

¿Qué es modularización?

Programación modular es una técnica de diseño de software que se basa en separar el código de un programa en componentes lógicos llamados **módulos**, cada uno con funcionalidades independientes. En términos muy simples, es separar de forma ordenada el código de un programa en múltiples archivos.

Importación en Python

Python está orientado a generar código modular. Cada archivo con extensión `.py` que tiene sentencias y definiciones es un módulo en Python. Con esto, se permite importar definiciones de un módulo en otro mediante la palabra reservada `import`. Esto permite importar variables, funciones, clases y cualquier otro tipo de definición creada.

Por ejemplo, junto a estos archivos podrás encontrar un módulo llamado `ejemplo.py`, que tiene la definición de una variable, una función y una clase. Revisa ese archivo y luego continúa leyendo. Mediante `import` podemos ingresar todas esas definiciones y utilizarlas en este documento:

```
import ejemplo
```

```
ejemplo.mi_variable
```

```
10
```

```
ejemplo.saludar()
```

```
¡Hola!
```

```
instancia = ejemplo.MiClase("42")
instancia.argumento
```

```
'42'
```

¿Cómo se ejecuta la importación?

Al ejecutar un archivo de extensión `.py` las líneas se ejecutan en orden, una línea a la vez. Cuando se llega a una sentencia `import`, Python **ejecuta** el archivo referenciado completo, creando variables y creando definiciones; luego vuelve al archivo original a continuar con el resto de las sentencias. En este último, se puede utilizar el código ejecutado en el módulo importado, pues ya fue ejecutado.

Esto se cumple para importaciones en cadena. Digamos que un archivo `a.py` importa a `b.py`, el cual a su vez importa a `c.py`. Si ejecutamos `a.py`:

1. Al llegar a la sentencia `import b` en `a.py`, se comienza ejecutando `b.py`
2. Al llegar a la sentencia `import c` en `b.py`, se comienza ejecutando `c.py`

3. Si hay más importaciones en `c.py`, se ejecuta sucesivamente de la misma forma esos módulos. Luego se ejecuta el resto de `c.py`
4. Se ejecuta el resto de `b.py`
5. Se ejecuta el resto de `a.py`

Python lleva registro de los módulos cuya importación ya se inició, y no los vuelve a ejecutar. Gracias a esto, un ciclo de importaciones no produce un ciclo infinito: si en el ejemplo anterior `c.py` tuviera una sentencia `import b`, Python no ejecutaría `b.py` de nuevo, sino que entregaría el módulo `b` tal como está en ese momento (parcialmente ejecutado). Ojo: si `c.py` intentara usar en ese instante definiciones de `b.py` que aún no alcanzan a ejecutarse, se produciría un error. Por esto, en general conviene evitar los ciclos de importación al diseñar un programa.

Sentencias de importación

Hasta el momento vimos una sola forma de importar un módulo, pero hay varias formas de hacerlo.

Importación completa

Es la mostrada anteriormente y es de la forma:

```
import modulo
```

Esto provee una nueva variable con el nombre del módulo que contiene todas las definiciones importadas. Para acceder a las definiciones del módulo, se acceden mediante esta variable, punto (`.`) y el nombre de la definición. Es similar a acceder a atributos o métodos de la instancia de una clase:

```
import ejemplo


print(ejemplo.mi_variable)
ejemplo.saludar()
```

```
10
¡Hola!
```

Importación completa con un alias

```
import modulo as alias
```

Funciona de igual forma a la anterior, pero es posible cambiar el nombre de la variable que contiene las definiciones del módulo. Esta opción es conveniente cuando el nombre puede generar confusión con otras definiciones del archivo en ejecución. También es conveniente cuando el nombre del módulo importado es muy largo y se quiere reducir.

```
import ejemplo as ej


print(ej.mi_variable)
ej.saludar()

ejemplo = "Otra variable"
print(ejemplo)
```

```
10
¡Hola!
Otra variable
```

Importación parcial

```
from modulo import variable, funcion, Clase
```

Este tipo de sentencia permite ahorrarse completamente utilizar una variable para acceder a los contenidos del módulo y permite utilizar directamente los nombres de esas definiciones en el archivo en ejecución. Se llama parcial, porque a su vez permite elegir qué definiciones se están importando y cuáles no:

```
from ejemplo import mi_variable, MiClase


print(mi_variable)
instancia = MiClase(23)
print(instancia.argumento)
```

```
10
23
```

```
saludar()  # No se importó en la sentencia, así que no está disponible para usarse
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 saludar()  # No se importó en la sentencia, así que no está disponible para usarse

NameError: name 'saludar' is not defined
```

Importación de un archivo dentro de una carpeta

```
import carpeta_con_modulo.modulo
import carpeta_con_modulo.modulo as modulo
from carpeta_con_modulo import modulo
from carpeta_con_modulo.modulo import variable
```

A veces, el tamaño de nuestro programa aumenta, y con este el número de archivos y módulos auxiliares que necesitamos. Por temas de orden, podemos crear subcarpetas en nuestro proyecto y ordenar los módulos auxiliares dentro de ellas. En estos casos, si queremos importar utilidades desde un módulo que se encuentra dentro de una subcarpeta, debemos seguir el formato `nombre_carpeta.nombre_modulo` para acceder a este.

Tal como en el ejemplo, podemos combinar esto con cualquiera de las tres formas de importación que mencionamos antes. Suponiendo que junto al archivo que queremos ejecutar, tenemos una carpeta llamada `carpeta_con_modulo` donde dentro existe un archivo llamado `modulo.py`, podemos importar de las siguientes formas:

```
import carpeta_con_modulo.modulo

print(carpeta_con_modulo.modulo.variable)
```

```
Sopaipillas
```

```
import carpeta_con_modulo.modulo as modulo

print(modulo.variable)
```

```
Sopaipillas
```

```
from carpeta_con_modulo import modulo

print(modulo.variable)
```

```
Sopaipillas
```

```
from carpeta_con_modulo.modulo import variable

print(variable)
```

```
Sopaipillas
```

Importación completa sin referencia al módulo

```
from modulo import *
```

Similar a la forma anterior, permite importar definiciones desde un módulo sin la necesidad de usar una variable para almacenarlos todos, pero el uso de `*` especifica que se desean importar **todas** las definiciones. El uso de estas sentencias se considera una **mala práctica**, ya que importa definiciones sin saber cuántas o cuáles son, y potencialmente importa nombres de definiciones que pueden coincidir con nombres de tu programa, sobrescribiendo definiciones y obteniendo resultados inesperados. Siempre se prefiere tener algún tipo de referencia al módulo de donde proviene una definición, o en su defecto definir, mediante importación parcial, cuáles son aquellas importadas. Esto hace que el código ajeno sea más legible y fácil de entender.

A lo largo del curso IIC2233 **penalizaremos el uso de esta práctica**.

Variable `__name__`

Al tener múltiples módulos que se importan unos entre ellos, con alias o no, a veces es necesario identificar módulos por sus nombres, e identificar si el módulo está siendo importado o directamente ejecutado. Aquí entra la variable global `__name__`, cuyo valor cambia en el contexto de cada módulo. En general, si se accede a esa variable dentro de un módulo, esta toma como valor (en forma de *string*) el nombre del módulo. Por ejemplo, importando con o sin alias, se puede ver el nombre de módulo accediendo a esta variable:

```
import ejemplo


ejemplo.__name__
```

```
'ejemplo'
```

```
import ejemplo as ej
```

```
ej.__name__
```

```
'ejemplo'
```

Hay una única excepción a la regla. Cuando se accede a `__name__` desde el módulo en que se inició la ejecución. Esto es, desde un módulo que no está siendo importado por otro módulo, entonces la variable `__name__` toma el valor `"__main__"`. Por ejemplo, al acceder a `__name__` en este documento obtenemos:

```
__name__
```

```
'__main__'
```

En el módulo adjunto `ejemplo2.py` se guarda el nombre de `__name__` en otra variable para ver su valor, y además se agrega una sentencia condicional que se ejecuta si es el módulo principal. Al importar el módulo, vemos que no entra al condicional, ya que no imprime nada, pero vemos que el nombre almacenado es efectivamente `"ejemplo2"`:

```
import ejemplo2
```

```
ejemplo2.mi_nombre
```

```
'ejemplo2'
```

En cambio, si ejecutamos ese condicional aquí, en el ”módulo” principal:

```
if __name__ == "__main__":
    print("Soy el módulo principal")
```

```
Soy el módulo principal
```

Sentencias como la anterior son muy comunes en Python: permiten que cierto código se ejecute **solo** cuando el archivo se corre directamente (por ejemplo, con `python3 modulo.py` en la terminal), y no cuando el módulo es importado por otro. Es la forma estándar de definir el punto de entrada de un programa, y también se usa en módulos que solo contienen definiciones, para incluir pequeñas demostraciones o pruebas de cómo utilizarlos al ser importados.

¿Y por qué usar módulos?

Al escribir pequeños programas es común escribirlos en un solo archivo, pero a medida que un programa crece en extensión y complejidad, esta alternativa no escala bien. El archivo se hace cada vez más largo y más difícil de organizar, y como consecuencia, se hace difícil de leer. Entender dicho programa no solo sería una ardua tarea para un tercero que no conoce previamente el código, sino que también lo es para el mismo autor del programa.

Por ejemplo, imaginemos que tenemos la tarea de crear un programa que, dado un archivo de contactos de usuarios, envíe un correo a cada uno con el contenido de otro archivo en formato PDF. Podemos identificar varias subtareas que realizar para crear este programa:

- Comprobar que el archivo de lista de contactos existe
- Identificar cuál es el formato del archivo de lista de contactos
- Leer el archivo de lista de contactos
- Transformar el contenido del archivo a una representación interna del programa
- Eliminar posibles contactos con datos faltantes
- Identificar el nombre y correo de cada usuario
- Comprobar que los correos son válidos
- Comprobar que el archivo PDF existe y es de ese formato
- Leer el archivo PDF
- Extraer el contenido del archivo PDF a una representación interna del programa
- Enviar el contenido por correo a usuarios

Un primer acercamiento para resolver el problema sería realizar una función dedicada a cada una de estas subtareas. Al nivel de detalle que se mostró, ya habría muchas funciones en un mismo archivo. Pero si lo pensamos, varias de estas funciones podrían agruparse por el tipo de tareas que realizan:

- Funciones para extraer y procesar la lista de usuarios:
  - Comprobar que el archivo de lista de contactos existe
  - Identificar cuál es el formato del archivo de lista de contactos
  - Leer el archivo de lista de contactos
  - Transformar el contenido del archivo a una representación interna del programa
  - Eliminar posibles contactos con datos faltantes
  - Identificar el nombre y correo de cada usuario
- Funciones para extraer el contenido de un archivo PDF:
  - Comprobar que el archivo PDF existe y es de ese formato
  - Leer el archivo PDF
  - Extraer el contenido del archivo PDF a una representación interna del programa
- Funciones para enviar correos:
  - Comprobar que los correos son válidos
  - Enviar el contenido por correo a usuarios

Es más, estas categorías de funciones son independientes entre sí: ninguna depende de una funcionalidad de la otra. Esto nos indica que una manera de organizar el código del programa es separando las funcionalidades en tres módulos: uno de contactos, uno de extracción de PDF y otro de emails, cada uno escrito en un archivo separado: `contactos.py`, `pdf.py` e `emails.py`.

Modularizar código provee varias ventajas. Primero: mejor organización y legibilidad de código. Esto trae como consecuencia otras ventajas: mayor facilidad de *debuggeo* y de inspección de código. Cuando un error aparece o se quiere buscar una función particular, es más fácil encontrar el origen si se encuentra en un módulo separado.

También permite reutilizar código fácilmente. El tener módulos completamente agnósticos al contexto donde se usan inicialmente permite que sea más fácil usarlos en otro programa. Si el módulo de lectura de archivos PDF del ejemplo anterior es capaz de leer cualquier archivo PDF que se le entregue, entonces ese módulo puede ser completamente reutilizado por otros programas, distintos al del ejemplo, que necesiten esa misma funcionalidad.

Abstracción de componentes

El concepto de modularización, además de organizar mejor el código de un programa, permite un mejor proceso de conceptualización del flujo de este. Al identificar distintos módulos, y las dependencias entre ellos, se permite la abstracción de los detalles de cada módulo y permite enfocarse en el funcionamiento general del programa.

Cada módulo pasa a ser una caja negra que provee funcionalidades. Un ejemplo tangible es una radio: es un objeto con perillas y botones que nos permite encenderla, cambiar la estación, cambiar el volumen y apagarla. Internamente, no sabemos cómo funciona ni cómo logra todas esas funciones, y no nos importa.

El trabajar con módulos permite un trabajo similar. Al importar las funcionalidades de un módulo en un programa, solo nos importa **lo que hacen** esas funcionalidades, y **no el cómo lo hacen**.

```
import radio


radio.encender()
radio.cambiar_estacion(99.3)
radio.subir_volumen(10)
radio.apagar()
```

Un ejemplo de código **real**, es la librería (y módulo) `random` de Python. En ella podemos encontrar varias funciones que permiten generar elementos aleatorios. ¿Cómo lo hace? No se sabe, a menos que se lea en detalle su código. Esto permite utilizarlo en cualquier contexto donde se necesite aleatoriedad, sin la preocupación de implementar efectivamente esa funcionalidad.

```
import random


print(random.randint(1, 10))
print(random.choice(["a", "b", "c"]))
```

```
7
c
```

Pensar en módulos permite separar las tareas de un programa en categorías y tipos; permite crear el mapa general de un programa de forma más sencilla y sin preocupación de los detalles. Hoy en día, la modularización es un concepto pilar del desarrollo de software. Pero este concepto no siempre fue obvio. En el siguiente video, Barbara Liskov, ganadora del Premio Turing 2008, habla sobre los tiempos en que este concepto recién surgió y conecta la idea de modularización con la programación orientada a objetos (uno de los contenidos del curso que veremos próximamente).