Trabajo Practico Integrador

Click - Creación de interfaces de línea de comandos.

Materia: Programación Avanzada

Docente de cátedra: Lic. Felipe Morales

Grupo 17-Comision 2:

Brandan Jorge Ciro

Tabla de contenido

Introducción al desarrollo de interfaz de línea de comandos en Python

En la programación, una interfaz de línea de comandos (en inglés, Command Line Interface o CLI) es una forma de interactuar con un programa mediante la emisión de comandos a través de una consola o terminal de línea de comandos. Es una alternativa a las interfaces gráficas de usuario (en inglés, Graphical User Interface o GUI) que se utilizan comúnmente en la mayoría de las aplicaciones.

Shell: la base detrás de la CLI

Si nos sumergimos desde CLI en la parte más profunda de un sistema operativo, nos encontraremos con shell.

Shell es una interfaz de usuario responsable de procesar todos los comandos escritos en CLI. Lee e interpreta los comandos e indica al sistema operativo que realice las tareas solicitadas.

En otras palabras, un shell es una interfaz de usuario que administra CLI y actúa como el intermediario, conectando a los usuarios con el sistema operativo.

En la práctica, hay muchas cosas que un shell puede procesar, como:

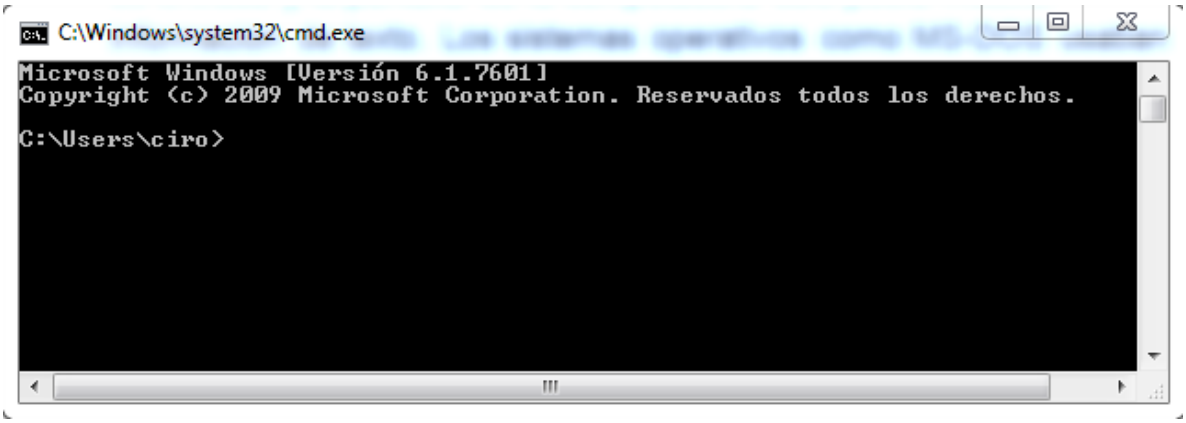
- Trabajar con archivos y directorios
- Abrir y cerrar un programa
- Gestión de procesos informáticos
- Realización de tareas repetitivas

Entre muchos tipos de shell, los más populares son Windows shell (para Windows) y bash (para Linux y MacOS).

Las raíces de CLI

En la década de 1960, CLI se utilizó de forma intensiva.

En ese entonces, las personas solo tenían un teclado como dispositivo de entrada y la pantalla de la computadora solo podía mostrar información de texto. Los sistemas operativos como MS-DOS usaban la CLI como interfaz de usuario estándar.



Básicamente, los usuarios tenían que escribir un comando en la CLI para realizar tareas, ya que esta era la única forma de comunicarse con la computadora.

La invención de un mouse marcó el comienzo del método de apuntar y hacer clic como una nueva forma de interactuar con la computadora.

Además, los sistemas operativos comenzaron a desarrollar una forma atractiva de computación, utilizando GUI (Graphical User Interacción). La propia GUI fue fenomenal debido al uso de botones y menús para representar comandos específicos. Se ha demostrado que este enfoque es muy intuitivo

Principales ventajas de la interacción a través de una CLI en lugar de una GUI

Facilita la creación de scripts más complejos y automatizaciones.

En primer lugar, una CLI puede facilitar la creación de scripts más complejos y automatizaciones. Piensen en una tarea que se deba realizar varias veces, como la creación de usuarios en un sistema. Con una CLI, es fácil para un usuario crear un script que automatice esta tarea, de modo que puede ser ejecutada nuevamente en cualquier momento con solo unos pocos comandos. De hecho, algunas tareas complejas pueden incluso requerir el uso de una CLI debido a la complejidad de las acciones que se deben realizar.

Funciones y operaciones en la CLI son más rápidas que en una GUI.

Funciones y operaciones en la CLI son más rápidas que en una GUI. Por ejemplo, en Linux, muchos administradores del sistema prefieren interactuar con sus sistemas a través de la CLI. Por lo que, trabajando en consola, pueden realizar varias tareas más rápido que utilizando una GUI. La cantidad de tiempo ahorrado es palpable ya que con la CLI no es necesario mover el ratón ni hace falta hacer clic en decenas de menús y submenús para llegar al lugar deseado. Todo se puede hacer desde el teclado, lo que permite que la persona sea más productiva.

Puede simplificar la depuración de problemas y la administración de sistemas más complejos.

La CLI también puede simplificar la depuración de problemas y la administración de sistemas más complejos. Por ejemplo, un problema complejo en un servidor podría ser difícil de diagnosticar usando una GUI compleja, pero con una CLI, es posible ejecutar comandos específicos y recibir resultados precisos y puntuales. Además, con la CLI es más sencillo saber qué está sucediendo con el sistema en el momento, pues la consola le devuelve todo lo que se está ejecutando para su análisis.

Utilizar la librería Click para crear una CLI en Python

Click

Click es una librería que te permite crear CLIs de manera elegante y sencilla en Python. La sintaxis de Click es fácil de aprender y de utilizar, y te permite crear una CLI en cuestión de minutos.

Para utilizar Click, primero debes instalarlo en tu entorno de Python. Puedes hacerlo fácilmente utilizando pip:

```
pip install click
```

Una vez instalado Click, puedes comenzar a crear tu propia CLI utilizando la función cli.command() de Click. Esta función te permite crear nuevos comandos para tu CLI.

Por ejemplo, aquí hay un ejemplo de cómo crear un comando “saludo” que imprime un mensaje de saludo:

```
import click

@click.group()
def cli():
    pass

@cli.command()
@click.option("--nombre", default="Mundo", help="Nombre de la persona a saludar")
def saludo(nombre):
    click.echo(f"Hola, {nombre}!")

if __name__ == "__main__":
    cli()
```

Este código crea una CLI con un único comando, “saludo”. Cuando ejecutas el comando saludo, Click imprimirá un mensaje de saludo con el nombre de la persona a saludar.

Algunas Aplicaciones que se Pueden Desarrollar con Click

La biblioteca Click es extremadamente versátil y se puede utilizar para crear una amplia variedad de aplicaciones de línea de comandos. A continuación, se detallan algunas aplicaciones prácticas que se pueden desarrollar con Click, junto con ejemplos y una breve explicación de cada una.

1. Gestor de Tareas

Un gestor de tareas es una herramienta que permite a los usuarios crear, ver, actualizar y eliminar tareas desde la línea de comandos.

Características:

- Añadir nuevas tareas.

  Permite a los usuarios crear y registrar tareas.
  Utiliza un comando *add* en Click.
  Requiere una descripción de la tarea.

- Listar todas las tareas.

  Permite a los usuarios ver todas las tareas registradas.
  Utiliza un comando *list* en Click.
  Muestra el estado y la descripción de cada tarea.

- Marcar tareas como completadas.

  Permite a los usuarios actualizar el estado de una tarea específica a completada.
  Utiliza un comando *complete* en Click.
  Requiere el ID de la tarea para marcarla como completada.

- Eliminar tareas.

  Permite a los usuarios eliminar una tarea específica.
  Utiliza un comando *delete* en Click.
  Requiere el ID de la tarea para eliminarla.

Ejemplo:

```python
import click

tasks = []

@click.group()
def cli():
    """Un simple gestor de tareas."""
    pass

@click.command()
@click.argument('description')
def add(description):
    """Añadir una nueva tarea con DESCRIPCIÓN."""
    tasks.append({'description': description, 'completed': False})
    click.echo(f'Tarea "{description}" añadida.')

@click.command()
def list():
    """Listar todas las tareas."""
    if not tasks:
        click.echo("No hay tareas.")
        return
    for i, task in enumerate(tasks):
        status = '✓' if task['completed'] else '✗'
        click.echo(f"{i+1}. [{status}] {task['description']}")


@click.command()
@click.argument('task_id', type=int)
def complete(task_id):
    """Marcar la tarea con TASK_ID como completada."""
    try:
        task = tasks[task_id - 1]
        task['completed'] = True
        click.echo(f'Tarea "{task["description"]}" marcada como completada.')
    except IndexError:
        click.echo('ID de tarea no válido.')

@click.command()
@click.argument('task_id', type=int)
def delete(task_id):
    """Eliminar la tarea con TASK_ID."""
    try:
        task = tasks.pop(task_id - 1)
        click.echo(f'Tarea "{task["description"]}" eliminada.')
    except IndexError:
        click.echo('ID de tarea no válido.')

cli.add_command(add)
cli.add_command(list)
cli.add_command(complete)
cli.add_command(delete)

if __name__ == '__main__':
    cli()

```

2. Gestor de Archivos

Un gestor de archivos permite a los usuarios realizar operaciones básicas en archivos y directorios desde la línea de comandos.

Características:

- Crear archivos y directorios.

  Permite a los usuarios crear nuevos archivos y directorios en el sistema.
  Utiliza los comandos *touch* para archivos y *mkdir* para directorios.
  Implementa la creación mediante *open* y *os.makedirs.*

- Listar contenido de directorios.

  Permite a los usuarios ver todos los archivos y subdirectorios dentro de un directorio.
  Utiliza el comando *ls.*
  Implementa el listado mediante *os.listdir..*

- Copiar, mover y eliminar archivos y directorios.

  Copiar: Crea una copia de un archivo o directorio en una nueva ubicación utilizando el comando *cp* y *shutil.copy.*
  Mover: Traslada un archivo o directorio de una ubicación a otra utilizando el comando *mv* y *shutil.move*.
  Eliminar: Elimina un archivo o directorio del sistema utilizando el comando *rm, shutil.rmtree* para directorios y *os.remove* para archivos.

Ejemplo:

```python
import click
import os
import shutil

@click.group()
def cli():
    """Un simple gestor de archivos."""
    pass

@click.command()
@click.argument('filename')
def touch(filename):
    """Crear un nuevo archivo llamado FILENAME."""
    with open(filename, 'w') as f:
        pass
    click.echo(f'Archivo "{filename}" creado.')

@click.command()
@click.argument('dirname')
def mkdir(dirname):
    """Crear un nuevo directorio llamado DIRNAME."""
    os.makedirs(dirname, exist_ok=True)
    click.echo(f'Directorio "{dirname}" creado.')

@click.command()
@click.argument('path')
def ls(path='.'):
    """Listar el contenido de PATH."""
    for item in os.listdir(path):
        click.echo(item)

@click.command()
@click.argument('src')
@click.argument('dst')
def cp(src, dst):
    """Copiar SRC a DST."""
    shutil.copy(src, dst)
    click.echo(f'Copiado "{src}" a "{dst}".')

@click.command()
@click.argument('src')
@click.argument('dst')
def mv(src, dst):
    """Mover SRC a DST."""
    shutil.move(src, dst)
    click.echo(f'Movido "{src}" a "{dst}".')

@click.command()
@click.argument('path')
def rm(path):
    """Eliminar el archivo o directorio en PATH."""
    if os.path.isdir(path):
        shutil.rmtree(path)
    else:
        os.remove(path)
    click.echo(f'Eliminado "{path}".')

cli.add_command(touch)
cli.add_command(mkdir)
cli.add_command(ls)
cli.add_command(cp)
cli.add_command(mv)
cli.add_command(rm)

```

3. Cliente API

Un cliente API permite a los usuarios interactuar con APIs web desde la línea de comandos.

Características:

- Realizar solicitudes GET, POST, PUT y DELETE.

  GET: Recuperar datos de un servidor.
  POST: Enviar datos al servidor para crear un nuevo recurso.
  PUT: Enviar datos al servidor para actualizar un recurso existente.
  DELETE: Eliminar un recurso en el servidor.

  Implementación con *requests.get, requests.post, requests.put, requests.delete.*

- Mostrar respuestas de la API.

  Imprimir la respuesta del servidor en la terminal.
  Usar *click.echo(response.text)* para mostrar los datos recuperados o mensajes del servidor.

- Manejar autenticación y encabezados personalizados.

  Enviar información adicional con las solicitudes HTTP.
  Usar el parámetro *headers* en *requests* para agregar encabezados personalizados.
  Incluir *tokens* de autenticación en los encabezados.
  Implementar autenticación y encabezados personalizados con *click.option*.

Ejemplo:

```python
import click
import requests

@click.group()
def cli():
    """Un simple cliente API."""
    pass

@click.command()
@click.argument('url')
def get(url):
    """Realizar una solicitud GET a URL."""
    response = requests.get(url)
    click.echo(response.text)

@click.command()
@click.argument('url')
@click.argument('data', required=False)
def post(url, data=None):
    """Realizar una solicitud POST a URL con DATA."""
    response = requests.post(url, data=data)
    click.echo(response.text)


@click.command()
@click.argument('url')
@click.argument('data', required=False)
def put(url, data=None):
    """Realizar una solicitud PUT a URL con DATA."""
    response = requests.put(url, data=data)
    click.echo(response.text)

@click.command()
@click.argument('url')
def delete(url):
    """Realizar una solicitud DELETE a URL."""
    response = requests.delete(url)
    click.echo(response.text)

cli.add_command(get)
cli.add_command(post)
cli.add_command(put)
cli.add_command(delete)

if __name__ == '__main__':
    cli()

```

4. Herramienta de Backup

Una herramienta de backup permite a los usuarios realizar copias de seguridad de archivos y directorios, y restaurarlos desde las copias de seguridad.

Características:

- Crear copias de seguridad de archivos y directorios.

  Permite duplicar datos y almacenarlos en una ubicación segura para protección contra pérdida, corrupción o eliminación accidental.
  Utiliza el comando *backup* en Click.
  Implementa la creación de copias de seguridad mediante *shutil.copytree*.

- Restaurar archivos y directorios desde las copias de seguridad.

  Permite copiar los datos respaldados de vuelta a su ubicación original o a una nueva ubicación.
  Utiliza el comando *restore* en Click.
  Implementa la restauración de datos mediante *shutil.copytree*.

- Listar las copias de seguridad disponibles.

  Permite mostrar todos los archivos y directorios de respaldo que se encuentran en una ubicación específica.
  Utiliza el comando *list_backups* en Click.
  Implementa el listado de copias de seguridad mediante *os.listdir*.

Ejemplo:

```python
import click
import shutil
import os

@click.group()
def cli():
    """Una simple herramienta de backup."""
    pass

@click.command()
@click.argument('source')
@click.argument('backup')
def backup(source, backup):
    """Crear una copia de seguridad de SOURCE en BACKUP."""
    shutil.copytree(source, backup)
    click.echo(f'Backup de "{source}" creado en "{backup}".')

@click.command()
@click.argument('backup')
@click.argument('destination')
def restore(backup, destination):
    """Restaurar BACKUP en DESTINATION."""
    shutil.copytree(backup, destination)
    click.echo(f'Backup "{backup}" restaurado en "{destination}".')


@click.command()
@click.argument('backup_dir')
def list_backups(backup_dir):
    """Listar todas las copias de seguridad en BACKUP_DIR."""
    backups = os.listdir(backup_dir)
    if not backups:
        click.echo("No hay copias de seguridad disponibles.")
    else:
        for backup in backups:
            click.echo(backup)

cli.add_command(backup)
cli.add_command(restore)
cli.add_command(list_backups)

if __name__ == '__main__':
    cli()

```

5. Gestor de Configuraciones

Un gestor de configuraciones permite a los usuarios leer, escribir y listar configuraciones desde la línea de comandos.

Características:

- Leer configuraciones desde un archivo.

  Extraer valores de configuración almacenados en un archivo específico.
  Utiliza el comando *read* en Click.
  Implementa la lectura de configuraciones mediante *configparser.ConfigParser.*

- Escribir configuraciones en un archivo.

  Guardar o actualizar valores de configuración en un archivo específico.
  Utiliza el comando *write* en Click.
  Implementa la escritura de configuraciones mediante *configparser.ConfigParser*.

- Listar todas las configuraciones.

  Mostrar todas las secciones y sus correspondientes opciones y valores en un archivo de configuración.
  Utiliza el comando list en Click.
  Implementa el listado de configuraciones mediante *configparser.ConfigParser*.

Ejemplo:

```python
import click
import configparser

@click.group()
def cli():
    """Un simple gestor de configuraciones."""
    pass

@click.command()
@click.argument('file')
@click.argument('section')
@click.argument('option')
def read(file, section, option):
    """Leer una configuración."""
    config = configparser.ConfigParser()
    config.read(file)
    value = config.get(section, option)
    click.echo(f'{section}.{option} = {value}')

@click.command()
@click.argument('file')
@click.argument('section')
@click.argument('option')
@click.argument('value')

def write(file, section, option, value):
    """Escribir una configuración."""
    config = configparser.ConfigParser()
    config.read(file)
    if not config.has_section(section):
        config.add_section(section)
    config.set(section, option, value)
    with open(file, 'w') as configfile:
        config.write(configfile)
    click.echo(f'{section}.{option} escrito con valor {value}')

@click.command()
@click.argument('file')
def list(file):
    """Listar todas las configuraciones."""
    config = configparser.ConfigParser()
    config.read(file)
    for section in config.sections():
        click.echo(f'[{section}]')
        for option in config.options(section):
            value = config.get(section, option)
            click.echo(f'{option} = {value}')
        click.echo()

cli.add_command(read)
cli.add_command(write)
cli.add_command(list)

if __name__ == '__main__':
    cli()

```

Conclusiones

Click es una biblioteca poderosa y fácil de usar que permite a los desarrolladores de Python crear aplicaciones de línea de comandos con una sintaxis limpia y mínima configuración. Las aplicaciones que se pueden desarrollar con Click son diversas y pueden incluir desde gestores de tareas y archivos, clientes API, herramientas de backup, hasta gestores de configuraciones, entre otras. Estas aplicaciones son útiles en diversos escenarios del mundo real, facilitando tareas administrativas y operativas desde la terminal.

Link a Repositorio Github

https://github.com/jcbrandan31/Trabajajo_Practico_Grupo_17_COM2