Documentación

Automatización CAMI APP

Contenido

CONCEPTO GENERAL

El objetivo es automatizar tareas repetitivas en Cami o comprobar que no existan errores, principalmente relacionados con la consulta de información. Las funcionalidades implementadas abarcan desde el inicio de sesión, selección de empresa, la creación de tipos documentales y su consulta, hasta la creación de contratos, la modificación de datos de los contratos, y la inserción de cláusulas, comentarios y aprobaciones/rechazos. Cada una de estas partes es independiente, lo que permite adaptar el programa según las necesidades específicas.

El proyecto está desarrollado en Python, utilizando la biblioteca Selenium para la automatización del navegador web. Se emplea el navegador Brave para las pruebas, aunque con la configuración adecuada, el código debería adaptarse a otros navegadores basados en Chromium.

Notas:

1. Los archivos llevan el nombre del módulo de Cami que se está automatizando. Si el módulo requiere más archivos, estos se encuentran en una carpeta con el mismo nombre.

2. Para modificar el flujo general del bot, solo es necesario cambiar el orden o comentar cada flujo en el archivo constantes.py. Este mismo archivo incluye todos los datos genéricos utilizados en los campos de entrada de los flujos de cada módulo, que pueden modificarse si es necesario. El código se adapta automáticamente a la cantidad de elementos en las listas. Por ejemplo, en la siguiente imagen se definen tres usuarios externos con datos genéricos utilizados en el flujo de creación de contratos. En este caso, como hay tres usuarios, el bot presiona el botón tres veces e ingresa los datos correspondientes.

- Si se desean agregar más usuarios externos, solo hay que añadirlos a este diccionario.
- Si no se quiere agregar ninguno, se puede comentar o eliminar el contenido del diccionario.
  En ambos casos, el bot se ajusta automáticamente y ejecuta correctamente el flujo, creando usuarios o no.

```
CONTRATO_USUARIOS_EXTERNOS = {
    "Usuario ext1": {
        "Correo electrónico": "ejemplo1@gmail.com",
        "Nombres": "1Nombre1 1Nombre2",
        "Apellidos": "1Apellido1 1Apellido2",
        "Permisos": ["Ver cláusulas y campos actuales", "Editar campos", "Comentar campos"]
    },
    "Usuario ext2": {
        "Correo electrónico": "ejemplo2@gmail.com",
        "Nombres": "2Nombre1 2Nombre2",
        "Apellidos": "2Apellido1 2Apellido2",
        "Permisos": ["Ver cláusulas y campos actuales", "Ver historial de campos"]
    },
    "Usuario ext3": {
        "Correo electrónico": "ejemplo3@gmail.com",
        "Nombres": "3Nombre1 3Nombre2",
        "Apellidos": "3Apellido1 3Apellido2",
        "Permisos": ["Ver cláusulas y campos actuales", "Firmar contrato"]
    },
}
```

3. Es necesario crear un archivo .env con las credenciales de inicio de sesión de la página, utilizando la siguiente estructura:

En .env

```
CAMI_EMAIL=email@gmail.com
CAMI_PASSWORD=password
```

4. Todo el flujo se registra tanto en la terminal como en un archivo llamado automatizacion.log. Existen varios niveles de registro (log) dependiendo de lo que esté ocurriendo:

INFO: Muestra los pasos que está siguiendo el bot o los datos encontrados.

WARNING: Indica que el bot no siguió el flujo exactamente como estaba definido, pero puede continuar. Este mensaje puede aparecer, por ejemplo, cuando se busca un elemento que no está visible (como un botón de "Guardar" que solo aparece al realizar cambios, y estos no se hicieron). No necesariamente implica un error en el código o en la página. Cada *warning* describe la razón por la que fue generado. Es importante revisarlos, aunque en la mayoría de los casos no afectan al flujo ni indican problemas graves en la página.

ERROR: Este mensaje se muestra cuando el bot no puede seguir el flujo normal debido a algo imprevisto o que no se consideraba posible. En algunos casos, puede impedir que el bot continúe con el flujo. Estos mensajes suelen aparecer si ha habido cambios en la página, por lo que siempre deben revisarse para identificar posibles ajustes necesarios.

CRITICAL: Este mensaje aparece únicamente cuando el bot queda "atorado" en alguna página. Por ejemplo, si al iniciar sesión las credenciales son inválidas y no se puede acceder al menú de Cami se genera el mensaje CRITICAL. Es raro que ocurra, pero cuando sucede, la automatización se detiene automáticamente y libera los recursos. Siempre es necesario revisar estos mensajes para resolver el problema y permitir que el flujo continúe correctamente.

5. En caso de que se muestre un error de este tipo en el log o terminal:

Este error significa que el bot no pudo encontrar o acceder a algún elemento del HTML de la página y no se esta manejando bien esa excepción, en el código actual se intentó cambiar este tipo de errores por solamente los errores del log que se vieron anteriormente para cada flujo, por lo que suelen aparecer en casos no previstos durante la creación del código. Este tipo de errores son buenos indicadores de que algo en la página ha cambiado desde el momento de creación del código o no se pensó en un caso específico con ciertos datos ingresados.

INFORMACIÓN GENERAL

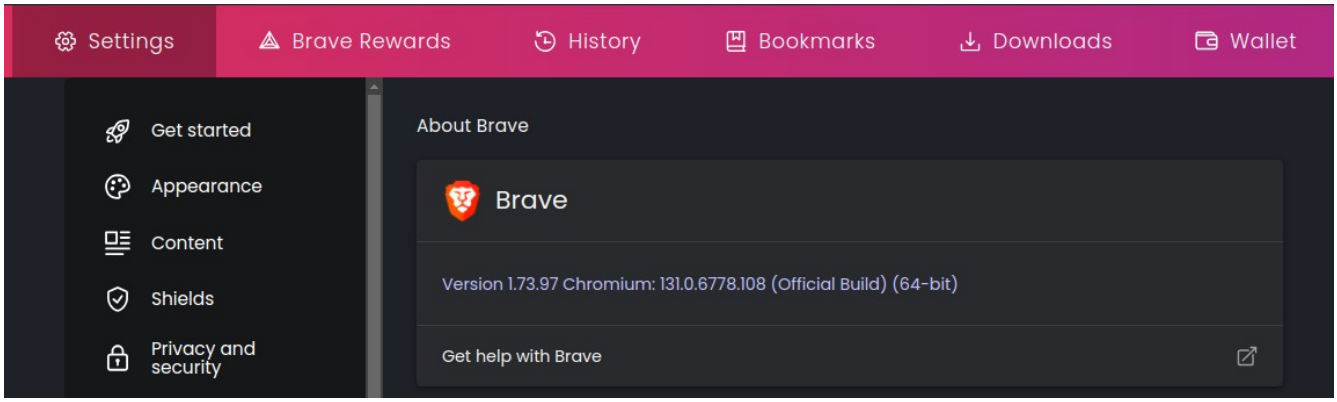
1. Requisitos de Instalación

Para ejecutar este script se deben cumplir los siguientes requisitos:

- **Sistema Operativo**: Cualquier distribución de Linux. Para la creación se utilizó Ubuntu 24.04.
- **Python**: Versión 3.12.3 o superior.

2. Herramientas Necesarias

1. **Selenium**: 4.6.0 (Se encuentra en el archivo requirements.txt)
2. **WebDriver**:
    - Descarga el WebDriver correspondiente a la versión de Chromium de tu navegador Brave:
    - https://googlechromelabs.github.io/chrome-for-testing/



Descargará un zip, lo único importante es el archivo chromedriver. Su ruta, al igual que la del navegador instalado, debe definirse en la primera sección del archivo constantes.py. Se recomienda verificar las rutas específicas de tu computadora y modificar estos datos si es necesario. Si no esta instalado, es mejor utilizar las rutas que ya están definidas.

```
# Rutas y configuraciones
CHROMEDRIVER_PATH = "/usr/local/bin/chromedriver"
BRAVE_BINARY_PATH = "/snap/brave/456/opt/brave.com/brave/brave-browser"
```

ARCHIVOS

configuracion_selenium.py

Este archivo centraliza las herramientas y flujos necesarios para ejecutar la automatización. A continuación, se describen las secciones principales:

- **Importación de bibliotecas y módulos:** En esta sección, se importan las bibliotecas necesarias y los módulos específicos del proyecto.

- **Módulos del proyecto:** Incluyen funciones que permiten navegar y ejecutar diferentes flujos, como la inserción de documentos, consulta de contratos y manejo de constantes. Estas funciones están asociadas a una palabra clave en el diccionario de la línea 32:

```python
# -x-x-x- FLUJOS DE SECCIONES Y FUNCIONES DISPONIBLES PARA EL FLUJO DE LA AUTOMATIZACIÓN -x-x-x-

FLUJOS_DISPONIBLES = {
    "Inserción de Documentos": lambda driver: (
        navegar_a_seccion(driver, "Inserción de Documentos"),
        insercion_seleccionar_documento(driver),
        completar_datos_tabla(driver),
    ),
    "Consulta de Documentos": lambda driver: (
        navegar_a_seccion(driver, "Consulta de Documentos"),
        consulta_seleccionar_documento(driver),
        consulta_busqueda_y_verificacion(driver)
    ),
    "Crear Contrato": lambda driver: (
        navegar_a_seccion(driver, "Contratos"),
        nuevo_contrato(driver),
    ),
    "Consultar Contrato": lambda driver: (
        navegar_a_seccion(driver, "Contratos"),
        consulta_contrato(driver),
        flujo_contrato_interno(driver),
    ),
}
```

Estas son las funciones de cada flujo (las mismas funciones se encuentran en el archivo con el mismo nombre). Desde aquí se define el orden de ellas. Si no se quiere ejecutar una función, se puede comentar en esta sección; sin embargo, **NO** es para modificar el flujo general del bot. Es decir, este diccionario solamente relaciona el nombre del flujo que se elige con las funciones correspondientes para facilitar la configuración. Por lo tanto, es mejor dejarlo tal como se encuentra, ya

que podría provocar que el bot intente acceder a elementos que no existen en esa página. Si se desea cambiar el flujo a ejecutar, debe hacerse en el archivo constantes.py en la lista FLUJOS con el mismo nombre:

```python
# -x-x-x-x-x- INGRESO DE DATOS -x-x-x-x-x-

# Nombre de la empresa a la que se quiere ingresar
NOMBRE_EMPRESA = "QA Cami"

# Flujos dinámicas definidas por el usuario, comentar lo que no se utilizará en la automatización.
FLUJOS = [
    #"Inserción de Documentos",
    "Consulta de Documentos",
    "Crear Contrato",
    "Consultar Contrato",
]
```

Esta es la lista que define el orden de ejecución de las secciones (acciones generales de los módulos de Cami). Cada elemento de la lista corresponde al ID del diccionario mostrado anteriormente. Aquí no hay ningún problema por cambiar el orden o comentar alguna de las secciones. Se irán ejecutando en el mismo orden en el que están definidos en esta lista.

Continuando con el archivo de configuración, encontramos:

3. Clase CriticalHandler

Sirve por si se registra un error crítico (CRITICAL) en el log, detiene la ejecución del programa. Intenta cerrar el navegador en caso de un fallo grave, garantizando que no queden procesos abiertos. Esto fue necesario implementarlo ya que naturalmente Selenium no se detiene hasta terminar de ejecutar todo el script, incluso si se encuentra atorado en alguna página, por ejemplo, si no se logró iniciar sesión se muestra un error CRITICAL terminando con la automatización ya que no hay forma de que pueda continuar.

4. Configuración del Logging

- **configurar_logging(driver=None)**: Configura el registro de logs para monitorear la ejecución:
    - Guarda los logs en un archivo (automatizacion.log) y los imprime en la terminal.
    - Incluye un manejador personalizado (CriticalHandler) para detener la ejecución si ocurre un error crítico.
    - Formato de los logs: timestamp - nivel - mensaje.

5. Configuración del WebDriver

- **configurar_driver()**: Configura el navegador Brave para la automatización:
    - Usa el CHROMEDRIVER_PATH y BRAVE_BINARY_PATH definidos en las constantes.
    - Maximiza la ventana del navegador al iniciar (Es importante que se muestre en pantalla completa, ya que en ocasiones el bot no logra acceder a componentes de la página que no están cargados, por lo que si por alguna razón el flujo no continúa intenta arrastrar la ventana a un monitor y no una pantalla de la laptop, aunque este error es muy poco común).
    - Navega automáticamente a la URL base (BASE_URL) configurada en las constantes.

En general, no es necesario modificar nada de este archivo, solamente el diccionario `FLUJOS_DISPONIBLES` en caso de que se quiera agregar o eliminar alguna función específica de ese módulo. Si se quiere agregar la automatización de otro módulo nuevo al proyecto, solo deben importarse las funciones correspondientes (en orden de ejecución) y agregarlas en el diccionario del archivo (configuracion_selenium.py).

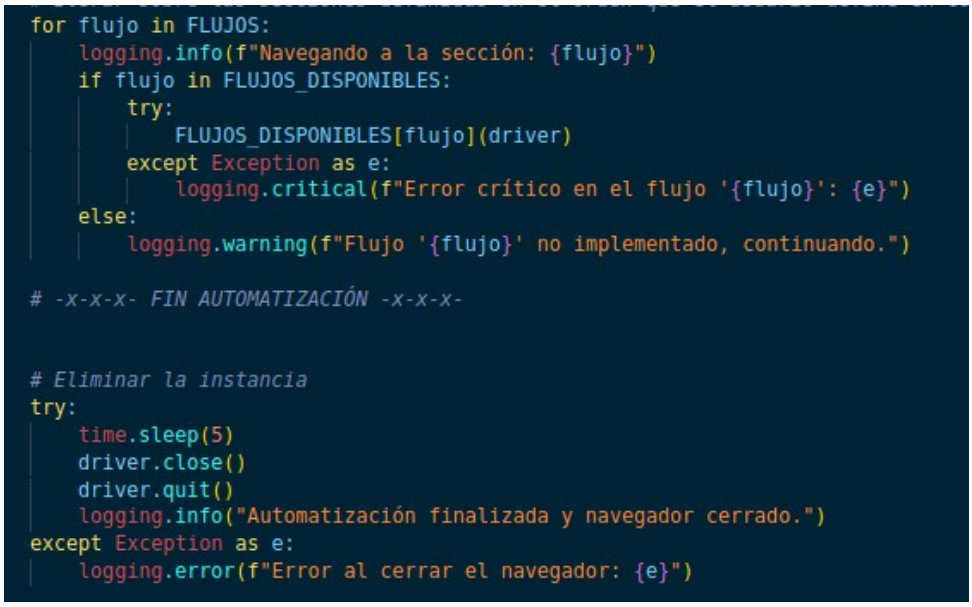
Main.py

En este archivo comienza la automatización. Primero ejecuta las funciones del archivo login.py. Al inicio, verifica en qué “estado” se encuentra, es decir, si ya inició sesión, si ya eligió empresa o si está en el home de Cami. Dependiendo del estado en el que se encuentre, ejecuta la función de login, seleccionar empresa (del archivo seleccionar_empresa.py) con el propósito de ingresar al portal.

```
# -x-x-x- INICIO AUTOMATIZACIÓN -x-x-x-

estado = verificar_estado(driver, driver.current_url)

# Comprueba si ya inició sesión y si ya eligió empresa, dependiendo del caso ejecuta esa parte
if estado == "sin_sesion":
    logging.info("No se ha iniciado sesión")
    auto_cami_login(driver, USER_EMAIL, USER_PASSWORD)
    auto_cami_seleccionar_empresa(driver, NOMBRE_EMPRESA)

elif estado == "sin_empresa":
    logging.info("No se ha seleccionado empresa")
    auto_cami_seleccionar_empresa(driver, NOMBRE_EMPRESA)

elif estado == "con_empresa":
    logging.info("Sesión iniciada y empresa ya seleccionada")
```

Esto existe porque, si el usuario da clic en “guardar contraseña” o si no se cerró correctamente el navegador, la sesión puede quedar abierta. Esto puede provocar que el bot intente iniciar sesión cuando ya está en el *home* o en la página de “elegir empresa”. No es necesario hacer ninguna modificación en esta parte del código. Si no se puede iniciar sesión correctamente, verifique el archivo .env con las credenciales o las funciones del archivo login.py.

A continuación (en el archivo main.py), se itera sobre los flujos definidos en el orden que el usuario establece en la lista FLUJOS de constantes.py, utilizando como referencia el diccionario FLUJOS_DISPONIBLES. Esto dirige al bot a cada uno de los módulos de los archivos. Finalmente, se detiene el bot, cerrando el navegador y liberando los recursos utilizados, concluyendo así la automatización.



```
for flujo in FLUJOS:
    logging.info(f"Navegando a la sección: {flujo}")
    if flujo in FLUJOS_DISPONIBLES:
        try:
            FLUJOS_DISPONIBLES[flujo](driver)
        except Exception as e:
            logging.critical(f"Error crítico en el flujo '{flujo}': {e}")
    else:
        logging.warning(f"Flujo '{flujo}' no implementado, continuando.")

# -x-x-x- FIN AUTOMATIZACIÓN -x-x-x-


# Eliminar la instancia
try:
    time.sleep(5)
    driver.close()
    driver.quit()
    logging.info("Automatización finalizada y navegador cerrado.")
except Exception as e:
    logging.error(f"Error al cerrar el navegador: {e}")
```

navegar_modulos.py

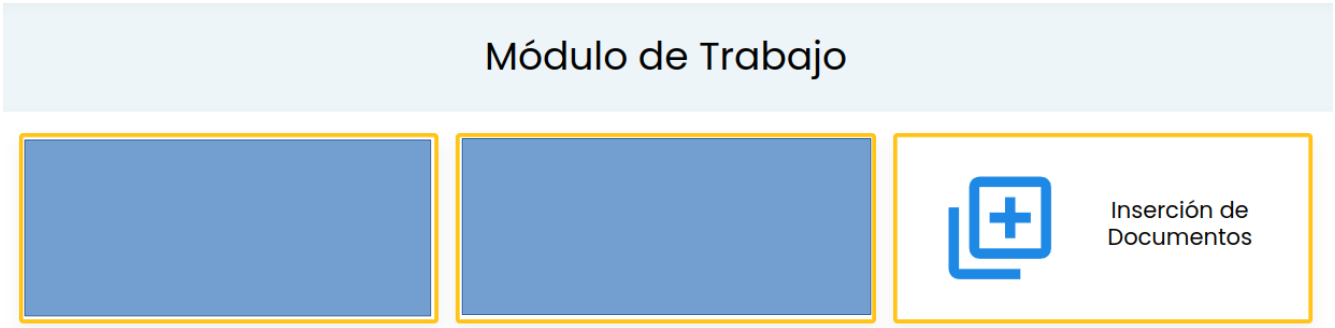
Como su nombre lo indica, su única función es ir a la sección de “trabajo” de Cami (la que aparece en el navbar general) e ingresar al módulo que se desee, utilizando el texto visible en la página para que sea fácil de usar. Esta función es genérica y se emplea en cada flujo. Por ejemplo:

La primera función que ejecuta el flujo de Inserción de Documentos es:

navegar_a_seccion(driver, "Inserción de Documentos").

```
FLUJOS_DISPONIBLES = {
    "Inserción de Documentos": lambda driver: (
        navegar_a_seccion(driver, "Inserción de Documentos"),
        insercion_seleccionar_documento(driver),
        completar_datos_tabla(driver),
    ),
```

Debe utilizarse exactamente el mismo texto que aparece en el módulo del menú de Cami para que el bot lo encuentre correctamente.

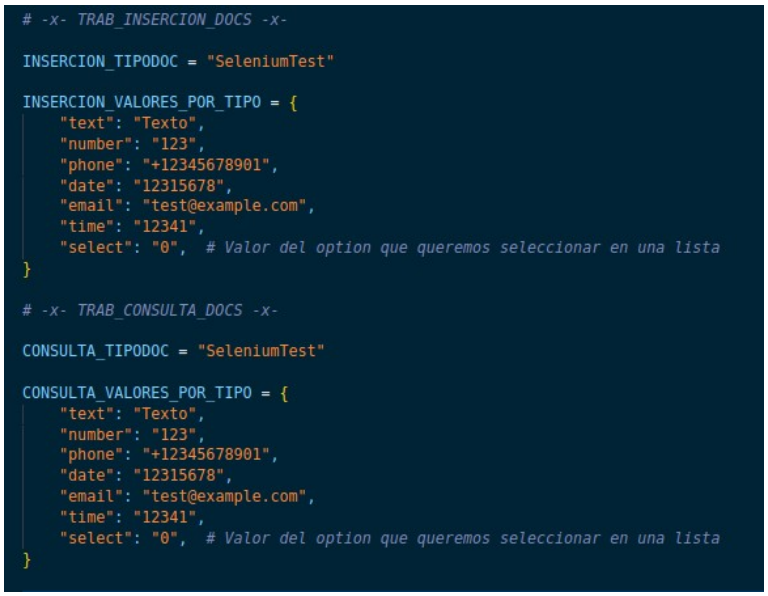


seleccionar_empresa.py

Este archivo tiene una única función, con la cual se selecciona la empresa en la página inmediata después del login. Esta función se ejecuta desde main.py después de verificar el estado (sin sesión, sin empresa o con empresa). Si se desea elegir otra empresa, se debe modificar NOMBRE_EMPRESA en constantes.py.

constantes.py

Este es el archivo principal donde se deben modificar los datos. Está dividido en secciones con los nombres de los archivos que utilizan esas constantes:



```
# -x- TRAB_INSERCION_DOCS -x-

INSERCION_TIPODOC = "SeleniumTest"

INSERCION_VALORES_POR_TIPO = {
    "text": "Texto",
    "number": "123",
    "phone": "+12345678901",
    "date": "12315678",
    "email": "test@example.com",
    "time": "12341",
    "select": "0",  # Valor del option que queremos seleccionar en una lista
}

# -x- TRAB_CONSULTA_DOCS -x-

CONSULTA_TIPODOC = "SeleniumTest"

CONSULTA_VALORES_POR_TIPO = {
    "text": "Texto",
    "number": "123",
    "phone": "+12345678901",
    "date": "12315678",
    "email": "test@example.com",
    "time": "12341",
    "select": "0",  # Valor del option que queremos seleccionar en una lista
}
```

Como se mencionó anteriormente, al inicio de este archivo hay una lista llamada FLUJOS. En esta lista, se puede comentar el flujo que no se desea ejecutar o cambiar el orden en el que se ejecutan. Las funciones asociadas a cada flujo se encuentran en el archivo configuracion_selenium.py.

- **Nota importante:** Algunas funciones en configuracion_selenium.py dependen de otras para ejecutarse correctamente, por lo que no se recomienda hacer cambios si no se entiende completamente cómo funcionan. Por ejemplo, en el caso del flujo "Consultar Contrato", las funciones se encuentran en el archivo trab_contratos_consultar.py, dentro de la función **flujo_contrato_interno**. Estas funciones no dependen de otras para ejecutarse correctamente, por lo que pueden comentarse si es necesario.

Dado que el archivo constantes.py contiene muchos datos, se recomienda ejecutar el flujo correspondiente para identificar en qué momento se utilizan y modificarlos según sea necesario. Dentro del archivo, cada dato está comentado con su propósito específico. Los datos suelen procesarse de arriba hacia abajo, según el flujo.

CÓDIGO DE LOS FLUJOS

Nota: Para continuar con la lectura se recomienda estar viendo el módulo al que hace referencia el flujo para entender la explicación.

1. “Inserción de Documentos”

Este flujo inicia en el módulo de Cami con el mismo nombre, el orden en el que se ejecutan las funciones es este:

```
FLUJOS_DISPONIBLES = {
    "Inserción de Documentos": lambda driver: (
        navegar_a_seccion(driver, "Inserción de Documentos"),
        insercion_seleccionar_documento(driver),
        completar_datos_tabla(driver),
    ),
```

Las funciones del flujo se encuentran en el archivo trab_insercion_docs.py:

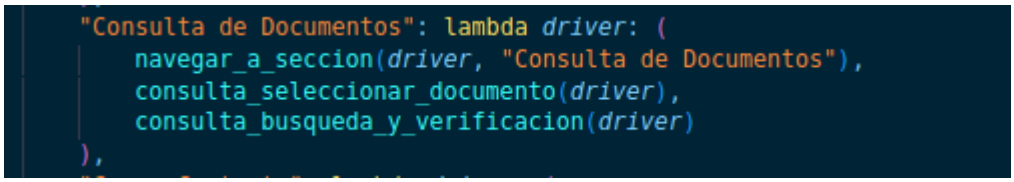
insercion_seleccionar_documento, se encarga de buscar un tipo específico de documento en un sistema. Para ello, utiliza una barra de búsqueda en la que escribe el texto definido en la constante INSERCION_TIPODOC. Una vez que el sistema filtra los resultados, la función identifica y selecciona el botón correspondiente al documento deseado. En caso de errores, como que el documento no se encuentre o los elementos no sean interactuables, la función registra advertencias en los logs.

La segunda función, **completar_datos_tabla**, aborda el llenado de formularios en una tabla. Identifica campos de entrada como input y menús desplegables (select) y los completa con valores genéricos definidos en un diccionario llamado INSERCION_VALORES_POR_TIPO. La función verifica si los campos tienen valores predefinidos o están deshabilitados antes de modificarlos, para evitar sobrescribir datos ya establecidos. Tras completar todos los campos, localiza el botón para guardar los cambios, asegurándose de que esté visible y operable antes de realizar clic en él. Si el clic convencional falla, utiliza JavaScript para forzar la interacción. El flujo incluye pausas estratégicas para permitir que las acciones se ejecuten correctamente en interfaces que podrían tener retrasos de carga.

Ambas funciones están diseñadas para manejar errores, registrando los problemas encontrados en los logs y capturando pantallas en caso de fallos críticos. Esto permite identificar y solucionar posibles inconvenientes en la automatización. Estas tareas se ejecutan de manera secuencial para garantizar que el documento se seleccione correctamente antes de proceder con la inserción de datos y el guardado.

2. “Consulta de Documentos”

Este flujo inicia en el módulo de Cami con el mismo nombre, el orden en el que se ejecutan las funciones es este:



La función **consulta_seleccionar_documento** busca y selecciona un tipo específico de documento ingresado en un campo de búsqueda. Una vez encontrado el documento, hace clic en el botón "Seleccionar". Además, localiza el checkbox llamado "Búsqueda Acumulativa" y, si está marcado, lo desmarca (esto para que solamente aparezcan los datos que se acaban de consultar en la tabla de abajo).

La función **consulta_busqueda_y_verificacion** realiza búsquedas dinámicas llenando campos de entrada (como `input` y `select`) con valores predefinidos almacenados en el diccionario CONSULTA_VALORES_POR_TIPO. Tras completar los campos, interactúa con un botón para realizar la búsqueda y selecciona la opción de "1000 resultados" en el listado. Finalmente, verifica que los valores ingresados coincidan con los datos mostrados en una tabla dinámica de abajo. La verificación se realiza columna por columna, asegurando que cada celda contiene el valor esperado. Lo que se busca comprobar con este flujo es que la búsqueda de documentos funcione correctamente.

Funciones Auxiliares (se llaman en repetidas ocasiones o en ciertos casos, por lo que no aparecen en los flujos de CONFIGURACION_SELENIUM, son necesarias para que cada función del flujo funcione correctamente):

- verificar_tabla: Valida que los valores en una columna específica de la tabla coincidan con los ingresados durante la búsqueda. Esta función incluye formateo especial para ciertos tipos de datos, como fechas y horas.

- consulta_seleccionar_mil_resultados: Cambia el número de resultados mostrados a "1000" utilizando un menú desplegable.

- limpiar_inputs: Limpia todos los campos visibles antes de realizar nuevas búsquedas, asegurando que no queden valores residuales.

- es_visible: Verifica si un elemento es visible en la interfaz para garantizar que las interacciones sean efectivas.

Se utiliza `WebDriverWait` para garantizar que los elementos estén presentes, visibles y clickeables antes de interactuar con ellos. También se emplean pausas (`time.sleep`) para manejar posibles retrasos en la carga de la interfaz.

3. “Crear Contrato”

Este flujo inicia en el módulo de Cami llamado Contratos, el orden en el que se ejecutan las funciones es este:

La función principal, `nuevo_contrato`, interactúa con varios elementos de la interfaz gráfica del usuario, como formularios, botones y listas desplegables, para completar las tareas necesarias.

Primero, la función inicia un proceso esperando que se carguen elementos interactivos de la página, como una lista desplegable para seleccionar el contrato deseado. Este contrato es identificado por su nombre, el cual está definido previamente en una constante (CONTRATO_NOMBRE). Una vez seleccionado el contrato, la función procede a confirmar la acción haciendo clic en un botón "Confirmar", asegurándose de que se actualice la vista a la nueva página requerida.

El siguiente paso consiste en rellenar los detalles del contrato. La descripción del contrato, almacenada en otra constante (CONTRATO_DESCRIPCION), se escribe en un área de texto específica, y después se agregan etiquetas asociadas al contrato (CONTRATO_TAGS). Estas etiquetas se introducen una por una, simulando la interacción del usuario con el campo correspondiente, utilizando la tecla TAB para completar cada etiqueta.

Luego, se selecciona el método de firma preferido, ya sea "Firma Electrónica" o "Subida de Documento", basándose en el valor de la constante CONTRATO_FIRMA. Estos archivos de firmas deben ser agregados desde antes manualmente en la página del módulo de Contratos para que se pueda generar el contrato nuevo.

Posteriormente, se rellenan campos adicionales del contrato según un diccionario de valores predefinidos (CONTRATO_VALORES_POR_NOMBRE), estos deben coincidir con la plantilla del contrato creada desde la sección de Administración en

caso de que se quiera utilizar otro contrato. Cada clave del diccionario representa un campo, y su valor es el texto que se debe introducir en dicho campo. La función incluye validaciones para manejar casos en los que los campos no se encuentren disponibles o no puedan ser editados.

La función también gestiona la adición de usuarios al contrato. Esto se realiza en dos partes: primero, se agregan usuarios internos utilizando un botón dedicado, y luego, usuarios externos siguiendo un procedimiento similar. Los datos de los usuarios internos y externos, como nombres, correos electrónicos y permisos, se obtienen de constantes (CONTRATO_USUARIOS y CONTRATO_USUARIOS_EXTERNOS). Cada usuario es asignado a un cuadro específico de la interfaz, asegurándose de que los campos de entrada correspondientes estén vacíos antes de llenarlos. Actualmente están “quemados” en el código con datos genéricos, si se quieren colocar datos de personas reales será necesario adaptarlos en el archivo .env.

Finalmente, una vez que todos los datos han sido introducidos y validados, se hace clic en un botón "Enviar" para completar la creación del contrato. La función incluye mensajes de registro (logging) en cada paso para proporcionar información detallada sobre el progreso y para facilitar la resolución de problemas en caso de errores.

Además de la función principal, se incluyen dos funciones auxiliares: agregar_usuario_cami y agregar_usuario_externo. Estas manejan de manera específica la lógica para procesar usuarios internos y externos respectivamente, asegurándose de identificar y evitar cuadros que ya estén ocupados o que estén colapsados en la interfaz.

4. "Consultar Contrato"

Este flujo inicia en el módulo de Cami llamado Contratos, el orden en el que se ejecutan las funciones es este:

Primero, el código importa las bibliotecas necesarias para la automatización se define una variable global `comentario_indice` para gestionar los comentarios en las cláusulas.

La función **consulta_contrato** realiza una búsqueda de un contrato específico en la página web. Esto implica ingresar una descripción en un campo de búsqueda, esperar la carga de resultados dinámicos y buscar en los desplegables el contrato con el nombre deseado. Si se encuentra, se accede a su contenido y se despliega la tabla asociada. A continuación, busca el contrato en una posición específica y, si está disponible, interactúa con su botón de acción, como "Ver Cláusulas".

La función **flujo_contrato_interno** coordina varias operaciones, como modificar datos del contrato, insertar cláusulas y aprobar o rechazar el contrato. También incluye una funcionalidad para generar un documento basado en los datos procesados. En esta sección se puede comentar o cambiar el orden de las llamadas a las otras funciones, no se encuentran en configuracion_selenium.py porque las funciones de FLUJOS_DISPONIBLES deben ir en ese orden en la mayoría de los casos mientras que dentro de flujo_contrato_interno pueden funcionar independientemente, cambiando el orden o comentando alguna, como se ve en la siguiente imagen.

```
def flujo_contrato_interno(driver):
    #   -x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-
    # -x-x-x-x-x- Comentar aqui la llamada a la función que no se quiera ejecutar -x-x-x-x-x-
    #   -x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-
    modificar_datos_contrato(driver)
    # comentar_clausula(driver)
    insertar_clausulas(driver)
    aprobar_rechazar_contrato(driver)
```

La función **modificar_datos_contrato** se encarga de actualizar los primeros valores en el contrato. Busca elementos con etiquetas y sus campos de entrada asociados, limpia los valores existentes y escribe los nuevos valores proporcionados. Si se detectan cambios en las clausulas, presiona un botón para guardar los datos, en caso de que no se realicen cambios o se ingrese un dato identico al original, no aparece el botón de guardar (puede mostrarse un warning sobre eso en el log).

En la función **insertar_clausulas**, se gestionan cláusulas específicas dentro de una tabla. Para cada cláusula en la lista proporcionada, la función localiza el área de edición correspondiente en una fila de la tabla, escribe el texto de la cláusula y, si es necesario, guarda los cambios realizados.

insertar_comentario añade comentarios a las cláusulas del contrato. La función utiliza un índice global que se declara al inicio del archivo para gestionar múltiples comentarios en secuencia, esta es genérica y solo se ejecuta cuando la ventana de comentario emergente esta en pantalla, por lo que se llama cada vez que se ingresa un comentario nuevo desde **comentar_clausula**. Si el comentario actual está vacío, simplemente cierra la ventana emergente asociada. Para comentarios no vacíos, localiza el campo correspondiente, escribe el comentario y lo envía.

En la función **comentar_clausula**, primero interactúa con un botón general para abrir una ventana emergente de comentarios. Luego, procesa cada fila de la tabla principal, localizando botones de comentario específicos para cada cláusula, y llama a **insertar_comentario** para realizar la acción.

Finalmente, la función **aprobar_rechazar_contrato** está destinada a manejar la aprobación o el rechazo de cada clausula basándose en la lista APROBAR_RECHAZAR_CONTRATO de constantes.py.

Luego de llamar a estas funciones, en **flujo_contrato_interno** se presiona el botón de generar contrato, terminando con el flujo.

AGREGAR NUEVOS FLUJOS

Para crear nuevos flujos en el programa sigue los siguientes pasos:

1. Crea el archivo con el nombre del módulo a automatizar, por ejemplo para el módulo “Bandeja de Trabajo” sería trab_bandeja_de_trabajo.py.

2. Define las funciones que quieras ejecutar dentro de este archivo, por ejemplo:

```
def nuevo_flujo_cami(driver)

def flujo_cami_solicitud_de_entrega(driver)

def buscar_flujo_cami(driver)
```

Estas funciones deben empezar a funcionar desde que el bot se encuentra dentro del módulo, ya que existe la función navegar_a_seccion.

Se pueden agregar otras funciones que se utilicen en repetidas ocasiones al final del código pero que son necesarias para el funcionamiento de alguna de las funciones principales (las que se acaban de definir), se puede indicar como “Funciones auxiliares”.

3. En el archivo configuracion_selenium.py, importar las funciones correspondientes y luego, en el diccionario FLUJOS_DISPONIBLES, agregar el nuevo flujo con sus respectivas funciones, quedaría algo como esto:

```
from testCamiApp.trab_bandeja_de_trabajo import (
        nuevo_flujo_cami,
        flujo_cami_solicitud_de_entrega,
        buscar_flujo_cami,
)
FLUJOS_DISPONIBLES = {
        # Los demás flujos...
        ),
        “Nuevo flujo de Cami”: lambda driver: (
                navegar_a_seccion(driver, “Bandeja de Trabajo”),
                nuevo_flujo_cami(driver)
                flujo_cami_solicitud_de_entrega(driver),
```

```
            buscar_flujo_cami(driver),
        ),
}
```

Notas:

Es importante siempre pasar como atributo de la función el driver, este incluye la configuración del bot de selenium.

Al agregar el flujo nuevo, utilizar al inicio la funcion navegar_a_seccion pasando como atributo el texto exacto del cuadro del módulo que aparece en la sección de trabajo de Cami como se mostró al inicio de este documento.

Se puede cambiar el nombre del elemento del diccionario por el nombre del flujo. En este ejemplo se colocó Nuevo flujo de Cami (que es lo que sucede en el módulo de Bandeja de Trabajo).

4. Una vez definidas las funciones en el orden que se ejecutarán en el flujo, solo se debe agregar el texto del diccionario en la lista del constantes.py, quedando algo como esto:

```
FLUJOS = [
        #“Inserción de Documentos”,
        #“Consulta de Documentos”,
        #“Crear Contrato”,
        #“Consultar Contrato”,
        “Nuevo flujo de Cami”,
]
```

Teniendo así esta lista el bot solamente ejecutará el flujo de “Nuevo flujo de Cami” después de iniciar sesión, elegir empresa, y finalizar este flujo, terminará la automatización.

Al hacerlo de esta forma cada flujo es independiente y el código se puede expandir sin nigún problema, siempre que en las nuevas funciones no se interactue con otras de otros archivos. Se recomienda ver el código de los flujos ya existentes para seguir la misma lógica e incluir los logs con la misma estructura.

USO DE SELENIUM

Selenium proporciona varios métodos para interactuar con elementos en una página web, siendo la eficiencia y claridad clave para elegir el método adecuado. El más eficiente es el uso de ID (`find_element(By.ID)`), ya que los navegadores están optimizados para localizar elementos con atributos `id`. Este método es ideal cuando el `id` del elemento es único y fijo. Si el atributo `id` no está disponible, el segundo método recomendado es buscar por nombre (`find_element(By.NAME)`), que utiliza el atributo `name` del elemento HTML y también ofrece un rendimiento rápido.

Cuando los identificadores únicos no están presentes al inspeccionar el elemento, se puede recurrir a los selectores CSS (`find_element(By.CSS_SELECTOR)`), que permiten buscar elementos por atributos, clases, o relaciones jerárquicas. Este método es flexible y eficiente, ideal para estructuras HTML complejas. Alternativamente, XPath (`find_element(By.XPATH)`) ofrece un alto grado de flexibilidad para buscar elementos utilizando condiciones complejas o rutas relativas, aunque generalmente es menos eficiente que los selectores CSS aunque mucho mas fácil de implementar, ya que solo se necesita copiar el XPATH del componente. Si se trata de buscar elementos con clases específicas o genéricas, se puede usar el método class (`find_element(By.CLASS_NAME)`), aunque debe hacerse con precaución, ya que las clases suelen repetirse en el HTML y Selenium solo utilizará el primer elemento encontrado que pertenece a esa clase.

Otras formas de interacción incluyen buscar elementos por etiquetas (`find_element(By.TAG_NAME)`), que puede ser útil para localizar grupos de elementos como tablas o listas, y por texto visible de enlaces (`find_element(By.LINK_TEXT)` o `find_element(By.PARTIAL_LINK_TEXT)`), ideal para encontrar enlaces en el DOM.

Además de localizar elementos, Selenium permite realizar diversas acciones sobre ellos. Puedes usar el método `click()` para hacer clic en un elemento, como un botón o enlace. Si necesitas ingresar texto en un campo, `send_keys()` es el método adecuado. Antes de escribir, `clear()` puede usarse para vaciar el contenido previo del campo. En casos donde el elemento no está visible en la pantalla, el método `execute_script()` con `scrollIntoView()` permite desplazar la página hasta que el elemento esté en vista, este último se recomienda utilizar siempre ya que en algunas ocasiones las pantallas son de diferentes tamaños y

aunque originalmente el código funciona bien, al ejecutarlo en una pantalla más pequeña puede que no pueda acceder al elemento, por esta razón, se recomienda hacer scroll hasta que el elemento seleccionado esté visible.

Cuando los elementos tardan en cargarse o dependen de eventos dinámicos, se utiliza WebDriverWait. Este método permite esperar explícitamente hasta que un elemento sea visible, clickeable, o cumpla con una condición específica, asegurando que las interacciones sean estables. Por ejemplo:

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

element = WebDriverWait(driver, 10).until(EC.visibility_of_element_located((By.ID, "element-id")))
```

Otra recomendación es utilizar time.sleep(), esta función nativa de Python sirve para detener la ejecución del código por cierto tiempo, ya que normalmente Selenium realiza las acciones muy rápido, en caso de que el elemento no sea accesible o no se esté haciendo click intente agregar un time.sleep(), en especial en casos en los que se utiliza scrollIntoView, para que le de tiempo al navegador de ubicar el elemento en pantalla antes de hacer click, ingresar texto, etc.

Finalmente, Selenium también permite ejecutar código JavaScript directamente con `execute_script()`. Esto es útil para realizar acciones que no están disponibles nativamente en Selenium o para manipular el DOM directamente. Por ejemplo, puedes simular clics complejos, ajustar valores de estilo o ejecutar funciones específicas de la página.