Comunicación entre procesos.

Procesos.

Un **proceso** es una instancia independiente de un programa en ejecución que se gestiona mediante el sistema operativo. A diferencia de los hilos, que comparten recursos en un solo proceso, los procesos tienen su propio espacio de memoria, por lo que solo pueden compartir datos a través de técnicas de comunicación entre procesos (IPC en sus siglas en inglés).

El módulo `multiprocessing` de Python permite crear y gestionar procesos mediante clases como **Process** y **Queue**, entre otras. Los procesos en `multiprocessing` se crean con la misma interfaz que `Thread`: bien mediante instancias de objetos `Process`, a los que se les pasa como argumentos la función que se ejecutará en el proceso y los argumentos de esa función, o bien mediante clases derivadas sobrecargando el constructor y el método `run()`. El proceso se inicia llamando al método `start()`, y se espera a que el proceso termine con el método `join()`.

Cada proceso tiene su propio intérprete de Python, lo que significa que **los errores de un proceso no afectarán a los demás procesos**. Además, **los procesos pueden ejecutarse en diferentes núcleos de la CPU**, lo que permite aprovechar mejor el procesamiento paralelo en sistemas multiprocesador.

Veamos un ejemplo:

```
from multiprocessing import Process
import time
import os

class Proceso(Process):
    "Define un proceso de ejemplo."

    def __init__(self, nombre: str):
        self.nombre = nombre
        super().__init__(name=nombre)

    def run(self):
        "Representa la actividad del proceso."
```

```
        while True:
            print(f"Saludos desde proceso '{self.nombre}' ({self.pid})")
            time.sleep(1)

proceso = Proceso("test")

proceso.start()
print(f'Desde el proceso {os.getpid()} lanzo {proceso.pid}')

time.sleep(30)
proceso.kill()  # Los procesos sí se pueden matar.

proceso.join()
```

Este código muestra cómo se puede crear y ejecutar un proceso en Python usando la clase `Process` como clase base.

En primer lugar, se define la clase `Proceso` que hereda de `Process`. La clase tiene un constructor que recibe el nombre del proceso, y un método `run()` que es el que se ejecuta en el proceso.

En el método `run()`, se crea un ciclo infinito que imprime un mensaje de saludo cada segundo, usando el nombre y el PID del proceso.

> **PID** (Process Identifier) es un número único asignado por el sistema operativo a un proceso en ejecución. Este número identifica de manera única al proceso durante toda su existencia y se utiliza para interactuar con el sistema operativo y otros procesos.

Después se crea una instancia de Proceso llamada proceso con el nombre `"test"`, y se inicia su ejecución con el método `start()`. También se imprime un mensaje que muestra lso PID del proceso padre y del proceso hijo.

A continuación, se espera durante 30 segundos, y se mata el proceso hijo con el método `kill()`. Finalmente, se llama al método `join()` para esperar a que el proceso hijo termine.

> Es importante mencionar que los procesos sí se pueden matar usando el método `kill()`, pero esta no es una práctica recomendada, ya que puede dejar recursos del sistema en un estado inconsistente. Lo ideal es que los procesos terminen su ejecución de manera natural.

Los procesos no comparten memoria entre sí de la forma en la que lo hacen los hilos, como se puede ver en el siguiente ejemplo:

```
from multiprocessing import Process
import time
from threading import Thread

class ListaSpam(Process):
    "Imprime lo que hay en una lista indefinidamente."

    def __init__(self, nombre: str, lista: list):
        self.nombre = nombre
```

```
        self.lista = lista
        super().__init__(name=nombre)

    def run(self):
        "Representa la actividad del proceso."

        while True:
            print("En mi lista tengo:", self.lista)
            time.sleep(1)


lista_compra = ["Huevos", "Leche"]
proceso = ListaSpam("Compra", lista_compra)
proceso.start()

for elemento in ["Patatas", "Carne", "Pescado"]:
    print("Añado", elemento)
    lista_compra.append(elemento)
    time.sleep(5)

time.sleep(5)

proceso.kill()  # Los procesos sí se pueden matar.
proceso.join()
```

El código comienza definiendo una clase `ListaSpam` que hereda de `Process`. Esta clase tiene un constructor que recibe un nombre y una lista, y el método run que se encargará de imprimir en pantalla el contenido de la lista cada segundo de forma indefinida.

Después, se crea una lista llamada `lista_compra` con los elementos `"Huevos"` y `"Leche"`. Se crea un objeto de la clase `ListaSpam` llamado `proceso` pasándole como argumentos `"Compra"` y `lista_compra`. Se inicia el proceso con `proceso.start()`.

A continuación, se entra en un bucle `for` que añade elementos a la lista `lista_compra` cada 5 segundos, imprimiendo en pantalla qué elemento se está añadiendo.

Después de añadir tres elementos, se espera otros 5 segundos y se mata el proceso con `proceso.kill()`. Se espera a que el proceso termine con `proceso.join()`.

El resultado de este código es que, a pesar de que se están añadiendo elementos a la lista `lista_compra`, el proceso proceso no los está viendo. **Esto se debe a que los procesos no comparten memoria, por lo que cada proceso tiene su propia copia de las variables y los objetos que se utilizan en él**. En este caso, la variable `lista_compra` en el proceso `proceso` es una copia de la lista original y no se actualiza automáticamente al cambiar la lista original.

> Cambia la clase base de `ListaSpam` de `Process` a `Thread` y observa como sí está cambia la lista. Esto se debe a que los hilos están en el mismo proceso y comparten memoria.

Comunicación entre procesos (IPC).

La **comunicación entre procesos** es un mecanismo que permite que los procesos en un sistema operativo se comuniquen entre sí, compartan recursos y realicen tareas de forma cooperativa. En sistemas operativos modernos, cada proceso se ejecuta en su propio espacio de memoria y no puede acceder directamente a la memoria de otros procesos. Como resultado, si un proceso necesita comunicarse con otro, debe hacerlo a través de algún mecanismo proporcionado por el sistema operativo.

La comunicación entre procesos puede ser necesaria en diversas situaciones, por ejemplo:

- Cuando dos o más procesos deben colaborar en una tarea común y deben intercambiar datos para hacerlo de manera efectiva.
- Cuando se requiere compartir recursos, como archivos o dispositivos, entre diferentes procesos en un sistema operativo.
- Cuando se necesitan servicios de red en un sistema distribuido para permitir que los procesos se comuniquen entre sí en diferentes máquinas.

Existen varios mecanismos de comunicación entre procesos, entre los que se incluyen:

- **Pipes y colas**: son canales unidireccionales que permiten a los procesos enviar y recibir datos.
- **Memoria compartida**: permite que varios procesos accedan a una región de memoria compartida para compartir datos.
- **Sockets**: son una forma común de comunicación entre procesos en redes, que permite la comunicación entre procesos en diferentes máquinas.
- **Archivos compartidos**: permiten que varios procesos accedan a un archivo compartido para compartir datos.

En resumen, la comunicación entre procesos es esencial para la cooperación y el intercambio de recursos entre procesos en un sistema operativo, y los mecanismos utilizados dependen de los requisitos específicos de la tarea o sistema en cuestión.

En este curso profundizaremos en **colas** y **sockets**.

Colas (multiprocessing).

Las colas en `multiprocessing` tienen la misma interfaz que las que se usan en `Thread`.

```
from multiprocessing import Process, Queue
import time

def productor(q):
    for i in range(5):
        mensaje = f"Mensaje {i}"
        q.put(mensaje)
        print(f"Productor: {mensaje}")
        time.sleep(1)

def consumidor(q):
    while True:
        mensaje = q.get()
        if mensaje == 'FIN':
            print("Consumidor: Fin del programa")
```

```
            break
        print(f"Consumidor: {mensaje}")

if __name__ == '__main__':
    queue = Queue()
    proceso_productor = Process(target=productor, args=(queue,))
    proceso_consumidor = Process(target=consumidor, args=(queue,))
    proceso_productor.start()
    proceso_consumidor.start()
    proceso_productor.join()
    queue.put('FIN')
    proceso_consumidor.join()
```

En este ejemplo, se crean dos procesos: un productor y un consumidor. El productor pone mensajes en una cola compartida mediante la función `put()`, y el consumidor los lee mediante la función `get()`. La función `Queue` se utiliza para crear la cola compartida. El programa se ejecuta hasta que el productor ha terminado de enviar todos los mensajes, momento en el que se envía un mensaje especial (`'FIN'`) a la cola para indicar que el consumidor debe finalizar la ejecución.

Este es solo un ejemplo básico, pero la comunicación mediante colas puede ser muy útil en situaciones en las que se necesite compartir información entre procesos de manera segura y eficiente.

Sockets

Los **sockets** son un mecanismo de comunicación que permiten que los procesos del mismo y diferentes dispositivos se comuniquen entre sí. En términos más simples, los sockets son un punto final de una conexión bidireccional entre dos programas (procesos).

Los sockets dan una interfaz común a diferentes protocolos de comunicación, lo que permite una misma aplicación que es capaz de comunicarse de adaptarse a diferentes dispositivos y protocolos.

Los sockets trabajan a muy bajo nivel en el sistema operativo y los datos que pasamos a través de ellos **están codificados en binario**.

Datos en binario.

En Python, los datos binarios se pueden representar utilizando dos tipos de datos principales: `bytes` y `bytearray`.

`bytes` es un objeto inmutable y secuencial que representa una secuencia de bytes. Se puede inicializar utilizando una **cadena de bytes** (una cadena normal precedida de `b`), un iterable de enteros (lista o tupla) o cualquier objeto objeto que implemente el método `__bytes__`. Funcionan igual que las cadenas de texto `str`y se pueden acceder a los elementos de un objeto `bytes` mediante índices, y también se pueden utilizar operadores de rebanado para acceder a partes del objeto.

```
class A:
   a = 10
   def __bytes__(self)-> bytes:
        return bytes([self.a])
```

```
obj = A()

datos_lit = b'\x02\x03'  # Construcción literal [2, 3]
datos_lista = bytes([2, 3]) # Desde un iterable [2, 3]
datos_clase = bytes(a)  # Conversión explícita [10]
```

También se puede convertir una cadena de texto a formato binario mediante el método `encode()` de `str` y volver de nuevo a texto mediante `decode()` de `bytes`:

```
texto = "Hola a todos"
texto_binario = texto.encode()
print(texto_binario.decode())
```

> Por defecto la codificación es **utf-8** para ambos métodos.

Por otro lado, `bytearray` es un objeto mutable y secuencial que representa una secuencia de bytes modificable. Se puede inicializar de la misma manera que `bytes`. Al ser mutable, se puede modificar directamente los elementos de un objeto bytearray.

```
datos = bytes([0x01, 0x02, 0x03])
datos[0] = 0x10  # Error! Inmutable.

array_datos = bytearray(datos)
array_datos[0] = 0x10  # Ok

datos = bytes(array_datos)
```

Ambos tipos de datos son muy útiles para trabajar con datos binarios, como en la lectura o escritura de archivos binarios, la comunicación a través de sockets o el cifrado de datos.

Sockets de red.

Los sockets más conocidos y usados son los **sockets de red**. Un socket de red se define por una **dirección IP**, un **número de puerto** y un **protocolo de comunicación**. Los sockets pueden ser utilizados para establecer diferentes tipos de conexiones, como **TCP** (Transmission Control Protocol) para conexiones confiables y orientadas a conexión, y **UDP** (User Datagram Protocol) para conexiones sin conexión y no confiables.

Los **sockets de red** son una herramienta fundamental para la comunicación de aplicaciones en red, ya que permiten que los programas interactúen con otros programas que se ejecutan en dispositivos separados. Es la base de internet e incluso son utilizados entre aplicaciones dentro del mismo sistema.

En el contexto de sockets, encontramos varias definiciones interesantes:

- **Dirección IP**: la dirección IP es una dirección única que identifica un dispositivo en una red. Es un número de cuatro bytes (en IPv4) o de dieciséis bytes (en IPv6) que se utiliza para direccionar los paquetes de datos a su destino en la red.

- **Puerto**: el puerto es un número entero de 16 bits que se utiliza para identificar un servicio específico en un dispositivo. Es decir, cuando un paquete de datos llega a un dispositivo, el puerto se utiliza para dirigir ese paquete a la aplicación o servicio adecuado en ese dispositivo. Los puertos más comunes están asignados a servicios bien conocidos, como el puerto 80 para el tráfico web o el puerto 25 para el tráfico de correo electrónico.
- **Protocolo**: el protocolo es un conjunto de reglas y procedimientos que se utilizan para permitir que los dispositivos de una red se comuniquen entre sí. Ejemplos de protocolos comunes son TCP (Protocolo de Control de Transmisión) y UDP (Protocolo de Datagrama de Usuario).
- **Servidor**: el servidor es aquel proceso que está a la espera de conexiones.
- **Cliente**: el cliente es aquel proceso que se conecta a un servidor mediante una petición. Si el servidor acepta se establece una conexión que dura hasta que uno de los dos la cierra.

Veamos un ejemplo sencillo de creación de sockets de red:

```
# Código del servidor #

import socket

# Creamos un objeto socket
servidor_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Configuramos el puerto y la dirección del servidor
host = 'localhost'  # Solo escucha dentro del ordenador
port = 51000
servidor_socket.bind((host, port))  # Enlaza el servidor a esa dirección y
puerto

# Esperamos una conexión entrante
servidor_socket.listen(1)
print(f"Servidor escuchando en {host}:{port}")

# Aceptamos la conexión entrante
cliente_socket, cliente_direccion = servidor_socket.accept()  # Se bloquea
hasta que llega una conexión
print(f"Conexión entrante desde {cliente_direccion}")

# Enviamos un mensaje al cliente
mensaje = "Hola, cliente!"
cliente_socket.send(mensaje.encode())

cliente_socket.close()
servidor_socket.close()
```

```
# Código del cliente #

import socket
```

```
cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = 'localhost'  # Solo escucha dentro del ordenador
port = 51000
cliente.connect((host, port))

data = cliente.recv(1024)  # De argumento el máximo de bytes que vamos a
recibir.

print(data.decode())
```

Estos dos códigos implementan un cliente y un servidor que se comunican a través de sockets utilizando el protocolo TCP. En el código del servidor, se crea un objeto `socket`, se configura el puerto y la dirección del servidor, y luego se espera una conexión entrante. Cuando llega una conexión entrante, se acepta la conexión y se envía un mensaje al cliente. Finalmente, se cierran las conexiones y el socket del servidor.

En el código del cliente, se crea un objeto `socket` y se establece una conexión con el servidor utilizando la dirección y el puerto especificados. Luego, se recibe un mensaje del servidor con un tamaño máximo de 1024 bytes y se imprime en la consola.

Sobre direcciones IP

Existen unas direcciones IP especiales que se deben conocer:

- `127.0.0.1`: Se refiere a la propia máquina.
- `0.0.0.0`: Cuando asignamos esta dirección a un servidor le decimos que acepte peticiones de cualquier dispositivo en la red.

Existe un mecanismo llamado DNS que permite asignarle nombres direcciones a la red. Por defecto, la red `127.0.0.1` se llama `localhost`.

Métodos de `socket`.

En Python, los sockets tienen varios métodos que permiten configurar y gestionar conexiones entre un cliente y un servidor. A continuación se explican los métodos más comunes:

- `bind((host, port))`: Este método enlaza el socket del servidor a una dirección y un puerto específicos. Debe especificarse una tupla con la dirección IP del host y el número de puerto que se desea usar. En el caso del cliente, se utiliza el método `connect((host, port))` para conectarse a la dirección del servidor.
- `listen(backlog)`: Este método permite al servidor esperar conexiones entrantes de clientes. El argumento `backlog` indica el número máximo de conexiones pendientes que se permiten en la cola del servidor.
- `accept()`: Este método bloquea la ejecución hasta que llega una conexión entrante, momento en el que devuelve una tupla con un nuevo objeto `socket` y la dirección del cliente que se ha conectado.
- `send(data)`: Este método envía los datos al otro extremo de la conexión, ya sea el cliente o el servidor. Los datos se deben pasar como un objeto de tipo `bytes`.

- `recv(buffer_size)`: Este método recibe los datos enviados por el otro extremo de la conexión. El argumento `buffer_size` indica el número máximo de bytes que se pueden recibir en una sola llamada a `recv()`. El método devuelve un objeto de tipo `bytes` con los datos recibidos.
- `close()`. se utiliza para cerrar la conexión del socket. Este método es importante para liberar los recursos del sistema que se han asignado al socket, como el puerto de escucha y cualquier otro recurso relacionado con la conexión. Cuando se llama al método close(), se envía un mensaje de cierre al otro extremo de la conexión, lo que indica que se va a cerrar la conexión. Después de enviar este mensaje, se espera una respuesta de cierre del otro extremo, y una vez que se recibe la respuesta, se liberan los recursos.

En resumen, `bind()` y `listen()` se usan en el servidor para esperar conexiones entrantes, mientras que `connect()` se usa en el cliente para conectarse al servidor. Una vez establecida la conexión, `send()` y `recv()` se usan para enviar y recibir datos entre el cliente y el servidor. Finalmente, `close()` cierra el servidor.

Ejercicio de clase.

Realiza un servidor "eco" que acepte conexiones y devuelva el mismo mensaje que recibe.

Manejar múltiples sockets (`select`).

La función `select()` del módulo `select` se utiliza para monitorear múltiples sockets en busca de actividad de entrada/salida. Es útil cuando se espera que haya múltiples conexiones entrantes y deseas manejarlas todas en una sola aplicación.

La función `select()` espera hasta que uno o más de los sockets en las listas de espera esté listo para leer, escribir o recibir excepciones. La función toma tres listas de sockets: una lista de sockets para leer, una lista de sockets para escribir y una lista de sockets que se están monitoreando para excepciones. El parámetro de tiempo de espera es opcional y especifica cuánto tiempo (en segundos) se esperará como máximo antes de que se salga de la función `select()`.

El método `select()` devuelve también tres listas de sockets. La primera lista contiene los **sockets listos para leer**, la segunda contiene los **sockets listos para escribir** y la tercera contiene los sockets que han generado una excepción.

En general, `select()` se utiliza en bucles para monitorear los sockets y manejar la actividad de entrada/salida según sea necesario. Esto significa que, en lugar de tener un hilo o proceso para cada conexión entrante, podemos manejar múltiples conexiones entrantes en una sola aplicación y, por lo tanto, reducir la sobrecarga del sistema.

Aquí tienes un ejemplo básico de cómo usar la función select de Python para monitorear varios sockets al mismo tiempo:

```
import socket
import select

# Creamos un objeto socket para escuchar conexiones entrantes
servidor_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
servidor_socket.bind(('0.0.0.0', 50600))
```

```
servidor_socket.listen()

# Creamos una lista de sockets que queremos monitorear con select
sockets = [servidor_socket]

while True:
    # Usamos select para esperar por eventos en los sockets de la lista
    lectura, escritura, excepcion = select.select(sockets, [], [])

    # Recorremos la lista de sockets que tienen datos para leer
    for socket in lectura:
        # Si el socket es el servidor, aceptamos una nueva conexión
        if socket == servidor_socket:
            cliente_socket, direccion = servidor_socket.accept()
            sockets.append(cliente_socket)
            print(f'Nueva conexión entrante desde {direccion}')

        # Si no es el servidor, recibimos datos del socket
        else:
            datos = socket.recv(1024)
            if datos:
                print(f'Datos recibidos de {socket.getpeername()}:
{datos.decode()}')
            else:
                print(f'{socket.getpeername()} se ha desconectado')
                sockets.remove(socket)
                socket.close()
```

En este ejemplo, creamos un objeto `socket` que escucha conexiones entrantes en el puerto `50600`. Luego, creamos una lista de sockets que queremos monitorear con `select`. En este caso, la lista solo contiene el objeto de `socket` del servidor.

Dentro del bucle principal, usamos la función `select` para esperar por eventos en los sockets de la lista. Cuando hay datos disponibles para leer en un `socket`, lo manejamos de acuerdo a si es el objeto del socket del servidor o un socket de cliente.

Si `socket` es el objeto del socket del servidor, aceptamos una nueva conexión entrante y agregamos el socket del cliente a la lista de sockets a monitorear. Si `socket` es un socket de cliente, recibimos los datos del socket y los imprimimos en la consola. Si el socket está cerrado, lo eliminamos de la lista de sockets y lo cerramos.

Ejemplo servidor de chat bidireccional.

> Debido a que trabajamos a bajo nivel, la creación y manejo de socket es algo compleja. Se dará un tutorial en el **ejercicio E7**. Ahora, paso a paso se realizará el siguiente de ejercicio con el fin de comprender mejor los sockets.

Profesor y alumnos

Realiza un servidor con sockets que permita que todos los clientes conectados a él chateen entre ellos. Los mensajes deberán tener la siguiente estrucutura:

```
{
    "user": "nombre_usuario",
    "message": "mensaje de texto"
}
```

Para ello usa hilos o select para gestionar las múltiples conexiones.

Alumnos

Crea un cliente para el anterior ejercicio. Debe pedir un nombre de usuario antes de empezar a mandar mensajes y al recibirlos y mostrarlos debe tener el siguiente formato:

```
[commodoro] - Hola gente! :)
[alvarito16] - Hola Mariano! Qué tal el finde?
...
[usuario] - Mensaje
```

Sockets en Linux.

Además de los sockets de red, que se utilizan para la comunicación a través de redes, también existen otros tipos de sockets que se utilizan para la comunicación entre procesos en una misma máquina. Estos son exclusivos de Unix/Linux y no están disponibles en Windows. A continuación se describen algunos de ellos:

- **Unix Domain Sockets**: Estos sockets se utilizan para la comunicación entre procesos en una misma máquina Unix o Unix-like (como Linux o macOS). A diferencia de los sockets de red, los Unix Domain Sockets no se comunican a través de la red, sino que utilizan el sistema de archivos del sistema operativo para la comunicación.
- **Named Pipes**: Los named pipes, también conocidos como FIFOs, son otro tipo de comunicación interproceso en Unix y Unix-like. A diferencia de los Unix Domain Sockets, los named pipes se comunican a través del sistema de archivos, pero en lugar de utilizar un socket, utilizan un archivo FIFO especial.
- **Memory-mapped Files**: Este tipo de comunicación interproceso se basa en el uso de archivos mapeados en memoria compartida. Los procesos pueden leer y escribir en el archivo mapeado en memoria, y los cambios se reflejan en todos los procesos que comparten la misma memoria.
- **Sockets de iterfaces personalizadas**: aplican la capa de abstracción común de los sockets a protocolos de comunicación como `CAN` o `RS232`.

Los sockets son una abstracción utilizada para la comunicación entre procesos, y existen diferentes tipos de sockets para diferentes situaciones y plataformas. Los sockets de red son los más comunes y se utilizan para la comunicación a través de redes, pero también existen otros tipos de sockets que se utilizan para la comunicación entre procesos en una misma máquina.

En Linux tenemos más herramientas que en Windows para desarrollar comunicación entre procesos y esa es una de las razones por las que se suele preferir aplicaciones en Linux/Unix sobre Windows cuando se

trabaja en sistemas en tiempo real.