Obiettivo del progetto: Progettare un semplice server HTTP in Python (usando socket) e servire un sito web statico con HTML/CSS.

Di seguito si analizzerà nel dettaglio il codice Python del file socket_server, che costituisce il server vero e proprio, e successivamente customhandler, una versione modificata di SimpleHTTPRequestHandler che presenta funzionalità specifiche rivolte al riconoscimento dei MIME types e al logging delle richieste.

Socket_server

Inizialmente se l'utente specifica una porta come argomento da riga di comando viene utilizzata quella, altrimenti la porta su cui il server si metterà in ascolto è la porta 8080.

```
if sys.argv[1:]:
  port = int(sys.argv[1])
else:
  port = 8080
```

L'oggetto server vero e proprio viene creato con l'istruzione socket.ThreadingTCPServer(), utilizzando la porta stabilita precedentemente e CustomHandler che verrà approfondito in seguito. Il server accetta connessioni da qualsiasi indirizzo IP sulla porta indicata.

Con l'istruzione os.chdir("www") il server viene "obbligato" a recuperare i file del sito web statico dalla cartella www che si trova sullo stesso percorso del server.

```
os.chdir("www")
server = socketserver.ThreadingTCPServer(('',port), CustomHandler)
```

La configurazione del server avviene con le due istruzioni immediatamente successive dove i thread del server vengono eseguiti come daemon, quindi terminano automaticamente quando il server si chiude, e viene permesso anche di riutilizzare la stessa porta anche se il socket precedente non è ancora stato rilasciato. Senza questa ultima impostazione potrebbe essere necessario aspettare un certo intervallo di tempo, tipicamente qualche secondo, prima di poter riavviare il server sulla stessa porta.

```
server.daemon_threads = True

server.allow_reuse_address = True
```

In seguito viene definita una semplice funzione (signal_handler) che viene utilizzata per chiudere in modo sicuro il socket del server e terminare il processo se l'utente preme da tastiera la combinazione di tasi Ctrl+C.

```
def signal_handler(signal, frame):
    print( 'Exiting http server (Ctrl+C pressed)')
    try:
      if( server ):
        server.server_close()
    finally:
      sys.exit(0)


signal.signal(signal.SIGINT, signal_handler)
```

Infine il server entra in un loop infinito, accettando le richieste http e gestendole con CustomHandler, fino a quando l'utente non interrompe il socket del server con il comando Ctrl+C.

```
try:
  while True:
    server.serve_forever()
except KeyboardInterrupt:
  pass

server.server_close()
```

CustomHandler

CustomHandler presenta degli "Override" di alcuni metodi di SimpleHTTPRequestHandler che permettono di gestire con maggior dettaglio alcuni elementi specifici della gestione delle richieste delle pagine per soddisfare le richieste facoltative della traccia del progetto.

Inizialmente la funzione guess_type() determina il MIME type del file richiesto (HTML, CSS, immagini…). Se ha un'estensione diversa da tutti i casi specificati richiama super.guess_type() in modo da poter gestire il MIME types in maniera predefinita.

```
  def guess_type(self, path):
      if path.endswith(".html"):
          return "text/html"
      elif path.endswith(".css"):
          return "text/css"
      elif path.endswith(".jpg"):
          return "image/jpeg"
      elif path.endswith(".png"):
          return "image/png"
      else:
          return super().guess_type(path)
```

Anche la funzione do_GET() viene approfondita, per evitare un errore del tipo 404 Not Found quando la richiesta riguarda favicon.ico, rispondendo invece con 204 no content. Altrimenti la richiesta viene gestita nel modo normale con super.do_GET() e registra un messaggio di log.

```
def do_GET(self):
    if self.path == "/favicon.ico":
        self.send_response(204)
        self.end_headers()
        return
    else:
        super().do_GET()
        self.log_message("GET %s - 200 OK", self.path)
```

Infine viene definita una nuova funzione: log_message(), la quale serve per registrare ogni richiesta http in un file server.log. Per ogni richiesta viene generato un timestamp che mostra la data e l'orario di quando è avvenuta la richiesta e l'indirizzo IP del client in modo da poter stabilire chiaramente quando e da chi viene richiesta la visione della pagine del sito web statico. Con l'ultima parte della funzione ci si assicura che se il file server.log non esiste ancora venga creato e che si trovi esattamente nella stessa cartella dove si trova anche il codice python.

L'ultima istruzione, ossia log_file.flush() forza il salvataggio immediato dei dati su disco, in modo che in seguito a errori imprevisti come per esempio il crash del programma non vengano persi gli ultimi dati scritti sul file.

```
def log_message(self, format, *args):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_entry = f"[{timestamp}] {self.address_string()} - {format % args}\n"


    code_dir = os.path.dirname(os.path.abspath(__file__))
    log_path = os.path.join(code_dir, "server.log")
    with open(log_path, "a+", encoding="utf-8") as log_file:
        log_file.write(log_entry)
        log_file.flush()
```