Supplementary Material

OmniACT: A Dataset and Benchmark for Enabling Multimodal Generalist Autonomous Agents for Desktop and Web

Raghav Kapoor[1]*, Yash Parag Butala[1]*, Melisa Russak[2], Jing Yu Koh[1], Kiran Kamble[2], Waseem AlShikh[2], and Ruslan Salakhutdinov[1]

[1] Carnegie Mellon University
[2] Writer.com
{raghavka, ypb}@cs.cmu.edu
https://huggingface.co/datasets/Writer/omniact

In this paper, we present a novel dataset OmniACT that aids in building more robust multimodal generalist autonomous agents for Desktop and Web. Along with this, we also propose a new continuous scale metric that allows better assessment of actions on a computer screen and the DetACT module which we integrate into multiple LLMs and (Vision Language Models) VLMs that can extract the useful features from the screen image and help us benchmark the dataset. We present the following items that give further insight into the dataset and experiments we performed:

1 List of applications and websites

The complete list of applications and websites chosen for the dataset are listed in table 1. We ensure that the data is equitably represented across all domains and operating systems. We choose desktop applications that are more commonly used for each of the operating systems. The commonly used applications we select are representative of eclectic demographics. We also select a few third-party applications that users commonly use on all operating systems. To avoid redundancy and ensure diversity in our tasks, we refrain from choosing the same set of applications for all three operating systems. For websites, we select our candidate set based on user intent. This is sub-categorized into six domains - (1) Shopping, (2) Entertainment, (3) Service, (4) Government, (5) Travel, and (6) Health.

Table 1: List of applications for desktop and web in OmniACT

Desktop					Websites		
MacOS			Linux	Windows			
Mail	Music	App Store	Audible	Grammarly	AMC	Coursera	NPS
Maps	News	Calculator	Camera	Outlook	Apartments	FlightAware	NY Gov
Message	Photos	Calendar	Desktop	Spotify	Aramex	Google Careers	Samsung
Prime	Preview	Clock	Files	WeChat	Armani	HealthLine	Trip Advisor
Terminal	Reminder	Expedia	Settings	Calendar	Asics	Hertz	UHaul
Weather	Stocks	Finder	Todo	Files	AT&T	Indeed	Udemy
Zoom	System Preferences	iBooks	Text Editor	Settings	BestBuy	Instacart	United Airlines
Desktop			VLC	Windows Store	Booking	Mayo Clinic	UPS
					Cmu.edu	Mint Mobile	Walmart

2 Additional Qualitative Results

2.1 Screen Region Attention

We perform an in-depth analysis of our dataset to see where the coordinate predictions are gravitated. We find interesting insights about the parts of the screen that are attended towards. Most of the predictions for the desktop data are focused either on the center of the screen or the bottom corners. While most of the web interactions are pointing toward the bottom half of the screen, especially concentrating toward the corners. This is depicted in figure 1. This presents some exciting directions for training multimodal models that will heavily rely on the screen image in addition to the elements extracted from DetACT module.

From the histograms at the top and right of the screens in figure 1, we notice that the dataset is not heavily biased towards any particular region of the screen. This implies that models that are likely to attend to only certain regions of the screen or predict the coordinates in the same region of the screen will not have a good performance over the dataset.

Table 2: Performance of DetACT Module with GPT-4 Series of Models

Model	Sequence Score	Action Score
GPT-4 (w/o DetACT)	31.87	0.32
GPT-4 + DetACT	32.75	11.60

2.2 Impact of DetACT Module

In this set of experiments we present the results without using our DetACT module and their comparison with the corresponding model with DetACT outputs. We can clearly observe in table 2 that even models like GPT-4 perform

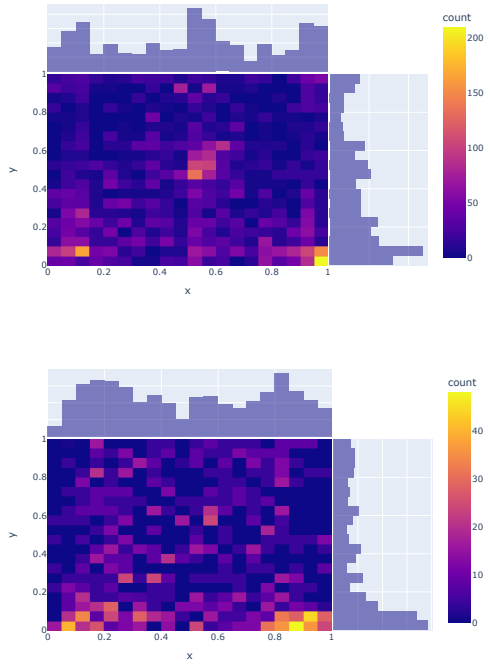

Fig. 1: Screen region attention in output scripts for desktop and web data respectively

close to zero when not prompted by DetACT as they are only language-based. This highlights the importance of well-equipped screen parsing modules such as our DetACT framework that effectively extract multimodal information in the form of text, which is an essential cue for every model.

2.3 Sequence length versus Performance

We analyze the performance of predicting the sequence of actions as a function of sequence length. Specifically, we want to check whether it becomes difficult for models to predict the sequence correctly when the target sequence is longer. Figure 2 shows the accuracy of sequence prediction as a function of sequence length for GPT-4 on the test set. Only when the entire sequence of actions predicted by the model is the same as the gold sequence of actions, do we give the score as 1. We report the accuracy percentage in the table. From table 2, we observe that the model performance is better for shorter sequences and gets worse as the sequence length increases.

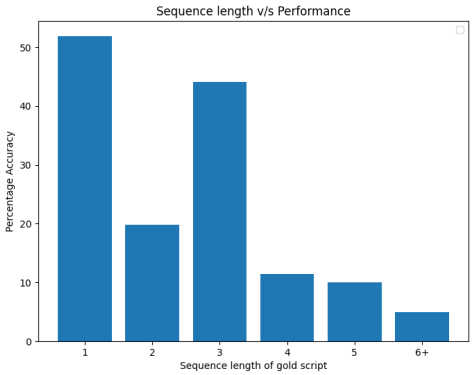

Fig. 2: Sequence prediction accuracy of GPT4 on the test set. The performance decreases as the sequence length increases.

3 Annotation Process

3.1 Creating the Bounding Boxes

To create the bounding boxes, we use two different techniques for desktop and web data. For desktop data, we use a custom PyQt5 tool that helps us create the bounding boxes using a click-and-drag mechanism. This enhances the bounding box creation process, making it more intuitive and user-friendly. It also enables us to encompass interactive regions of UI elements. For instance, when clicking on a search bar, users can input text, while clicking on a search bar icon (such as a magnifying glass) does not activate the typing area. Next, for the website, we use the DOM from HTML and draw bounding boxes using JavaScript. Both these processes ensure a high degree of accuracy and are done by the authors of the paper. Samples for each of them are shown in figure 3. After the bounding boxes are established, the authors exchange their work and review the boxes created by each other.

3.2 Annotator Guidelines

For labeling the bounding boxes for both desktop applications and websites, we leverage MTurk. This is depicted in figure 4. MTurk workers are present with a single bounding box on the screen removing the other boxes and are then asked to label the box based on the functionality of the UI element in context to the screen. They are asked to come up with less than a 5-word description or label for each box. Every box is doubly annotated keeping the more appropriate label during the final filtering. Every MTurk worker is paid at an hourly rate of $20, which is well above state's minimum wage rate.

Next for formulating the tasks, we recruit student volunteers who have basic knowledge of Python and coding. We train every student volunteer about the

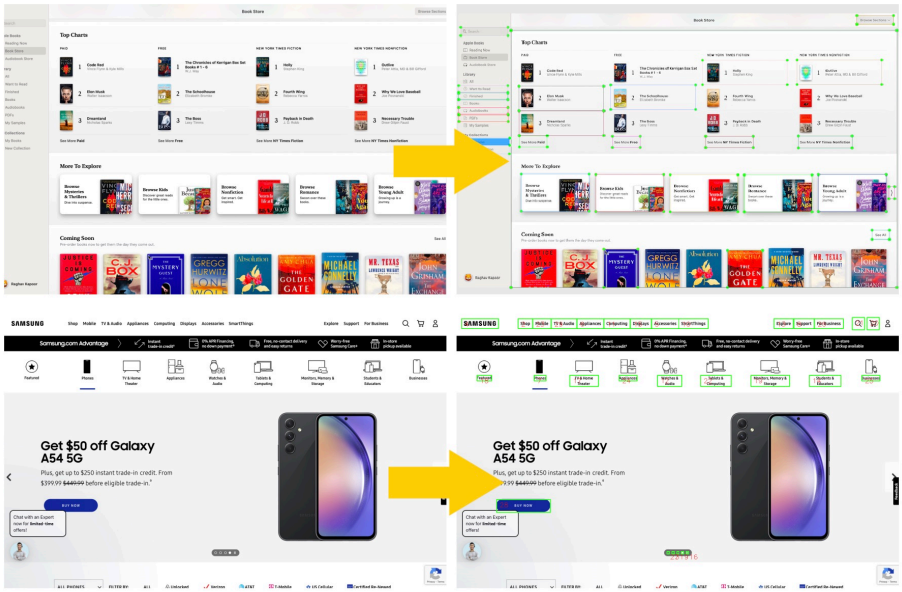

Fig. 3: Annotation Process for creating bounding boxes for desktop applications and websites.

PyAutoGUI usage in a thirty-minute session before starting the task formulation process. Every student is then handed over a sample of tasks that can be created using that screen and is asked to come up with more tasks along with their scripts, which are only a few lines. Every volunteer is asked to come up with as many tasks as possible in one hour along with its linguistic variations to ensure diversity in the language of the tasks. Every volunteer is paid an hourly wage rate of $25, which again surpasses the state's minimum rate.

We build scripts to take in the tasks generated by the students and run over the screens to ensure the tasks are executable. Once done, the authors perform a final pass to keep only the high-quality tasks.

3.3 Human Feedback and Evaluations

To evaluate the human baselines, we again recruit student volunteers. This time the volunteers are not given the reference output script and only a pair of an image and the task description. The students are asked to then come up with an automation script using a few samples as references. The volunteers use their experience and intuition to comprehend the screen and the functionalities and write scripts based on that. Here the student worker is compensated at an hourly rate of $25, well exceeding the state's minimum wage.

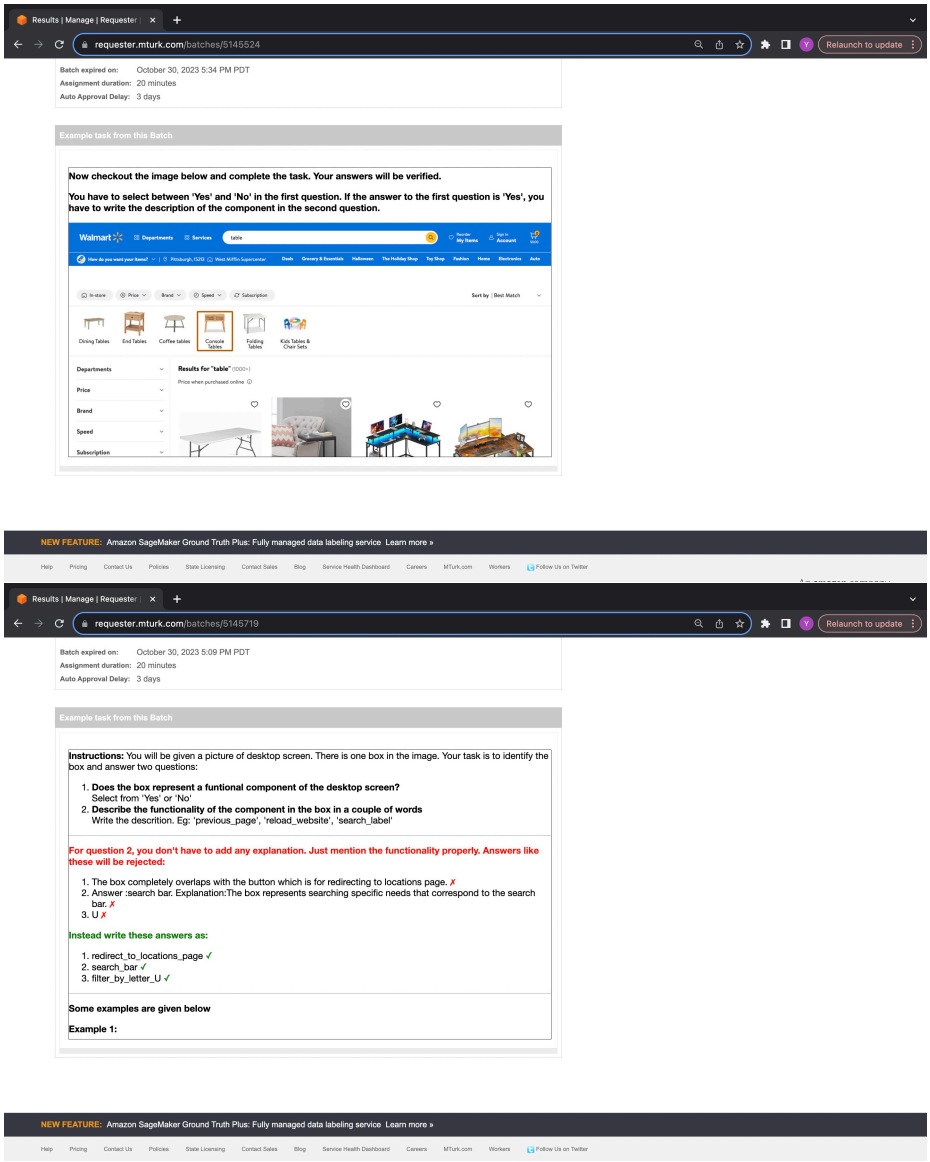

Fig. 4: Screenshots from MTurk Portal that depict the labeling process for all boxes.

Table 3: Hyperparameters used for each of the models for DetACT filtering and Baselines.

	Model	Hyperparameters
DetACT Filtering	GPT-4	temperature = 0.1
LLM Prompt Only Baselines	LLaMA-7B Vicuna-7B	max_new_tokens = 400, temperature = 0.8, repetition_penalty = 1.1
	LLaMA-13B Vicuna-13B	max_new_tokens = 400, temperature = 0.8
	Palmyra-Instruct 30 B	temperature = 0.7
	CodeLLaMA-34B	max_new_tokens = 400, temperature = 0.7
	Palmyra-X 43B	temperature = 0.7
	GPT-3.5-turbo GPT-4	temperature = 0.3
LLM Fine-tuned baselines	LLaMA-13B FT Vicuna-13B FT	LoRA rank = 64, scaling factor = 16, LoRA dropout = 0.05, batch size = 4, learning rate = 5e-5
Multimodal Model Prompt Only Baselines	LLaVA-v1.5-7B LLaVA-v1.5-13B GPT-4V Gemini-Pro	temperature = 0.2, max_new_tokens = 512

4 Dataset and Metadata Format

The dataset consists of three files, one for each split. Every file has a list of task numbers for that particular split. Every task directory consists of (1) *image.png* file, which is a screenshot of the screen, and (2) *task.txt* file which contains the task description and the ground truth output script. We also include another file, (3) *box.json*, which is part of the metadata and is meant to be only used for running the evaluations that require the coordinates of the boxes and not for performing the task or aiding the model in any way.

5 Parameters for Model Training

We enlist the model specifications for models we run for DetACT filtering and baselines using DetACT in table 3.

6 Broader Impact

The broader impact of our intricately curated dataset is multifaceted. Firstly, our dataset holds the potential to empower the technologically illiterate, providing them with a more automated and accessible way to navigate and operate computers.

Additionally, our dataset opens avenues for groundbreaking UI grounding research, enabling a deeper understanding of user interactions at the operating

system level. This, in turn, contributes to advancing the field of artificial intelligence (AI) by expanding the scope of AI to encompass OS-level interactions. The dataset's rich content further facilitates the enhancement of large language models (LLMs), broadening their skill sets and improving their capabilities in comprehending and responding to diverse user inputs.

This dataset serves as a foundation for the development of multimodal agents capable of operating seamlessly across various screens and executing tasks with long-term horizons. This versatility in application positions the dataset as a valuable resource for the creation of innovative AI-driven solutions that transcend traditional limitations.

An additional and impactful use case involves building assistive tools for handicapped individuals who may face challenges in operating computer systems or performing routine tasks that involve human-computer interaction. By tailoring technologies to meet the needs of individuals with varying abilities, our dataset contributes to the development of assistive technologies that enhance accessibility and improve the quality of life for a diverse range of users.

In summary, the broader impact of our meticulously curated dataset manifests in its potential to drive advancements in technology accessibility, UI grounding research, AI capabilities, and assistive tools, ultimately fostering a more inclusive and empowered technological landscape.

7 Sample of task execution

We present figures 5 containing screenshots and corresponding output scripts for a selected task. In figure 5, the task is taken from a MacOS native application, *Books*. Here we have a book chapter opened along with the task description to highlight the last line in green color and scroll to the next page. Followed by the task are descriptive screenshots representing every action in the sequence along with its output.

8 Prompts for DetACT and Baselines and Sample Responses

In the following pages, we present the full length prompts that we use for the following tasks:

- Listing 1: Filtering UI elements in DetACT Module
- Listing 2: Running baselines models using DetACT

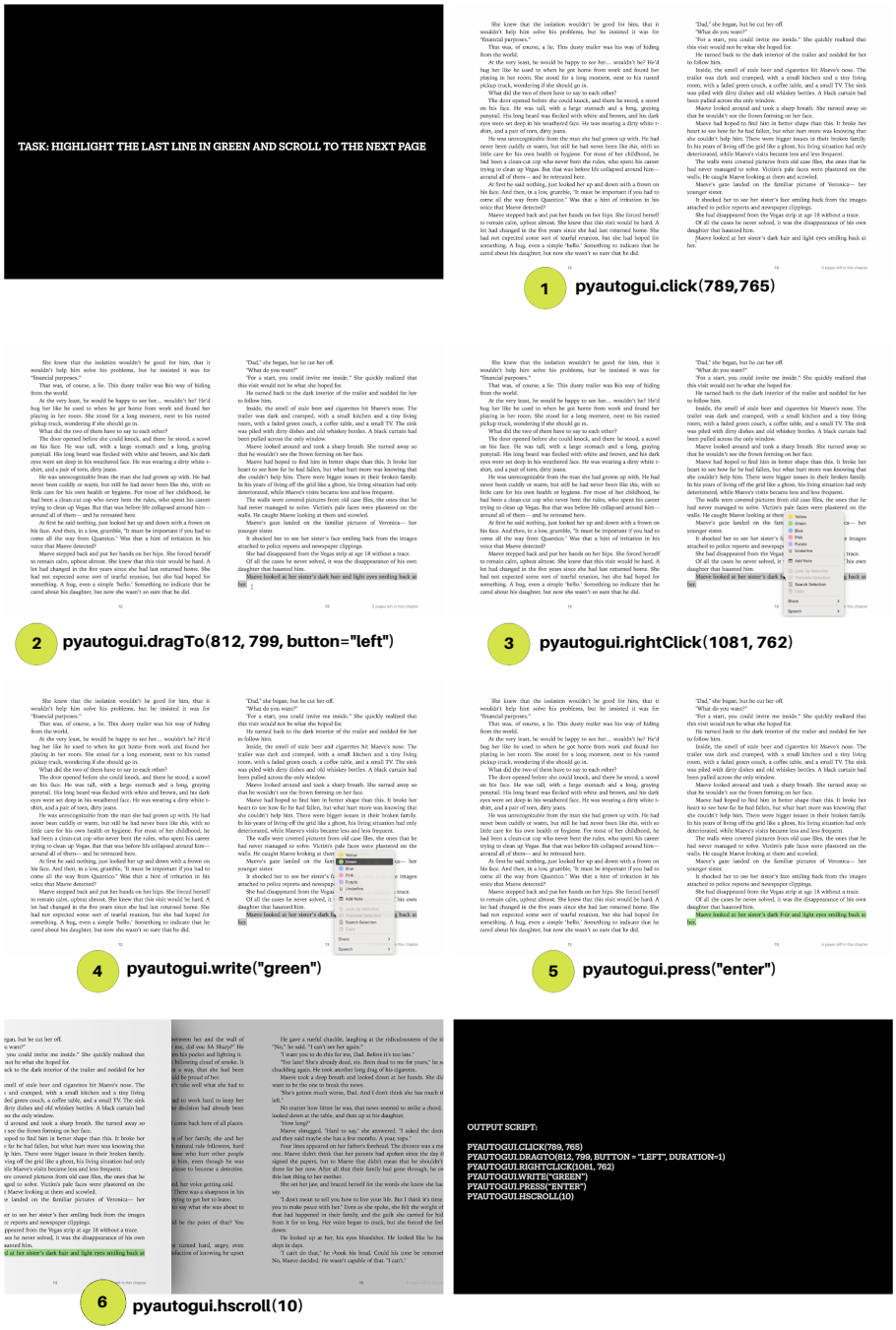

Fig. 5: Task Demo along with intermediate steps and corresponding scripts for every action sequence.

Listing 1.1: Prompt for Filtering UI elements in DetACT Module

```
Given below are the UI elements extracted from the screen. You have to
    filter the elements for this UI screen that are relevant for
    carrying out the task given below.
Make sure to filter the UI elements that may be helpful to carry out the
    task and mention the element description and the corresponding
    coordinates for the task.
Format for every element would be in the form of (Element Label,
    X-Coordinate, Y-Coordinate).

[IMPORTANT!!] Only remove the elements if you are sure that they will
    not help the task.
[IMPORTANT!!] Follow the output structure strictly as given in the below
    example and output nothing else other than the required output.

Sample Task:
{SAMPLE_TASK}

Sample UI Elements:
{LIST_OF_SAMPLE_UI_ELEMENTS}

Sample Filtered UI Elements:
{LIST_OF_SAMPLE_FILTERED_UI_ELEMENTS}


Given Task:
{TASK}

Given UI Elements:
{LIST_OF_UI_ELEMENTS}
```

Listing 1.2: Prompt for baselines models using DetACT outputs

```
You are an excellent robotic process automation agent who needs to
    generate a pyautogui script for the tasks given to you. You will be
    given UI elements on the screen that you can interact with as a list
    of tuples, where each tuple will be of the form [UI Text, X
    coordinate, Y coordinate]. You will also be given an example to help
    with the format of the script that needs to be generated.

[IMPORTANT!!] Stick to the format of the Output scripts in the example
[IMPORTANT!!] Use only the functions from API docs
[IMPORTANT!!] Follow the Output format strictly. Only write the script
    and nothing else.

Here is the API reference for generating script:

def click(x=moveToX, y=moveToY):
    """takes the mouse to location (moveToX, moveToY) and does a left
        click
    Example:
    High Level Goal: Click at co-ordinate (150, 230).
    Python script:
    import pyautogui
    pyautogui.click(150, 230)
    """
    pass

def rightClick(x=moveToX, y=moveToY):
    """takes the mouse to location (moveToX, moveToY) and does a right
        click
    Example:
    High Level Goal: Right click at co-ordiante (350, 680).
    Python script:
    import pyautogui
    pyautogui.rightClick(350, 680)
    """
    pass

def doubleClick(x=moveToX, y=moveToY):
    """takes the mouse to location (moveToX, moveToY) and does a right
        click
    Example:
    High Level Goal: Right click at co-ordiante (350, 680).
    Python script:
    import pyautogui
    pyautogui.rightClick(350, 680)
    """
    pass

def scroll(clicks=amount_to_scroll):
```

```
    """scrolls the window that has mouse pointer by float value
        (amount_to_scroll)
    Example:
    High Level Goal: Scroll screen by (30).
    Python script:
    import pyautogui
    pyautogui.scroll(30)
    """
    pass

def hscroll(clicks=amount_to_scroll):
    """scrolls the window that has mouse pointer horizontally by float
        value (amount_to_scroll)
    Example:
    High Level Goal: Scroll screen horizontally by (30).
    Python script:
    import pyautogui
    pyautogui.hscroll(30)
    """
    pass

def dragTo(x=moveToX, y=moveToY, button=holdButton):
    """drags the mouse to (moveToX, moveToY) with (holdButton) pressed.
        holdButton can be 'left', 'middle' or 'right'.
    Example:
    High Level Goal: drag the screen from current position to (450, 600)
        with left click of the mouse.
    Python script:
    import pyautogui
    pyautogui.dragTo(450, 600, 'left')
    """
    pass

def pyautogui.moveTo(x=moveToX, y=moveToY)
    """take the mouse pointer to (moveToX, moveToY)
    Example:
    High Level Goal: hover the mouse pointer to (450, 600).
    Python script:
    import pyautogui
    pyautogui.moveTo(450, 600)
    """
    pass

def write(str=stringType, interval=secs_between_keys):
    """writes the string wherever keyboard cursor is at the function
        calling time with (secs_between_keys) seconds between characters
    Example:
    High Level Goal: Write "Hello world" with 0.1 seconds rate
    Python script:
    import pyautogui
```

```
    pyautogui.write("Hello world", 0.1)
    """
    pass

def press(str=string_to_type):
    """simulate pressing a key down and then releasing it up. Sample
        keys include 'enter', 'shift', arrow keys, 'f1'.
    Example:
    High Level Goal: Press enter key now
    Python script:
    import pyautogui
    pyautogui.press("enter")
    """
    pass

def hotkey(*args = list_of_hotkey):
    """Keyboard hotkeys like Ctrl-S or Ctrl-Shift-1 can be done by
        passing a list of key names to hotkey(). Multiple keys can be
        pressed together with a hotkey.
    Example:
    High Level Goal: Use Ctrl and V to paste from clipboard
    Python script:
    import pyautogui
    pyautogui.hotkey("ctrl", "v")
    """
    pass


Here are a few examples for generating the script and format to be
    followed:

Example 1:
{RETREIVED_EXAMPLE_1}

Example 2:
{RETREIVED_EXAMPLE_2}

Example 3:
{RETREIVED_EXAMPLE_3}

Example 4:
{RETREIVED_EXAMPLE_4}

Example 5:
{RETREIVED_EXAMPLE_5}


Here are the Textual Elements found on the screen along with their
    coordinates:
{TEXTUAL_ELEMENTS_FROM_DETACT}
```

```
Here are the Icon/Image Elements found on the screen along with their
    coordinates:
{ICON_ELEMENTS_FROM_DETACT}

Here are the Color Elements found on the screen along with their
    coordinates:
{COLOR_ELEMENTS_FROM_DETACT}

Based on the list of UI elements and the examples above, generate the
    pyautogui script for the following task:

{TASK}
```