REPORT S2/L5

Comprensione e analisi di un codice PYTHON

Traccia

Per agire come un Hacker bisogna capire come pensare fuori dagli schemi. L'esercizio di oggi ha lo scopo di allenare l'osservazione critica. Dato il codice si richiede allo studente di:

- Capire cosa fa il programma senza eseguirlo.
- Individuare nel codice sorgente le casistiche non standard che il programma non gestisce (esempio, comportamenti potenziali che non sono stati contemplati).
- Individuare eventuali errori di sintassi / logici.
- Proporre una soluzione per ognuno di essi.

Di seguito il codice proposto

```
import datetime
def assistente_virtuale(comando):
        if comando == "Qual è la data di oggi?":
                oggi = datetime.datetoday()
                risposta = "La data di oggi è " + oggi.strftime("%d/%m/%Y")
        elif comando == "Che ore sono?":
                ora_attuale = datetime.datetime.now().time()
                risposta = "L'ora attuale è " + ora_attuale.strftime("%H:%M")
        elif comando == "Come ti chiami?":
                risposta = "Mi chiamo Assistente Virtuale"
        else: risposta = "Non ho capito la tua domanda."
        return risposta
while True
        comando_utente = input("Cosa vuoi sapere? ")
                if comando_utente.lower() == "esci":
                        print("Arrivederci!")
                        break
                else:
                        print(assistente_virtuale(comando_utente)
```

1. Introduzione

In questo report analizzeremo un codice python che simula un assistente virtuale. Questo tipo di applicazione è un tipico esempio di interazione uomo-macchina, dove l'utente invia comandi e il programma restituisce risposte predeterminate.

L'obiettivo dell'esercizio è osservare e analizzare il codice, individuando eventuali errori di sintassi, di logica o di comportamento non previsto, e quindi proporre una soluzione per ognuno.

2. Descrizione del codice originale

Il programma in questione prevede di simulare un assistente virtuale che risponde a domande specifiche. Quelle riconosciute dal sistema sono:

1. "Qual è la data di oggi?"
2. "Che ore sono?"
3. "Come ti chiami?"

Se l'utente fornisce uno di questi comandi, il sistema risponde con la data odierna, l'ora corrente o il nome dell'assistente stesso. Se il comando non è tra quelli riconosciuti, la risposta preimpostata sarà: "Non ho capito la tua domanda."

Il programma è progettato per funzionare in un ciclo infinito, ricevendo input dall'utente fino a quando l'utente non digita "esci", momento in cui il programma si ferma.

3. Analisi critica del codice

3.1. Struttura e flusso del programma

Il programma presenta un flusso di esecuzione relativamente semplice:

1. Il programma accetta un comando dall'utente.
2. In base al comando, il programma fornisce una risposta.
3. Se il comando non è riconosciuto, restituisce un messaggio di errore.
4. Se l'utente digita "esci", il programma termina.

Tuttavia, sono presenti alcuni problemi di sintassi e logica che potrebbero influire sul corretto funzionamento del programma. Di seguito, descriviamo i problemi individuati nel codice.

3.2. Errori di sintassi e logica

3.2.1. Mancanza dei due punti in while True

Una delle prime problematiche evidenti è che nel ciclo `while True` dove si nota l'assenza del simbolo dei due punti (`:`). Questo è un errore di sintassi in python, poiché i cicli e le strutture condizionali devono terminare con i due punti.

Soluzione proposta: Aggiungere i due punti alla fine della dichiarazione del ciclo:

```
while True:
```

3.2.2. Errore nella chiamata del metodo datetoday()

Nel codice originale, viene invocato un metodo non esistente: `datetime.datetoday()` e questo genererà un errore in fase di esecuzione. Il metodo corretto per ottenere la data odierna in python è `datetime.date.today()`.

Soluzione proposta: Correggere la chiamata al metodo in:

```
oggi = datetime.date.today()
```

3.2.3. Uso di datetime.datetime.now().time()

Il metodo `datetime.datetime.now().time()` restituisce solo l'orario (senza la data). Sebbene ciò non sia un errore di per sé, la lettura del codice potrebbe essere migliorata per ottenere sia la data che l'ora in una forma più completa e comprensibile.

Soluzione proposta: Per una maggiore chiarezza si può utilizzare direttamente:

```
ora_attuale = datetime.datetime.now().strftime("%H:%M")
```

In questo modo, l'orario completo verrà formattato direttamente come stringa.

3.2.4. Risposta poco chiara per comandi non riconosciuti

Il programma risponde con "Non ho capito la tua domanda." se il comando non è riconosciuto. Questo messaggio potrebbe essere più specifico come, ad esempio dicendo "Comando non riconosciuto."

Soluzione proposta: Modificare il messaggio di errore in:

```
risposta = "Comando non riconosciuto."
```

3.2.5. Potenziale mancanza di gestione degli errori

Il programma non prevede una gestione adeguata degli errori, che potrebbe risultare problematica quando, ad esempio, l'utente inserisce un comando con caratteri speciali o un input non valido, dovuto ad un errore di battitura.

Soluzione proposta: Aggiungere una gestione degli errori per prevenire possibili interruzioni del programma tramite un try-except. Di seguito riportiamo un esempio di gestione degli errori generici:

```
try:
    comando_utente = input("Cosa vuoi sapere? ")
    if comando_utente.lower() == "esci":
        print("Arrivederci!")
        break
    else:
        print(assistente_virtuale(comando_utente))
except Exception as e:
    print(f"Si è verificato un errore: {e}")
```

4. Modifiche e miglioramenti al codice

4.1. Ottimizzazione della leggibilità del codice

Per migliorare la leggibilità, è consigliato usare le f-string per formattare le risposte. Le f-string sono più chiare ed efficienti rispetto alla concatenazione delle stringhe.

Codice originale:

```
risposta = "La data di oggi è " + oggi.strftime("%d/%m/%Y")
```

Codice migliorato:

```
risposta = f"La data di oggi è {oggi.strftime('%d/%m/%Y')}"
```

4.2. Aggiunta di una gestione delle risposte non riconosciute

Invece di limitarsi a rispondere genericamente con "Non ho capito la tua domanda.", sarebbe utile dare all'utente un'indicazione più precisa sul fatto che il comando non è stato riconosciuto.

Soluzione proposta:

```
risposta = "Comando non riconosciuto. Per favore, prova con una
domanda valida."
```

4.3. Aggiunta di un feedback per l'utente

Potrebbe essere utile includere un messaggio che inviti l'utente a fornire uno dei comandi riconosciuti. Questo può migliorare l'interazione e prevenire confusione.

Soluzione proposta: Aggiungere un messaggio di aiuto all'inizio del programma o quando il comando non è riconosciuto:

```
print("Ciao! Puoi chiedermi: 'Qual è la data di oggi?', 'Che ore sono?' o 'Come ti chiami?'")
```

5. Codice corretto e migliorato

Dopo aver analizzato e corretto gli errori, il codice risultante è il seguente:

```
import datetime

def assistente_virtuale(comando):
    if comando == "Qual è la data di oggi?":
        oggi = datetime.date.today()  # Correzione del metodo per ottenere la data
        risposta = f"La data di oggi è {oggi.strftime('%d/%m/%Y')}"  # Uso delle f-string
    elif comando == "Che ore sono?":
        ora_attuale = datetime.datetime.now().strftime("%H:%M")  # Miglioramento nell'estrazione dell'ora
        risposta = f"L'ora attuale è {ora_attuale}"  # Uso delle f-string
    elif comando == "Come ti chiami?":
        risposta = "Mi chiamo Assistente Virtuale"
    else:
        risposta = "Comando non riconosciuto. Per favore, prova con una domanda valida." # Miglioramento nella risposta di errore

    return risposta

while True:
    try: #Gestione delle eccezioni
        comando_utente = input("Cosa vuoi sapere? ")
        if comando_utente.lower() == "esci":
            print("Arrivederci!")
            break
        else:
            print(assistente_virtuale(comando_utente))
    except Exception as e:
        print(f"Si è verificato un errore: {e}")
```

6. Conclusioni e riflessioni finali

In questo esercizio, abbiamo analizzato un programma python che simula un assistente virtuale. Abbiamo identificato vari errori di sintassi e logica, come la mancanza dei due punti nel ciclo `while`, un errore nel metodo `datetoday()`, e l'uso di una risposta poco chiara per i comandi non riconosciuti. Inoltre, sono stati proposti miglioramenti nella leggibilità e nella gestione degli errori.

L'implementazione finale fornisce un programma più robusto, leggibile e user-friendly. La gestione degli errori e la formattazione migliorata delle risposte renderanno l'assistente virtuale più affidabile e facile da usare.

L'esercizio ha dimostrato quindi quanto sia importante un'analisi critica del codice. Anche in un programma relativamente semplice, è fondamentale osservare ogni parte del codice per scoprire potenziali errori e punti deboli. Ogni piccolo miglioramento può fare la differenza in termini di affidabilità, usabilità e leggibilità del software.

REPORT S2/L5 bonus

Creazione di un programma in PYTHON

Traccia

Scrivi un programma Python per gestire una lista della spesa. Il programma deve permettere all'utente di:

1. Aggiungere un elemento alla lista.
2. Rimuovere un elemento dalla lista (se presente).
3. Visualizzare tutti gli elementi della lista ordinati in ordine alfabetico.
4. Salvare la lista su file.
5. Caricare una lista da file.

Il programma deve avere un menu che consente all'utente di scegliere le varie operazioni e deve terminare solo quando l'utente lo richiede.

1. Introduzione al programma

Il programma sviluppato serve a gestire una lista della spesa. L'utente può aggiungere, rimuovere, visualizzare, salvare e caricare la lista da un file. Le operazioni sono eseguite tramite un menu interattivo, e l'utente può eseguire queste operazioni ripetutamente fino a quando non decide di uscire.

2. Funzionalità principali

Il programma consente di eseguire le seguenti operazioni:

- **Aggiungere un elemento alla lista**: l'utente inserisce un elemento che viene aggiunto alla lista.
- **Rimuovere un elemento dalla lista**: l'utente scrive quale elemento verrà rimosso, se presente nella lista.
- **Visualizzare la lista ordinata**: la lista della spesa viene mostrata in ordine alfabetico.
- **Salvare la lista su file**: la lista viene salvata in un file (se il file esiste, viene sovrascritto).
- **Caricare la lista da file**: la lista viene caricata da un file, se esiste.

3. Scrittura del codice

Di seguito riportiamo il codice che è stato scritto per la risoluzione della traccia:

```
import os  # Importa il modulo os per interagire con il sistema operativo (necessario per gestire i file)

print("PROGRAMMA PER LA GESTIONE DELLA LISTA DELLA SPESA")

lista_della_spesa = []  # Inizializza una lista vuota che conterrà gli elementi della spesa
filename = "lista_spesa.txt"  # Nome del file su cui verranno salvati e da cui verranno caricati gli elementi della lista

while True:  # Ciclo principale che continua a mostrare il menu finché l'utente non sceglie di uscire
    # Mostra il menu delle opzioni disponibili
    print("\nMenu:")
    print("1) Aggiungi un elemento alla lista")
    print("2) Rimuovi un elemento dalla lista")
    print("3) Visualizza la lista ordinata")
    print("4) Salva la lista su file")
    print("5) Carica la lista da file")
    print("6) Esci\n")

    # Chiede all'utente di scegliere un'operazione
    scelta = input("Scegli un'operazione (1-6): ")

    # Usa il costrutto match-case per eseguire l'operazione corrispondente alla scelta dell'utente
    match scelta:
        case '1':  # Se l'utente sceglie '1', aggiungi un elemento alla lista
            elemento = input("Inserisci l'elemento da aggiungere: ")  # Chiede all'utente di inserire un elemento
            lista_della_spesa.append(elemento)  # Aggiunge l'elemento alla lista
            print(f"Elemento '{elemento}' aggiunto alla lista.")  # Conferma l'aggiunta

        case '2':  # Se l'utente sceglie '2', rimuovi un elemento dalla lista
            if lista_della_spesa:  # Verifica se la lista non è vuota
                # Mostra la lista ordinata prima di rimuovere un elemento
                print("\nLista attuale:")
                lista_della_spesa.sort()  # Ordina la lista alfabeticamente
                for item in lista_della_spesa:
                    print(f"- {item}")  # Stampa ogni elemento della lista

                # Chiede all'utente di inserire l'elemento da rimuovere
                elemento = input("Inserisci l'elemento da rimuovere: ")

                # Controlla se l'elemento esiste nella lista
```

```
                if elemento in lista_della_spesa:
                    lista_della_spesa.remove(elemento)  # Rimuove l'elemento dalla lista
                    print(f"Elemento '{elemento}' rimosso dalla lista.")  # Conferma la rimozione
                else:
                    print(f"L'elemento '{elemento}' non è presente nella lista.")  # Messaggio se l'elemento non esiste
            else:
                print("La lista è vuota. Nessun elemento da rimuovere.")  # Messaggio se la lista è vuota

        case '3':  # Se l'utente sceglie '3', visualizza la lista ordinata
            if lista_della_spesa:  # Verifica se la lista non è vuota
                lista_della_spesa.sort()  # Ordina la lista alfabeticamente
                print("\nLista della spesa ordinata:")
                for item in lista_della_spesa:
                    print(f"- {item}")  # Stampa ogni elemento della lista ordinata
            else:
                print("La lista è vuota.")  # Messaggio se la lista è vuota

        case '4':  # Se l'utente sceglie '4', salva la lista su file
            if os.path.exists(filename):  # Verifica se il file esiste già
                with open(filename, 'w') as file:  # Apre il file in modalità scrittura ('w'), sovrascrivendo il contenuto
                    for item in lista_della_spesa:  # Per ogni elemento nella lista
                        file.write(item + '\n')  # Scrive l'elemento nel file, separandolo da una nuova riga
                print(f"Lista salvata su {filename}. Il file esisteva già ed è stato sovrascritto.")  # Conferma che il file è stato sovrascritto
            else:
                with open(filename, 'w') as file:  # Se il file non esiste, lo crea e apre in modalità scrittura
                    for item in lista_della_spesa:  # Per ogni elemento nella lista
                        file.write(item + '\n')  # Scrive l'elemento nel file
                print(f"Lista salvata su {filename}. Il file è stato creato.")  # Conferma che il file è stato creato

        case '5':  # Se l'utente sceglie '5', carica la lista da un file
            if os.path.exists(filename):  # Verifica se il file esiste
                with open(filename, 'r') as file:  # Apre il file in modalità lettura ('r')
                    lista_della_spesa = [line.strip() for line in file.readlines()]  # Legge le righe del file, rimuovendo eventuali spazi bianchi
                print(f"Lista caricata da {filename}.")  # Conferma che la lista è stata caricata
            else:
                print(f"Il file {filename} non esiste.")  # Messaggio se il file non esiste

        case '6':  # Se l'utente sceglie '6', esce dal programma
```

```
            print("Uscita dal programma.")  # Messaggio di uscita
            break  # Esce dal ciclo e termina il programma

        case _:  # Se l'utente inserisce una scelta non valida
            print("Scelta non valida, riprova.")  # Messaggio che avvisa l'utente della scelta non valida
```

4. Funzionamento del programma

Il programma è strutturato come un ciclo che offre continuamente il menu delle opzioni finché l'utente non sceglie di uscire. La gestione delle operazioni avviene attraverso la selezione dell'utente, con input corrispondente al numero dell'elenco di ogni operazione. Ogni operazione è stata poi sviluppata all'interno di un ciclo `match-case`, che ci permette di gestire diverse casistiche e a differenza di un `if-elif`, analogo nel funzionamento, il codice risulta più organizzato e leggibile.

A seguire inseriamo alcuni screen che dimostrano la funzionalità e l'utilizzabilità del programma

Figura 1. Menu di base

Figura 2. Inserimento di 3 elementi

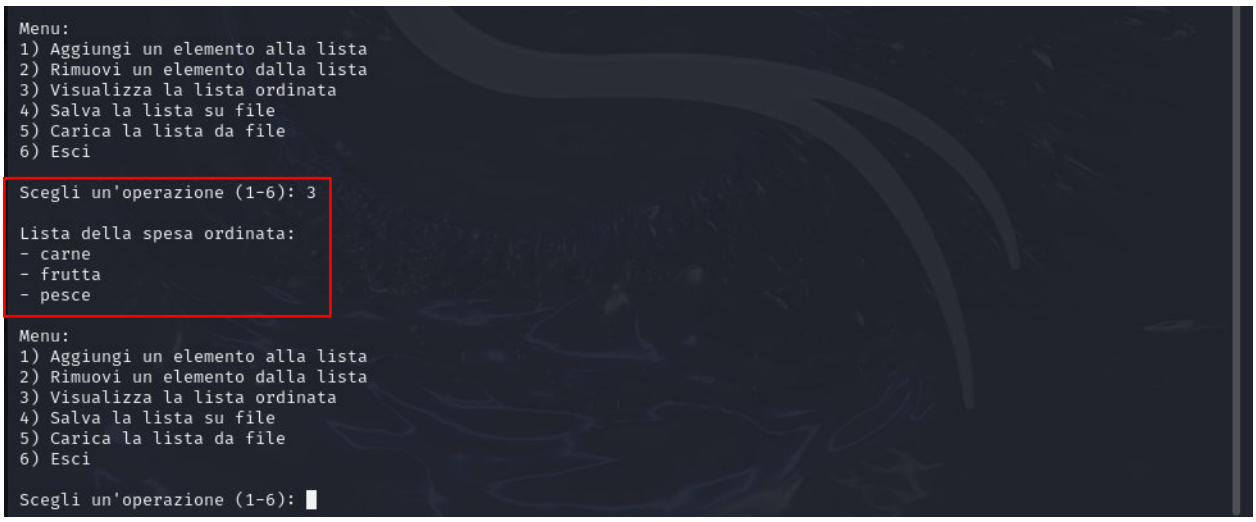

Figura 3. Visualizzazione ordinata della lista

Figura 4. Eliminazione di un elemento dalla lista

Figura 5. Salvataggio della lista su file

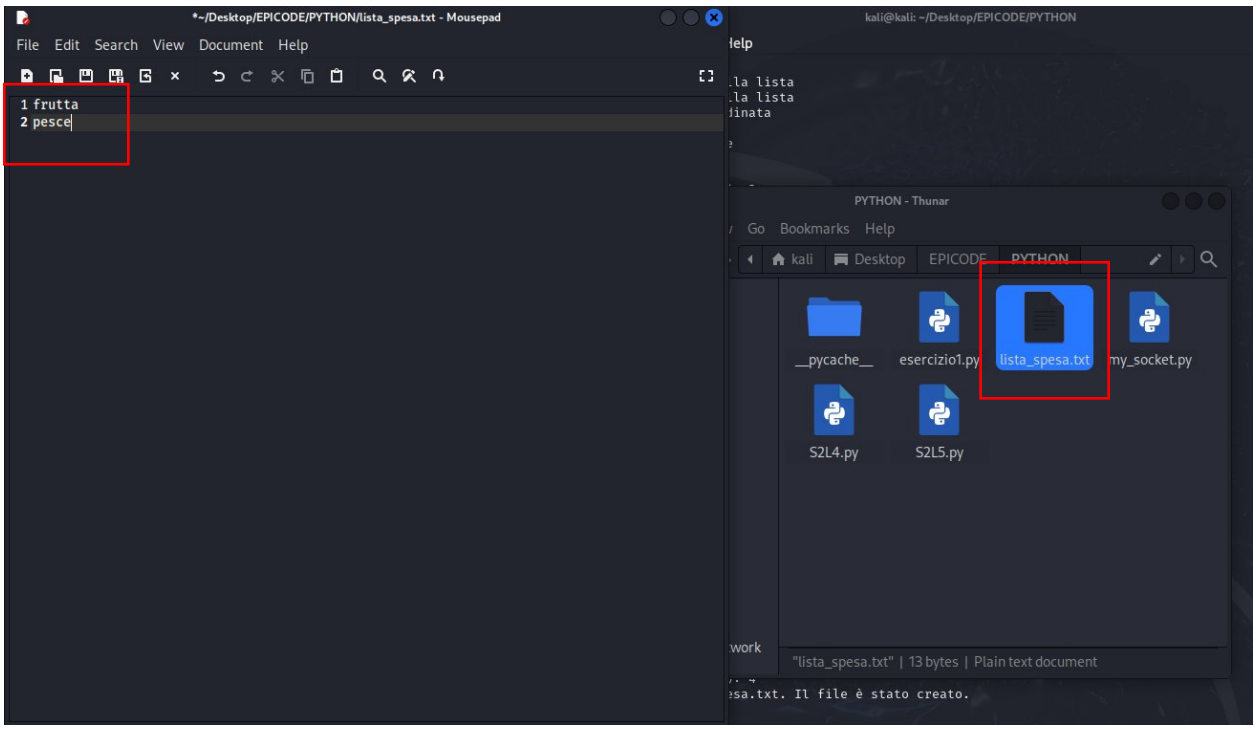

Figura 6. File salvato nella stessa directory del programma

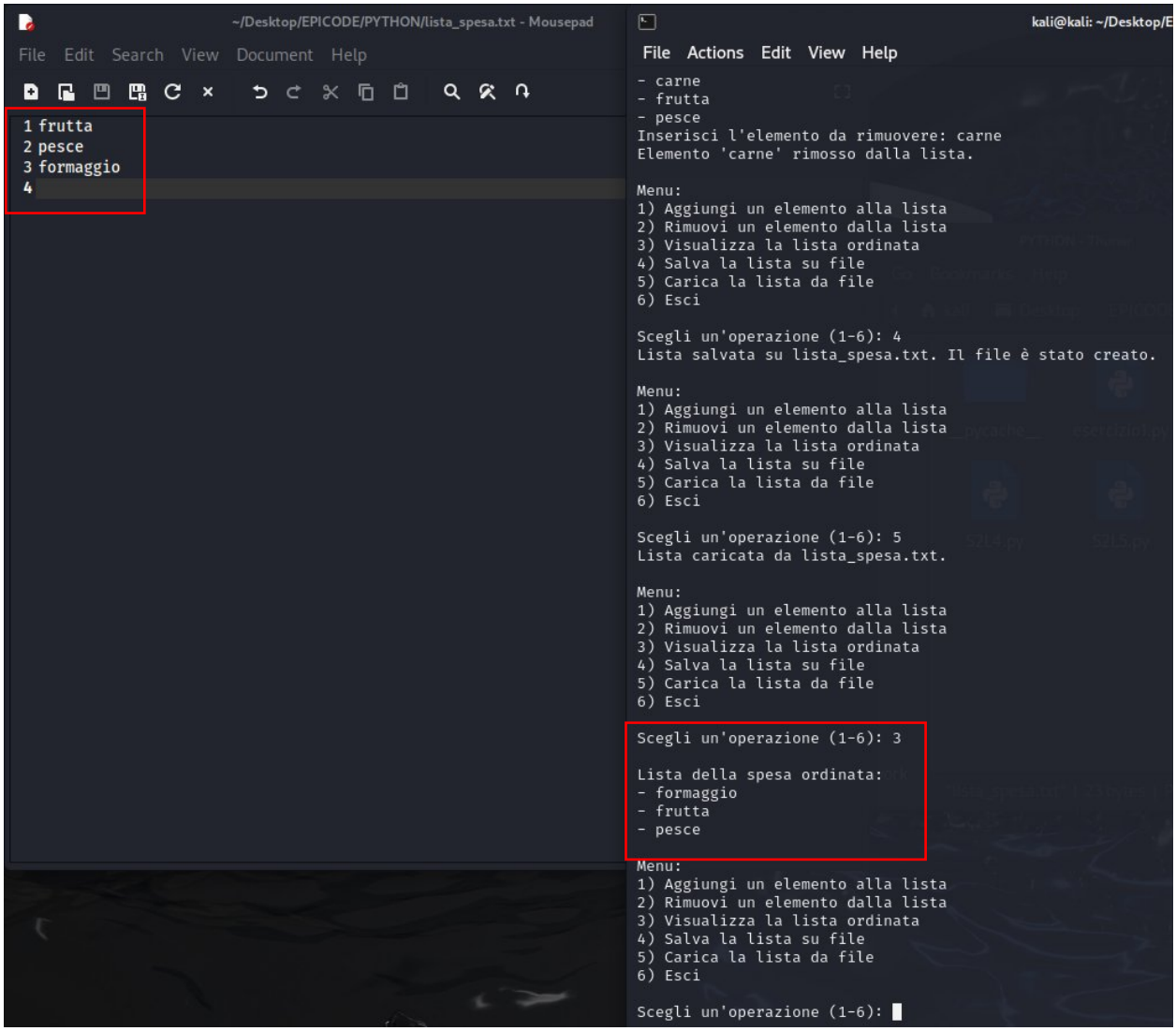

Figura 7. Inserimento di un nuovo elemento nel file e caricamento della nuova lista

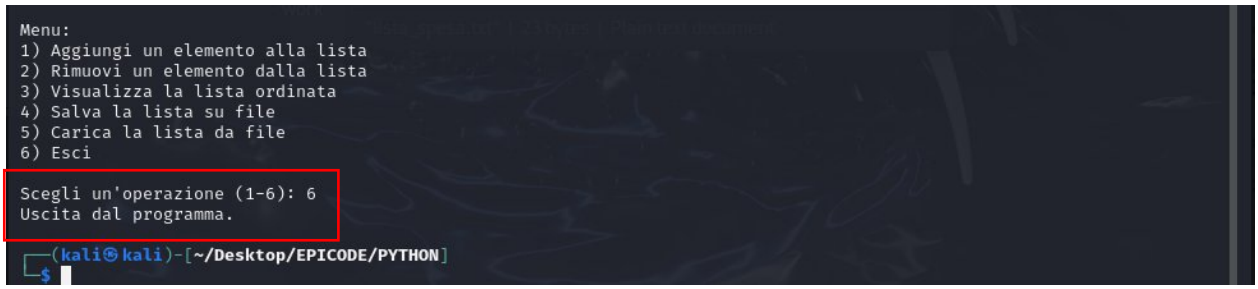

Figura 8. Uscita dal programma

5. Casistiche e comportamenti non gestiti

Nonostante il programma sia funzionale, siamo consapevoli che non tutte le casistiche sono state gestite o correttamente implementate. Pertanto in questo paragrafo andiamo ad analizzare e spiegare cosa può ancora essere migliorato. Quello che segue è un breve elenco di alcune situazioni particolari o casi di errore che potrebbero verificarsi:

- **Gestione degli errori durante l'input dell'utente**:
  - Se l'utente inserisce un dato non valido (ad esempio, un numero al posto di un testo), il programma non gestisce correttamente l'errore, causando un comportamento non previsto.
  - Non c'è una validazione per evitare spazi vuoti o input speciali che potrebbero essere considerati come elementi validi.
- **Gestione dei file**:
  - Se il file di salvataggio è aperto in un altro programma, il programma potrebbe fallire nel tentativo di scrittura o lettura senza avvisare l'utente in modo chiaro.
  - La funzione di salvataggio sovrascrive sempre il file, senza chiedere conferma all'utente prima di farlo.
- **Manipolazione della lista**:
  - Se l'utente prova a rimuovere un elemento che non esiste nella lista, il programma non offre una vera opzione per correggere l'errore e richiede un altro tentativo senza permettere un miglior flusso di operazioni.
- **Caricamento di file con formati non corretti**:
  - Il programma non gestisce errori nei file di input che non rispettano il formato previsto. Se il file non è in formato semplice (un elemento per riga), il programma potrebbe fallire nel leggere i dati.
- **Caricamento di file con filename diverso**:
  - Il programma non gestisce il caricamento di liste tramite file di input che non abbiano il nome previsto.

A seguito delle criticità sopra elencate possiamo individuare alcuni miglioramenti ancora applicabili al codice scritto:

- **Gestione degli errori durante l'input**: sarebbe utile aggiungere dei controlli per verificare che l'input dell'utente sia valido. Per esempio, per evitare che l'utente inserisca solo spazi vuoti o caratteri speciali, si potrebbe utilizzare un controllo che valida l'input.
- **Conferma prima di sovrascrivere il file**: aggiungere una conferma prima di sovrascrivere un file esistente, in modo da evitare la perdita accidentale di dati.
- **Gestione migliorata dei file**: sarebbe utile aggiungere un controllo di errore che avvisi l'utente se il file è già aperto in un altro programma o se ci sono problemi di permessi nel leggere/scrivere il file.
- **Gestione dell'elenco**: modificare il modo in cui vengono visualizzati gli elementi della lista, non con un semplice elenco puntato, ma tramite elenco numerato. Questo facilita l'individuazione di un singolo elemento sia da parte dell'utente che da parte del programma stesso.
- **Funzione di ricerca nell'elenco**: aggiungere una funzione che consenta all'utente di cercare un elemento nella lista per nome. Questo migliorerebbe l'interazione e la navigabilità della lista.
- **Gestione dell'eliminazione nell'elenco**: aggiungere la possibilità di scegliere l'elemento della lista da eliminare non solo tramite il reinserimento completo del nome, ma anche tramite la selezione del numero di elenco.
- **Supporto per file con formato diverso**: il programma potrebbe essere migliorato per gestire file con formati più complessi o con più informazioni, come un file CSV o JSON, per consentire operazioni avanzate.
- **Supporto per file con nome diverso**: il programma potrebbe essere migliorato per gestire file con un nome diverso da quello stabilito, facendo scegliere all'utente il file da caricare, dopo aver visualizzato l'elenco di files presenti nella directory da cui si vuole prendere il file stesso.

6. Conclusione

Questo programma è un esempio semplice ma funzionale di come gestire una lista della spesa in modo interattivo e salvare/caricare i dati su file. Sebbene il codice sia realizzato per scenari di base, ci sono alcune aree di miglioramento sia a livello di affidabilità dell'applicazione che di interazione con l'utente finale.