Programación de servicios y procesos en Python

1. Programas y procesos

1.1 Programas y procesos

Un programa es un conjunto de instrucciones escritas en un lenguaje de programación que realiza una tarea específica. Un proceso es una instancia en ejecución de un programa.

```
def ejemplo_programa():
    print("Esto es un programa")

ejemplo_programa()
```

1.2 Multitarea

La multitarea es la capacidad de un sistema operativo para ejecutar múltiples tareas (procesos) al mismo tiempo.

```
import threading

def tarea(nombre):
    print(f"Tarea {nombre} en ejecución")

hilo1 = threading.Thread(target=tarea, args=("1",))
hilo2 = threading.Thread(target=tarea, args=("2",))
hilo1.start()
hilo2.start()
```

1.3 Procesos y sistemas monoprocesadores

y multiprocesadores

1.3.1 Sistemas monoprocesadores

Un sistema monoprocesador tiene un solo procesador que ejecuta tareas secuencialmente.

```
def tarea_secuencial():
    print("Tarea en un sistema monoprocesador")

tarea_secuencial()
```

1.3.2 Sistemas multiprocesadores

Un sistema multiprocesador tiene múltiples procesadores que pueden ejecutar tareas simultáneamente.

```
import multiprocessing

def tarea_multiprocesador(nombre):
    print(f"Tarea {nombre} en un sistema multiprocesador")

if __name__ == "__main__":
                                        proceso1         =
multiprocessing.Process(target=tarea_multiprocesador,
args=("1",))
                                        proceso2         =
multiprocessing.Process(target=tarea_multiprocesador,
args=("2",))
    proceso1.start()
    proceso2.start()
```

1.4 Ventajas e inconvenientes de la programación concurrente

La programación concurrente permite la ejecución simultánea de múltiples tareas, mejorando la eficiencia. Sin embargo, puede introducir problemas de sincronización y complejidad adicional.

```
def tarea_concurrente():
    print("Ejemplo de programación concurrente")

import threading

hilo = threading.Thread(target=tarea_concurrente)
hilo.start()
```

1.5 Kernel (o núcleo) del sistema operativo y llamadas al sistema

El kernel es el núcleo del sistema operativo que gestiona los recursos del sistema y las llamadas al sistema son las interfaces que los programas utilizan para interactuar con el kernel.

```
import os

def llamada_sistema():
    os.system("echo Llamada al sistema")

llamada_sistema()
```

1.6 Estados de ejecución de un proceso

Un proceso puede estar en varios estados, como ejecutándose, esperando o terminado.

```
import time

def proceso():
    print("Proceso en ejecución")
    time.sleep(1)
    print("Proceso terminado")

proceso()
```

1.7 Hilos y procesos

Un hilo es la unidad más pequeña de un proceso que puede ser ejecutada. Un proceso puede contener múltiples hilos.

```
import threading

def hilo():
    print("Hilo en ejecución")

hilo1 = threading.Thread(target=hilo)
hilo2 = threading.Thread(target=hilo)
hilo1.start()
hilo2.start()
```

1.8 Servicios

1.8.1 Servicios en Windows

Los servicios en Windows son programas que se ejecutan en segundo plano y se inician automáticamente durante el arranque del sistema.

```
import subprocess

def servicio_windows():
    subprocess.run(["sc", "query"])

servicio_windows()
```

1.8.2 Servicios en Linux

Los servicios en Linux son similares a los de Windows y se gestionan mediante el comando systemctl.

```
import subprocess

def servicio_linux():
    subprocess.run(["systemctl", "status"])
```

```
servicio_linux()
```

1.9 Concurrencia en Python

1.9.1 La clase Runtime

La clase Runtime permite interactuar con el entorno de ejecución en Python. En Python, podemos utilizar la librería os para realizar tareas similares.

```
import os

def obtener_runtime():
    print(os.name)

obtener_runtime()
```

1.9.2 Las clases Process y ProcessBuilder

En Python, las clases Process y ProcessBuilder se utilizan para crear y controlar procesos. En Python, podemos usar la librería subprocess para tareas similares.

```
import subprocess

def crear_proceso():
      proceso = subprocess.run(["echo", "Hola, mundo"], capture_output=True, text=True)
    print(proceso.stdout)

crear_proceso()
```

1.9.3 Redirección de entrada y salida estándares y de error

La redirección de entrada y salida estándares y de error permite controlar dónde se envía la salida de un proceso. En Python, esto se puede hacer con la librería subprocess.

```
import subprocess

def redireccion_salida():
    with open("salida.txt", "w") as archivo_salida:
            subprocess.run(["echo", "Hola, mundo"],
stdout=archivo_salida)

redireccion_salida()
```

2. Programación de hilos en Python

2.1 Programación de hilos en Python

En Python, la programación de hilos se maneja utilizando la clase Thread y la interfaz Runnable. En Python, utilizamos el módulo threading para crear y manejar hilos.

```
import threading

def tarea():
    print("Hilo en ejecución")

hilo = threading.Thread(target=tarea)
hilo.start()
```

2.2 Creación de hilos en Python

La creación de hilos en Python se puede hacer extendiendo la clase Thread o implementando la interfaz Runnable. En Python, simplemente creamos instancias de la clase Thread y pasamos la función a ejecutar.

```
import threading

class MiHilo(threading.Thread):
    def run(self):
        print("Hilo creado con la clase MiHilo")
```

```
hilo = MiHilo()
hilo.start()
```

2.3 La clase Thread

La clase Thread en Python se utiliza para crear hilos. En Python, la clase Thread en el módulo threading cumple la misma función.

```
import threading

def ejecutar():
    print("Ejecutando hilo con la clase Thread")

hilo = threading.Thread(target=ejecutar)
hilo.start()
```

2.4 Sincronización de hilos

La sincronización de hilos es crucial para evitar condiciones de carrera y asegurar que los recursos compartidos se manejen correctamente.

2.4.1 Exclusión mutua. Condiciones de carrera y secciones críticas

La exclusión mutua evita que varios hilos accedan a un recurso compartido al mismo tiempo, lo que podría causar condiciones de carrera.

```
import threading

lock = threading.Lock()
contador = 0

def incrementar():
    global contador
    with lock:
        contador += 1
```

```
        print(f"Contador: {contador}")

hilos = [threading.Thread(target=incrementar) for _ in range(5)]
for hilo in hilos:
    hilo.start()
for hilo in hilos:
    hilo.join()
```

2.4.2 Bloqueo intrínseco, bloques de código sincronizados

El bloqueo intrínseco asegura que un bloque de código solo sea ejecutado por un hilo a la vez.

```
import threading

lock = threading.Lock()

def tarea_sincronizada():
    with lock:
        print("Bloque sincronizado")

hilo1 = threading.Thread(target=tarea_sincronizada)
hilo2 = threading.Thread(target=tarea_sincronizada)
hilo1.start()
hilo2.start()
hilo1.join()
hilo2.join()
```

2.4.3 Compartición de recursos. Interbloqueo

El interbloqueo ocurre cuando dos o más hilos esperan indefinidamente por recursos que los otros hilos poseen.

```
import threading

lock1 = threading.Lock()
lock2 = threading.Lock()
```

```
def tarea1():
    with lock1:
        print("Tarea 1 adquiere lock1")
        with lock2:
            print("Tarea 1 adquiere lock2")

def tarea2():
    with lock2:
        print("Tarea 2 adquiere lock2")
        with lock1:
            print("Tarea 2 adquiere lock1")

hilo1 = threading.Thread(target=tarea1)
hilo2 = threading.Thread(target=tarea2)
hilo1.start()
hilo2.start()
hilo1.join()
hilo2.join()
```

2.4.4 Compartición de recursos con bloqueo dependiente de su estado

La compartición de recursos con bloqueo dependiente de su estado asegura que los recursos solo se bloqueen cuando están en un estado apropiado para ser utilizados.

```
import threading

class Recurso:
    def __init__(self):
        self.estado = False
        self.lock = threading.Lock()

    def cambiar_estado(self):
        with self.lock:
            self.estado = not self.estado
            print(f"Estado cambiado a: {self.estado}")

recurso = Recurso()
hilo1 = threading.Thread(target=recurso.cambiar_estado)
hilo2 = threading.Thread(target=recurso.cambiar_estado)
```

```
hilo1.start()
hilo2.start()
hilo1.join()
hilo2.join()
```

2.4.5 Clases thread-safe o seguras para su uso en aplicaciones multihilo

Las clases thread-safe son seguras para su uso en aplicaciones multihilo porque manejan correctamente la sincronización interna.

```
import queue

cola_segura = queue.Queue()

def productor():
    for i in range(5):
        cola_segura.put(i)
        print(f"Producido: {i}")

def consumidor():
    while not cola_segura.empty():
        item = cola_segura.get()
        print(f"Consumido: {item}")

hilo1 = threading.Thread(target=productor)
hilo2 = threading.Thread(target=consumidor)
hilo1.start()
hilo2.start()
hilo1.join()
hilo2.join()
```

2.5 Interrupción de hilos

La interrupción de hilos permite detener la ejecución de un hilo desde otro hilo.

```
import threading
import time
```

```
def hilo_interrumpible():
    try:
        while True:
            print("Hilo en ejecución...")
            time.sleep(1)
    except threading.ThreadError:
        print("Hilo interrumpido")

hilo = threading.Thread(target=hilo_interrumpible)
hilo.start()
time.sleep(3)
hilo._stop()
```

2.6 Prioridades

En Python, no hay una forma directa de establecer la prioridad de un hilo, pero se puede simular utilizando diferentes intervalos de tiempo de espera.

```
import threading
import time

def hilo_prioritario():
    while True:
        print("Hilo prioritario")
        time.sleep(0.5)

def hilo_no_prioritario():
    while True:
        print("Hilo no prioritario")
        time.sleep(1)

hilo1 = threading.Thread(target=hilo_prioritario)
hilo2 = threading.Thread(target=hilo_no_prioritario)
hilo1.start()
hilo2.start()
```

2.7 Depuración (debugging) de

aplicaciones multihilo

La depuración de aplicaciones multihilo puede ser complicada debido a la concurrencia. Utiliza herramientas de depuración y agrega registros de seguimiento para ayudar en el proceso.

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
format='%(threadName)s: %(message)s')

def hilo_debug():
    logging.debug("Inicio del hilo")
    time.sleep(1)
    logging.debug("Fin del hilo")

hilo = threading.Thread(target=hilo_debug)
hilo.start()
hilo.join()
```

2.8 Mecanismos de alto nivel para concurrencia

2.8.1 Colecciones concurrentes

Las colecciones concurrentes permiten el acceso seguro a las estructuras de datos desde múltiples hilos.

```
import queue

cola_concurrente = queue.Queue()

def productor():
    for i in range(5):
        cola_concurrente.put(i)
        print(f"Producido: {i}")

def consumidor():
```

```
    while not cola_concurrente.empty():
        item = cola_concurrente.get()
        print(f"Consumido: {item}")

hilo1 = threading.Thread(target=productor)
hilo2 = threading.Thread(target=consumidor)
hilo1.start()
hilo2.start()
hilo1.join()
hilo2.join()
```

2.8.2 Variables atómicas

Las variables atómicas aseguran que las operaciones sobre ellas sean indivisibles.

```
from threading import Lock

class VariableAtomica:
    def __init__(self, valor):
        self.valor = valor
        self.lock = Lock()

    def incrementar(self):
        with self.lock:
            self.valor += 1
            return self.valor

variable_atomica = VariableAtomica(0)
print(variable_atomica.incrementar())
```

2.8.3 Números aleatorios

El módulo random de Python proporciona funciones para generar números aleatorios que son seguras para su uso en aplicaciones multihilo.

```
import random
import threading
```

```
def generar_aleatorio():
    print(f"Número aleatorio: {random.randint(1, 100)}")

hilos = [threading.Thread(target=generar_aleatorio) for _ in
range(5)]
for hilo in hilos:
    hilo.start()
for hilo in hilos:
    hilo.join()
```

2.9 Uso de concurrencia en juegos y simulaciones gráficas

La concurrencia es útil en juegos y simulaciones gráficas para manejar múltiples tareas simultáneamente, como la actualización de la lógica del juego y la renderización de gráficos.

```
import threading
import time

def actualizar_logica():
    while True:
        print("Actualizando lógica del juego")
        time.sleep(1)

def renderizar_graficos():
    while True:
        print("Renderizando gráficos")
        time.sleep(0.5)

hilo_logica = threading.Thread(target=actualizar_logica)
hilo_graficos = threading.Thread(target=renderizar_graficos)
hilo_logica.start()
hilo_graficos.start()
```

3. Comunicación en Red en Python

3.1 Distintos modelos de comunicaciones entre procesos

En Python, los procesos pueden comunicarse a través de diferentes modelos, como pipes, colas y sockets.

```
import multiprocessing

def worker(pipe):
    pipe.send('Hola desde el otro proceso')
    pipe.close()

parent_conn, child_conn = multiprocessing.Pipe()
p = multiprocessing.Process(target=worker, args=(child_conn,))
p.start()
print(parent_conn.recv())
p.join()
```

3.2 Modelo de niveles de TCP/IP

El modelo de niveles de TCP/IP se divide en cuatro capas: enlace, red, transporte y aplicación.

3.2.1 Niveles de enlace y de red

Los niveles de enlace y de red se encargan de la transmisión de datos a través de la red.

```
# En Python, estos niveles son manejados automáticamente por las bibliotecas de red.
# No hay una implementación directa de bajo nivel como en otros lenguajes.
```

3.2.2 Nivel de transporte

El nivel de transporte se encarga de la transmisión de datos entre dos hosts. En Python, esto se maneja comúnmente con

sockets.

```
import socket

def servidor_tcp():
        server_socket  =  socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    server_socket.bind(('localhost', 12345))
    server_socket.listen(1)
    print("Servidor esperando conexiones...")
    conn, addr = server_socket.accept()
    print(f"Conexión establecida con {addr}")
    conn.send(b"Hola, cliente!")
    conn.close()

def cliente_tcp():
        client_socket  =  socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    client_socket.connect(('localhost', 12345))
    print(client_socket.recv(1024))
    client_socket.close()

# Para probar, ejecuta primero el servidor_tcp() y luego el
cliente_tcp()
```

3.2.3 Nivel de aplicación

El nivel de aplicación se encarga de las comunicaciones específicas de la aplicación, como HTTP, FTP, etc.

```
import requests

def cliente_http():
                                        respuesta         =
requests.get('https://jsonplaceholder.typicode.com/posts')
    print(respuesta.json())

# Llama a la función cliente_http() para obtener los datos
```

3.3 Resolución de nombres

3.3.1 Resolución de nombres local

La resolución de nombres local traduce nombres de host a direcciones IP dentro de una red local.

```
import socket

def resolver_nombre_local():
    host = 'localhost'
    ip = socket.gethostbyname(host)
    print(f"La dirección IP de {host} es {ip}")

# Llama a la función resolver_nombre_local()
```

3.3.2 El protocolo DNS

El protocolo DNS se utiliza para la resolución de nombres en la red global, traduciendo nombres de dominio a direcciones IP.

```
import socket

def resolver_dns():
    dominio = 'www.example.com'
    ip = socket.gethostbyname(dominio)
    print(f"La dirección IP de {dominio} es {ip}")

# Llama a la función resolver_dns()
```

3.4 Clases de Python para comunicaciones en red

En Python, utilizamos bibliotecas como socket y requests para manejar las comunicaciones en red.

```
import socket
```

```
# Ejemplo básico de un servidor y cliente TCP usando la biblioteca socket
# Se muestra en las secciones 3.2.2 y 3.9.1
```

3.5 Clases de Python para interfaces de red

En Python, la información de las interfaces de red se puede obtener usando la biblioteca psutil.

```
import psutil

def listar_interfaces_red():
    interfaces = psutil.net_if_addrs()
    for interface, direcciones in interfaces.items():
        print(f"Interface: {interface}")
        for direccion in direcciones:
            print(f"  Dirección: {direccion.address}")

# Llama a la función listar_interfaces_red()
```

3.6 Clases de Python para direcciones IP

En Python, el módulo ipaddress proporciona clases para manejar direcciones IP.

```
import ipaddress

def manejar_direcciones_ip():
    ip = ipaddress.ip_address('192.168.0.1')
    print(f"Dirección IP: {ip}")
    red = ipaddress.ip_network('192.168.0.0/24')
    for direccion in red:
        print(direccion)

# Llama a la función manejar_direcciones_ip()
```

3.7 Resolución de nombres con Python

En Python, la resolución de nombres se puede realizar con el módulo socket.

```
import socket

def resolver_nombres():
    dominio = 'www.google.com'
    ip = socket.gethostbyname(dominio)
    print(f"La dirección IP de {dominio} es {ip}")

# Llama a la función resolver_nombres()
```

3.8 Clases de Python para sockets de UDP

3.8.1 Programación de aplicaciones servidores y clientes basadas en UDP

Las aplicaciones basadas en UDP pueden ser implementadas usando el módulo socket en Python.

```
import socket

def servidor_udp():
       server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind(('localhost', 12345))
    print("Servidor UDP esperando datos...")
    data, addr = server_socket.recvfrom(1024)
    print(f"Recibido de {addr}: {data.decode()}")
    server_socket.sendto(b"Hola, cliente!", addr)

def cliente_udp():
       client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
     client_socket.sendto(b"Hola, servidor!", ('localhost', 12345))
    data, addr = client_socket.recvfrom(1024)
```

```
    print(f"Recibido del servidor: {data.decode()}")

# Para probar, ejecuta primero el servidor_udp() y luego el cliente_udp()
```

3.8.2 Programación distribuida basada en UDP

La programación distribuida basada en UDP permite la comunicación entre múltiples procesos distribuidos en diferentes máquinas.

```
# Similar al ejemplo anterior, pero puedes extender la funcionalidad
# para manejar múltiples clientes y servidores en diferentes máquinas.
```

3.8.3 Programación de servidores multihilo basados en UDP

Los servidores UDP multihilo pueden manejar múltiples clientes simultáneamente.

```
import threading

def manejar_cliente(data, addr):
    print(f"Recibido de {addr}: {data.decode()}")
    server_socket.sendto(b"Hola, cliente!", addr)

def servidor_udp_multihilo():
    global server_socket
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind(('localhost', 12345))
    print("Servidor UDP multihilo esperando datos...")
    while True:
        data, addr = server_socket.recvfrom(1024)
         threading.Thread(target=manejar_cliente, args=(data, addr)).start()

# Ejecuta servidor_udp_multihilo() para iniciar el servidor
```

3.9 Clases de Python para sockets de TCP

3.9.1 Programación de aplicaciones servidores y clientes basadas en TCP

Las aplicaciones basadas en TCP pueden ser implementadas usando el módulo socket en Python.

```
import socket

def servidor_tcp():
        server_socket  =  socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 12345))
    server_socket.listen(1)
    print("Servidor esperando conexiones...")
    conn, addr = server_socket.accept()
    print(f"Conexión establecida con {addr}")
    conn.send(b"Hola, cliente!")
    conn.close()

def cliente_tcp():
        client_socket  =  socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 12345))
    print(client_socket.recv(1024))
    client_socket.close()

# Para probar, ejecuta primero el servidor_tcp() y luego el cliente_tcp()
```

3.9.2 Programación de servidores multihilo basados en TCP

Los servidores TCP multihilo pueden manejar múltiples conexiones simultáneamente.

```
import threading
```

```
def manejar_cliente(conn, addr):
    print(f"Conexión establecida con {addr}")
    conn.send(b"Hola, cliente!")
    conn.close()

def servidor_tcp_multihilo():
       server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    server_socket.bind(('localhost', 12345))
    server_socket.listen(5)
    print("Servidor TCP multihilo esperando conexiones...")
    while True:
        conn, addr = server_socket.accept()
         threading.Thread(target=manejar_cliente, args=(conn,
addr)).start()

# Ejecuta servidor_tcp_multihilo() para iniciar el servidor
```

3.9.3 Programación distribuida basada en TCP

La programación distribuida basada en TCP permite la comunicación entre múltiples procesos distribuidos en diferentes máquinas.

```
# Similar al ejemplo de servidor multihilo, pero puedes
extender
# la funcionalidad para manejar múltiples clientes y
servidores
# en diferentes máquinas.
```

4. Comunicación en Red en Python

4.1 Clases de Python para comunicaciones en red

4.1.1 Clases de alto nivel del paquete estándar socket

En Python, el módulo socket proporciona clases de alto nivel

para la comunicación en red.

```
import socket

# Ejemplo básico de un servidor TCP usando la biblioteca socket
def servidor_tcp():
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 12345))
    server_socket.listen(1)
    print("Servidor esperando conexiones...")
    conn, addr = server_socket.accept()
    print(f"Conexión establecida con {addr}")
    conn.send(b"Hola, cliente!")
    conn.close()

def cliente_tcp():
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 12345))
    print(client_socket.recv(1024))
    client_socket.close()

# Para probar, ejecuta primero el servidor_tcp() y luego el cliente_tcp()
```

4.1.2 Ftplib

En Python, se pueden usar bibliotecas como ftplib para gestionar FTP.

```
from ftplib import FTP

def conectar_ftp():
    ftp = FTP('ftp.dlptest.com')
    ftp.login()
    ftp.retrlines('LIST')
    ftp.quit()
```

```
# Llama a la función conectar_ftp() para conectarte a un servidor FTP y listar archivos
```

4.1.3 Otras bibliotecas de clases

Además de socket y ftplib, existen otras bibliotecas en Python como requests para HTTP y paramiko para SSH.

```
import requests

def cliente_http():
    respuesta = requests.get('https://jsonplaceholder.typicode.com/posts')
    print(respuesta.json())

# Llama a la función cliente_http() para obtener datos desde una URL
```

4.2 El protocolo TELNET

4.2.1 Uso de TELNET para ejecución de una shell remota

En Python, se puede usar el módulo telnetlib para interactuar con servidores TELNET.

```
import telnetlib

def conectar_telnet():
    tn = telnetlib.Telnet('towel.blinkenlights.nl')
    tn.interact()

# Llama a la función conectar_telnet() para conectarte a un servidor TELNET
```

4.2.2 Uso de TELNET como teletipo sobre TCP

El módulo telnetlib también permite usar TELNET como teletipo sobre TCP.

4.2.3 Equivalentes a TELNET para teletipo sobre conexión segura con TLS/SSL

Para conexiones seguras, se puede usar paramiko para SSH en lugar de TELNET.

```
import paramiko

def conectar_ssh():
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect('example.com',  username='user',
password='passwd')
    stdin, stdout, stderr = ssh.exec_command('ls')
    print(stdout.read().decode())
    ssh.close()

# Llama a la función conectar_ssh() para conectarte a un servidor SSH y ejecutar un comando
```

4.2.4 Uso del protocolo TELNET

En Python, se puede usar telnetlib para interactuar con TELNET.

4.3 El protocolo SSH

En Python, el módulo paramiko se utiliza para implementar SSH.

```
import paramiko

def conectar_ssh():
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect('example.com',  username='user',
password='passwd')
    stdin, stdout, stderr = ssh.exec_command('ls')
    print(stdout.read().decode())
    ssh.close()
```

```
# Llama a la función conectar_ssh() para conectarte a un servidor SSH y ejecutar un comando
```

4.4 Servicios de transferencia de ficheros

4.4.1 FTP anónimo y autenticado

```
from ftplib import FTP

def conectar_ftp():
    ftp = FTP('ftp.dlptest.com')
    ftp.login()  # Conexión FTP anónima
    ftp.retrlines('LIST')
    ftp.quit()

def conectar_ftp_autenticado():
    ftp = FTP('ftp.example.com')
    ftp.login(user='username', passwd='password')
    ftp.retrlines('LIST')
    ftp.quit()

# Llama a la función conectar_ftp() o conectar_ftp_autenticado() según el caso
```

4.4.2 FTP activo y pasivo

El modo de conexión FTP se puede cambiar entre activo y pasivo en Python.

```
from ftplib import FTP

def conectar_ftp_pasivo():
    ftp = FTP('ftp.example.com')
    ftp.set_pasv(True)  # Modo pasivo
    ftp.login(user='username', passwd='password')
    ftp.retrlines('LIST')
    ftp.quit()
```

```
def conectar_ftp_activo():
    ftp = FTP('ftp.example.com')
    ftp.set_pasv(False)  # Modo activo
    ftp.login(user='username', passwd='password')
    ftp.retrlines('LIST')
    ftp.quit()

# Llama a la función conectar_ftp_pasivo() o conectar_ftp_activo() según el caso
```

4.4.3 FTP en modo binario y de texto

Los modos binario y de texto se pueden usar para transferencias específicas en FTP.

```
from ftplib import FTP

def descargar_archivo_binario():
    ftp = FTP('ftp.example.com')
    ftp.login(user='username', passwd='password')
    with open('archivo.bin', 'wb') as f:
        ftp.retrbinary('RETR archivo_remoto.bin', f.write)
    ftp.quit()

def descargar_archivo_texto():
    ftp = FTP('ftp.example.com')
    ftp.login(user='username', passwd='password')
    with open('archivo.txt', 'w') as f:
        ftp.retrlines('RETR archivo_remoto.txt', lambda line: f.write(line + '\n'))
    ftp.quit()

# Llama a la función descargar_archivo_binario() o descargar_archivo_texto() según el caso
```

4.4.4 Uso del protocolo FTP

En Python, se puede usar ftplib para interactuar con servidores FTP.

4.5 Servicios de correo electrónico

4.5.1 Instalación y configuración de programas servidores y clientes para correo electrónico

En Python, se puede usar smtplib para enviar correos electrónicos y poplib para recibir correos.

```
import smtplib
from email.mime.text import MIMEText

def enviar_correo():
    msg = MIMEText('Este es el cuerpo del mensaje.')
    msg['Subject'] = 'Asunto del mensaje'
    msg['From'] = 'tu_correo@example.com'
    msg['To'] = 'destinatario@example.com'

    with smtplib.SMTP('smtp.example.com') as server:
        server.login('usuario', 'contraseña')
        server.send_message(msg)

# Llama a la función enviar_correo() para enviar un correo electrónico
```

4.5.2 Estructura de un mensaje de correo electrónico

La estructura de un mensaje de correo electrónico en Python se puede manejar usando el módulo email.

```
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

def crear_mensaje():
    msg = MIMEMultipart()
    msg['From'] = 'tu_correo@example.com'
    msg['To'] = 'destinatario@example.com'
    msg['Subject'] = 'Asunto del mensaje'
```

```
    cuerpo = MIMEText('Este es el cuerpo del mensaje.')
    msg.attach(cuerpo)
    return msg

# Llama a la función crear_mensaje() para crear un mensaje de correo electrónico
```

4.5.3 El protocolo POP3

El módulo poplib en Python se usa para acceder a correos electrónicos a través del protocolo POP3.

```
import poplib

def recibir_correo():
    pop_conn = poplib.POP3('pop.example.com')
    pop_conn.user('usuario')
    pop_conn.pass_('contraseña')
    numMessages = len(pop_conn.list()[1])
    for i in range(numMessages):
        for msg in pop_conn.retr(i+1)[1]:
            print(msg.decode())
    pop_conn.quit()

# Llama a la función recibir_correo() para leer correos electrónicos
```

4.5.4 El protocolo SMTP

El módulo smtplib se usa para enviar correos electrónicos a través del protocolo SMTP.

4.6 URI y URL

En Python, los módulos urllib y urllib.parse manejan URI y URL.

```
from urllib.parse import urlparse

def analizar_url(url):
```

```
    parsed_url = urlparse(url)
    print(f'Esquema: {parsed_url.scheme}')
    print(f'Host: {parsed_url.netloc}')
    print(f'Ruta: {parsed_url.path}')

# Llama a la función analizar_url() con una URL para descomponerla
```

4.7 Clases estándar de Python para URL

El módulo urllib en Python proporciona clases para manejar URL.

```
from urllib.request import urlopen

def obtener_contenido(url):
    with urlopen(url) as response:
        html = response.read()
        print(html.decode())

# Llama a la función obtener_contenido() con una URL para obtener su contenido
```

4.8 Clases estándar de Python para conexiones con URL de HTTP

4.8.1 Funcionamiento del protocolo HTTP

Python usa http.client para trabajar directamente con el protocolo HTTP.

```
import http.client

def obtener_status_http():
    conn = http.client.HTTPConnection('www.example.com')
    conn.request('HEAD', '/')
    response = conn.getresponse()
    print(f'Código de estado: {response.status}')
    conn.close()
```

```
# Llama a la función obtener_status_http() para obtener el código de estado HTTP
```

4.8.2 Datos adicionales de la respuesta: código de respuesta y cabeceras

Los datos adicionales de una respuesta HTTP se pueden obtener con el módulo http.client.

```
import http.client

def obtener_cabeceras_http():
    conn = http.client.HTTPConnection('www.example.com')
    conn.request('GET', '/')
    response = conn.getresponse()
    print(f'Cabeceras: {response.getheaders()}')
    conn.close()

# Llama a la función obtener_cabeceras_http() para obtener las cabeceras de la respuesta HTTP
```

4.8.3 Envío de datos a servidores de HTTP

Para enviar datos a servidores HTTP, se puede usar el módulo http.client o requests.

```
import requests

def enviar_datos(url, datos):
    response = requests.post(url, data=datos)
    print(f'Respuesta: {response.status_code}')
    print(f'Contenido: {response.text}')

# Llama a la función enviar_datos() con una URL y un diccionario de datos para enviar una solicitud POST
```

5. Seguridad en Comunicaciones y

Criptografía en Python

5.1 Integridad en las Comunicaciones de Datos

En Python, puedes usar bibliotecas como hashlib para verificar la integridad de los datos mediante funciones hash.

```
import hashlib

def calcular_hash_dato(dato):
    sha256 = hashlib.sha256()
    sha256.update(dato.encode())
    return sha256.hexdigest()

# Ejemplo de uso
dato = "Este es un dato importante"
print(f"Hash SHA-256 del dato: {calcular_hash_dato(dato)}")
```

5.2 Criptografía

5.2.1 Criptografía de Clave Privada (o Simétrica)

La criptografía simétrica usa una sola clave para cifrar y descifrar datos. La biblioteca cryptography en Python proporciona esta funcionalidad.

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

def cifrar_dato(dato, clave):
     cipher = Cipher(algorithms.AES(clave), modes.ECB(), backend=default_backend())
    encryptor = cipher.encryptor()
    return encryptor.update(dato) + encryptor.finalize()

def descifrar_dato(dato_cifrado, clave):
```

```
    cipher = Cipher(algorithms.AES(clave), modes.ECB(),
backend=default_backend())
    decryptor = cipher.decryptor()
        return decryptor.update(dato_cifrado) +
decryptor.finalize()

clave = b'0123456789abcdef'  # Clave de 16 bytes para AES-128
dato = b'1234567890123456'  # Dato de 16 bytes

dato_cifrado = cifrar_dato(dato, clave)
print(f"Dato cifrado: {dato_cifrado}")
print(f"Dato descifrado: {descifrar_dato(dato_cifrado,
clave)}")
```

5.2.2 Criptografía de Clave Pública (o Asimétrica)

La criptografía asimétrica utiliza un par de claves, una pública y una privada. La biblioteca cryptography también soporta criptografía asimétrica.

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

# Generar par de claves RSA
private_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048)
public_key = private_key.public_key()

# Serializar claves
pem =
private_key.private_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PrivateFormat.TraditionalOpenSSL,
encryption_algorithm=serialization.NoEncryption())
public_pem =
public_key.public_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo)

# Cifrar y descifrar datos
mensaje = b'Este es un mensaje secreto'
```

```
ciphertext = public_key.encrypt(mensaje, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
plaintext = private_key.decrypt(ciphertext, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))

print(f"Mensaje cifrado: {ciphertext}")
print(f"Mensaje descifrado: {plaintext}")
```

5.3 Firma Digital

Las firmas digitales garantizan la autenticidad e integridad de los mensajes. La biblioteca cryptography también se usa para firmar digitalmente datos.

```
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

def firmar_dato(dato, clave_privada):
    signature = clave_privada.sign(dato, padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
    return signature

def verificar_firma(dato, firma, clave_publica):
    try:
        clave_publica.verify(firma, dato, padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
        return True
    except Exception as e:
        return False

# Generar claves para firma digital
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
public_key = private_key.public_key()

# Firmar y verificar mensaje
```

```
mensaje = b'Mensaje importante'
firma = firmar_dato(mensaje, private_key)
print(f"Firma: {firma}")
print(f"Firma válida: {verificar_firma(mensaje, firma,
public_key)}")
```

5.4 Certificados Digitales

5.4.1 Estructura de un Certificado Digital X.509

En Python, puedes usar la biblioteca cryptography para manejar certificados X.509.

```
from cryptography import x509
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

# Generar clave y certificado de ejemplo
private_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048)
builder = x509.CertificateBuilder()
certificate =
builder.subject_name(x509.Name([x509.NameAttribute(x509.NameOI
D.COMMON_NAME, u'example.com')])) \
.issuer_name(x509.Name([x509.NameAttribute(x509.NameOID.COMMON
_NAME, u'example.com')])) \
                    .public_key(private_key.public_key()) \
.serial_number(x509.random_serial_number()) \
                    .not_valid_before(datetime.utcnow()) \
                      .not_valid_after(datetime.utcnow() +
timedelta(days=10)) \
                    .sign(private_key, hashes.SHA256())

# Serializar el certificado
cert_pem =
certificate.public_bytes(encoding=serialization.Encoding.PEM)
print(cert_pem.decode())
```

5.4.2 Tipos de Ficheros para Certificados Digitales

Los certificados digitales pueden estar en formatos como PEM, DER y PFX. La biblioteca cryptography maneja estos formatos.

```
# Los certificados se pueden guardar en diferentes formatos
from cryptography.hazmat.primitives import serialization

# Ejemplo de serialización de clave privada en formato PEM
pem = private_key.private_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PrivateFormat.TraditionalOpenSSL,
encryption_algorithm=serialization.NoEncryption())
print(pem.decode())
```

5.4.3 Infraestructura de Clave Pública

La infraestructura de clave pública (PKI) se utiliza para gestionar certificados digitales y claves públicas.

5.5 TLS/SSL

5.5.1 Funcionamiento de TLS/SSL

Para trabajar con TLS/SSL en Python, puedes usar el módulo ssl para envolver conexiones encriptadas.

```
import ssl
import socket

def crear_conexion_ssl():
    context = ssl.create_default_context()
    with socket.create_connection(('www.example.com', 443)) as
sock:
                    with  context.wrap_socket(sock,
server_hostname='www.example.com') as ssock:
          print(ssock.version())

# Llama a la función para crear una conexión SSL
```

```
crear_conexion_ssl()
```

5.5.2 Túneles con SSH

Para crear túneles SSH, puedes usar la biblioteca paramiko.

```
import paramiko

def crear_tunel_ssh():
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
       ssh.connect('ssh.example.com',  username='usuario',
password='contraseña')
     stdin, stdout, stderr = ssh.exec_command('echo "Hello,
SSH!"')
    print(stdout.read().decode())
    ssh.close()

# Llama a la función para crear un túnel SSH
crear_tunel_ssh()
```

5.6 Seguridad en Python

La seguridad en Python se aborda mediante el uso de bibliotecas para la criptografía, manejo seguro de datos y conexiones.

5.7 Criptografía con Python

5.7.1 Resúmenes de Mensajes (Funciones de Hash o Digest)

Python utiliza el módulo hashlib para generar resúmenes de mensajes con diversas funciones hash.

```
import hashlib

def generar_hash(dato):
    hash_obj = hashlib.sha256()
```

```
    hash_obj.update(dato.encode())
    return hash_obj.hexdigest()

# Ejemplo de uso
dato = "Mensaje para hash"
print(f"Hash SHA-256 del dato: {generar_hash(dato)}")
```

5.7.2 Generación y Gestión de Claves

La biblioteca cryptography permite generar y gestionar claves criptográficas.

```
from cryptography.hazmat.primitives.asymmetric import rsa

def generar_clave():
                                        private_key         =
rsa.generate_private_key(public_exponent=65537, key_size=2048)
    return private_key

# Generar clave
clave_privada = generar_clave()
print(clave_privada)
```

5.7.3 Criptografía de Clave Privada

La criptografía de clave privada se usa para cifrar y descifrar datos usando una clave secreta compartida.

5.8 Generación y Uso de Certificados Digitales

5.8.1 Generación de un Certificado Digital Autofirmado

Para generar un certificado digital autofirmado en Python, usa la biblioteca cryptography.

```
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
```

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import serialization
from cryptography import x509
from datetime import datetime, timedelta

def generar_certificado():
                                        private_key         =
rsa.generate_private_key(public_exponent=65537, key_size=2048)
    builder = x509.CertificateBuilder()
                                        certificate         =
builder.subject_name(x509.Name([x509.NameAttribute(x509.NameOI
D.COMMON_NAME, u'example.com')])) \
.issuer_name(x509.Name([x509.NameAttribute(x509.NameOID.COMMON
_NAME, u'example.com')])) \
                         .public_key(private_key.public_key())
\
.serial_number(x509.random_serial_number()) \
                        .not_valid_before(datetime.utcnow()) \
                         .not_valid_after(datetime.utcnow() +
timedelta(days=365)) \
                        .sign(private_key, hashes.SHA256())
    return certificate

# Generar certificado
certificado = generar_certificado()
print(certificado.public_bytes(serialization.Encoding.PEM).dec
ode())
```

5.8.2 Obtención y Uso de la Clave Pública de un Certificado

Para extraer y usar la clave pública de un certificado digital en Python, se puede usar la biblioteca cryptography.

```
from cryptography import x509
from cryptography.hazmat.primitives import serialization

def extraer_clave_publica(certificado_pem):
                                        certificado         =
x509.load_pem_x509_certificate(certificado_pem)
```

```
    clave_publica = certificado.public_key()
    return clave_publica

# Usar clave pública
certificado_pem = certificado.public_bytes(serialization.Encoding.PEM)
clave_publica = extraer_clave_publica(certificado_pem)
print(clave_publica)
```

5.9 Generación y Uso de Firma Digital

Las firmas digitales garantizan la autenticidad e integridad de los datos. La biblioteca cryptography facilita la firma y verificación de datos.

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes

def firmar_dato(dato, clave_privada):
    firma = clave_privada.sign(dato, padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
    return firma

def verificar_firma(dato, firma, clave_publica):
    try:
        clave_publica.verify(firma, dato, padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
        return True
    except Exception:
        return False

# Generar claves para firma digital
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
public_key = private_key.public_key()

# Firmar y verificar mensaje
```

```
mensaje = b'Mensaje importante'
firma = firmar_dato(mensaje, private_key)
print(f"Firma: {firma}")
print(f"Firma válida: {verificar_firma(mensaje, firma,
public_key)}")
```