
UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMATICA
CURSO EM SISTEMAS DE INFORMAÇÃO

GABRIEL RAMOS RODRIGUES OLIVEIRA

# PYAUTOTK: Bridging Usability and Flexibility in Web Automation Frameworks

Recie
2025

GABRIEL RAMOS RODRIGUES OLIVEIRA

GABRIEL RAMOS RODRIGUES OLIVEIRA

**PYAUTOTK: Bridging Usability and Flexibility in Web Automation Frameworks**

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Centro de Informatica da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Sistemas de Informação.

Aprovado em: 05/04/2025

**BANCA EXAMINADORA**

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Leopoldo Motta Teixeira (Examinador Interno)

Universidade Federal de Pernambuco

_____

# PyAutoTk:
# Bridging Usability and Flexibility in Web Automation Frameworks

**Gabriel Ramos**[1], **Breno Miranda**[1]

[1] Informatics Center – Federal University of Pernambuco (UFPE)
Recife – PE – Brazil

{grro,bafm}@cin.ufpe.br

***Resumo.*** *The accelerating adoption of agile methodologies, DevOps practices, and digital transformation has increased the demand for faster, more efficient, and higher-quality deliverables across various industries [Rajkumar et al. 2016]. Automation has become essential not only for software development and QA teams but also for professionals in marketing, sales, and operations seeking to streamline repetitive tasks [Ahmed et al. 2010]. However, existing tools often present significant challenges in terms of learning curve, initial setup, and accessibility, especially for users with limited technical background [Ijaz and Andlib 2014].*

PyAutoTk *emerges as a Python-based automation framework focused on lowering the learning curve and simplifying the creation of automated scripts. By aligning the concept of widgets with fundamental HTML and CSS principles, it reinforces understanding of how elements work and interact. The framework also integrates with the Page Object Model (POM) architecture, supporting both beginners and QA professionals starting in automation.*

*Its modular design, flexibility, and usability-focused approach address common challenges in test automation. Preliminary validation suggests that* PyAutoTk *offers benefits in terms of usability and maintainability, although further empirical studies are necessary to confirm these results. Current limitations include OS compatibility, lack of Playwright support, and limited advanced action features—prioritized for future development.*

PyAutoTk *is open-source and available at* `https://pyautotk.readthedocs.io/en/dev-icst/`. *For a demonstration of* PyAutoTk *in action, visit:* `https://www.youtube.com/watch?v=PzATPf17maY`.

**Palavras-chave**: Software Testing, Web Automation, Page Object, Python

## 1. Introduction

In today's fast-paced technological landscape, the growing emphasis on agility, continuous delivery, and digital transformation has heightened the demand for faster, high-quality deliverables [Rajkumar et al. 2016, Ahmed et al. 2010]. Automation has become a fundamental tool not only for QA teams but also for professionals in areas like marketing and operations, who rely on it to optimize repetitive workflows such as web scraping, data collection, and task orchestration. However, many existing tools pose barriers to entry due to steep learning curves, complex setup processes, and the need for advanced programming skills [Wiklund and Wiklund 2018].

PyAutoTk was developed to address these challenges by providing an accessible, Python-based framework that simplifies the understanding and implementation of web

automation. Its primary goal is to reduce the learning curve associated with both programming and automation concepts. By combining high-level abstractions—such as widgets and session decorators—with fundamental web development knowledge (HTML and CSS), PyAutoTk allows users to interact with web elements intuitively. In addition, the framework aligns with the Page Object Model (POM), reinforcing modularity and scalability in automation design. This makes it easier for beginners and QA professionals to progressively advance in their understanding of automation concepts and best practices.

Although PyAutoTk introduces a minor trade-off in execution speed compared to direct Selenium usage, it is hypothesized that this is offset by improvements in usability, maintainability, and onboarding speed. Ongoing usability studies aim to evaluate how effectively the framework supports users in developing automation scripts with minimal technical friction. If validated, PyAutoTk may empower a wider range of professionals to adopt automation in their daily workflows, boosting efficiency and productivity.

The remainder of this paper is structured as follows: Section 4 details the tool's architecture and core functionalities, including the widget abstraction and browser session management. Section 5 presents a technical comparison between PyAutoTk and raw Selenium implementations. Section 6 describes the hypothesized user learning journey and pedagogical approach. Section 7 provides motivating examples demonstrating real-world applications of the framework. Section 8 discusses the user feedback methodology and collected responses. Section 9 examines the framework's limitations and performance trade-offs. Finally, Section 10 concludes the paper and outlines directions for future research.

## 2. Conceptual Foundations: Required Background for Learning

This section introduces essential concepts required to understand and effectively use PyAutoTk. It aims to minimize the learning curve by establishing foundational knowledge.

- **HTML and CSS Fundamentals:** HTML (HyperText Markup Language) structures the content of web pages, while CSS (Cascading Style Sheets) defines their visual presentation. Understanding basic HTML tags (e.g., `<div>`, `<button>`, `<input>`) and CSS properties (e.g., `color`, `position`, `display`) is crucial to recognize how elements appear and interact.
- **What is a Widget?:** In UI/UX and web automation, a widget represents any user interface element with which a user can interact. Examples include buttons, scrollbars, input fields, and icons. In the context of PyAutoTk, widgets abstract these components into programmable objects, facilitating interaction.
- **Context and Session in Automation:** In browser automation, a *context* refers to the environment in which automation runs—typically, a browser tab or session. Managing sessions ensures browser resources are initialized and disposed correctly. PyAutoTk uses decorators to handle the session lifecycle automatically.

## 3. Simplifying Automation Through Abstraction

To bridge the gap between technical automation frameworks and educational use, PyAutoTk introduces intuitive abstractions that hide complexity while promoting learning.

- **Widget Abstraction:** The `Widget` class is the central element in PyAutoTk, designed to encapsulate all user interactions with web elements into a single, consistent

interface [Statter and Armoni 2020]. By concentrating all widget types into one object model, users are encouraged to explore the HTML structure (DOM) to identify unique attributes such as `text`, `class_`, or `name`. This approach is based on the hypothesis that engaging directly with element attributes builds conceptual familiarity and strengthens the user's ability to reason about UI behavior.
The example in Listing Code 1 demonstrates how a single line of code is sufficient to locate and interact with a button on the page:

**Listing 1. Widget basic instance example**

```
home_btn =
   Widget(session, class_="nav-link scrollto", text="Home")
home_btn.click()
```

This abstraction allows learners to focus on intent (e.g., clicking, typing) instead of syntax (e.g., XPath, selectors), as shown in the following Listing Code 2 used to fill a form based on named fields:

**Listing 2. Widget instance to fill a form**

```
def fill_contact_form(self, **kwargs):
for key, value in kwargs.items():
    input_field = Widget
       (self.session, class_="form-control", name=key)
    input_field.enter_text(value)
```

- **Session and Context Alignment:** The `@browser_session` decorator defines the runtime environment of scripts. It abstracts away repetitive boilerplate such as initializing and terminating sessions, allowing learners to focus on actions within the browser as shown in the following Listing Code 3. This aligns with the metaphor of opening a tab and interacting with a page.

**Listing 3. Browser Session example**

```
@browser_session("example.html", browser_type="firefox")
def main(session):
    Widget(session, text="Get Started").click()
```

- **Composition for Learning the Page Object Pattern:**
A core goal of PyAutoTk is to help learners, especially aspiring QA professionals, internalize one of the most important concepts in test automation: the Page Object Model (POM). This design pattern organizes code into logical representations of each screen or section of an application, enabling reusability, readability, and scalability.
To support this learning, PyAutoTk encourages users to group related widgets into Python classes that represent sections of a webpage. Each class mimics a "page object" by encapsulating both locators (widgets) and actions (methods) related to that section. This structure not only reflects industry best practices but also simplifies code comprehension for beginners.
The example in the following Listing Code 4 shows how an interface section with a button is declared in a modular way:

**Listing 4. Ilustration of POM with PyAutoTk**

```
class ArshaMainSection:
    def __init__(self, session):
        self.watch_video_btn
            = Widget(session, text="Watch Video")

    def watch_main_video(self):
        self.watch_video_btn.click()
```

As learners structure their scripts this way, they gradually build fluency in the Page Object approach. This progressive exposure helps users transition from high-level abstractions to industry-grade practices, serving as a scaffolded learning path from beginner-friendly code to more advanced automation design principles.

- **Visual Metaphor and Learning Aids:**
  PyAutoTk adopts a screen-based metaphor: the browser is a canvas, and widgets are the components. This aids cognitive mapping between code and UI behavior. As a result, learners develop an internal model of how frontend structures can be manipulated through backend logic. This abstraction supports progressive learning, where users begin with simple tasks and gradually explore more complex logic without being overwhelmed by low-level details.

## 4. Tool

### 4.1. Architecture

The architecture, illustrated in Figure 1, of the PyAutoTk framework is designed with three distinct layers:

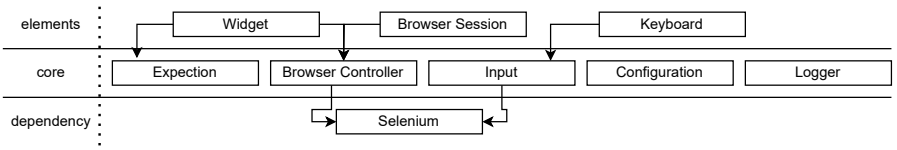


**Figura 1. Overview of PyAutoTk Architecture**

- **Dependency Layer:** This is the foundational layer and currently relies solely on Selenium. It serves as the external dependency, providing the essential capabilities for browser interaction.
- **Core Layer:** The core layer handles critical functionalities such as exception management, logging, configuration settings, as show in Listing Code 5, and the browser controller. Among these, the configuration module is the only interface directly accessible to the user, allowing customization of framework behaviors. These configurations include selecting the default browser, setting browser behaviors like starting in full-screen mode, enabling or disabling the GUI, and toggling logging levels between info and debug. Additionally, the core layer includes support for Input controller, enabling automated simulations of keyboard interactions such as pressing specific keys or navigating through elements using the arrow keys.
- **Elements Layer:** This is the most user-facing layer, where high-level functionalities such as widgets, the browser session decorator, and the Input component are implemented. Widgets simplify interactions with web elements, while the browser session

decorator abstracts session handling. The Keyboard component, similar to the Widget, seamlessly integrates with web elements to provide a unified API for managing external user inputs, such as keyboard and mouse interactions. The elements layer depends solely on the core layer (specifically the browser controller) and maintains complete abstraction from Selenium, ensuring a clear and modular separation of concerns.

The architecture is intentionally designed to prioritize flexibility and adaptability. For instance, the dependency layer could be replaced with a different tool without impacting the other layers. Similarly, the browser controller in the core layer acts as an API, exposing functionalities to the elements layer without revealing implementation details tied to Selenium.

### 4.2. Browser Session

The browser session decorator in PyAutoTk plays a crucial role in simplifying browser initialization and session management. This abstraction removes the need for users to manually configure browsers, handle sessions, or close resources, streamlining the process of starting and completing automation tasks. By leveraging the browser session decorator, users can:

- **Effortlessly Manage Browser Configuration:** Specify browser type (e.g., Chrome or Firefox) and configurations such as headless mode or maximized windows without modifying the core scripts. This practical implementation is illustrated in Figure **??**, which demonstrates how these configurations can be seamlessly applied using the PyAutoTk framework.
- **Streamline Session Management:** Automatically open and close browser sessions, ensuring resources are released appropriately after execution.
- **Eliminate Redundant Code:** Replace repetitive boilerplate setup and teardown code with a single decorator, resulting in cleaner and more concise scripts.
- **Improve Debugging and Logging:** Integrated logging captures browser behavior, aiding debugging and identifying issues during automation runs.

**Listing 5. PyAutoTk Configuration example**

```
from pyautotk.core.config_loader import config

config.browser_type = "firefox"
config.headless_mode = True
config.maximize_browser = True
```

#### 4.2.1. Structured Learning Through Constraints

The session decorator intentionally limits direct control over setup/teardown processes to reinforce foundational automation concepts. By abstracting these steps, PyAutoTk:

- **Reduces Cognitive Overload**: Beginners avoid getting stuck on low-level details (e.g., driver instantiation, implicit waits) and focus on core automation logic.
- **Teaches Best Practices**: Enforces the importance of proper resource cleanup (e.g., closing sessions) without requiring manual implementation.
- **Scaffolds Understanding**: Users implicitly learn the value of setup/teardown phases, preparing them to later customize these steps in tools like Selenium.

For example, automating navigation to a webpage and performing basic actions can be achieved with just a few lines of code using the browser session decorator, enabling users to focus on the logic rather than setup details.

### 4.3. Widget

The concept of widgets in PyAutoTk revolves around encapsulating individual web elements into reusable and manageable objects. A Widget represents a UI component (such as a button, input field, or link) and abstracts the complexities of interacting with these elements. Widgets serve as the building blocks of automation in PyAutoTk, providing an intuitive API for:

- **Clicking Elements:** Perform click actions with built-in error handling and retries.
- **Entering Text:** Automate text input fields seamlessly, including handling element focus and visibility.
- **Scrolling:** Bring elements into view automatically before interacting with them.
- **Extracting Properties:** Retrieve element attributes or data such as text content and dimensions.

The widget abstraction allows users to automate tasks ranging from simple form submissions to advanced interactions across multiple pages. For instance, in the Figure 2, widgets are used to automate instance the ”Get Started”and ”Watch Video”buttons to be able to interact. Each widget is tied to specific UI attributes, making it easy to locate and interact with elements without relying on fragile or complex locators.

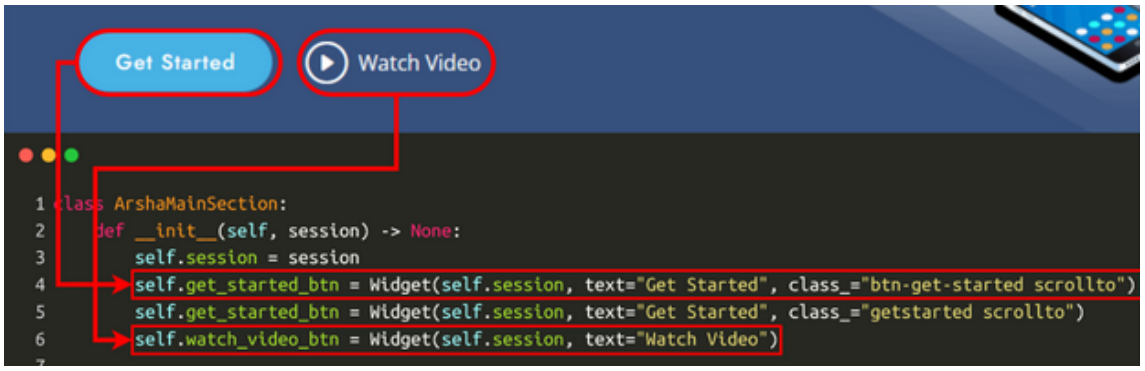


**Figura 2. Widget instance code example**

### 4.4. Keyboard and Input Control

An important aspect of web automation is the simulation of user inputs—especially keyboard events. PyAutoTk includes a dedicated abstraction layer to support common keyboard interactions through the `KeyboardController` class, which wraps Selenium's `ActionChains` to provide a simplified and readable API.

This controller enables users to simulate key presses for both navigation and interaction purposes. The following actions are available and can be triggered through intuitive method calls:

- Navigation keys: `press_arrow_up()`, `press_arrow_down()`, `press_arrow_left()`, `press_arrow_right()`
- Functional keys: `press_enter()`, `press_escape()`, `press_tab()`, `press_space()`, `press_backspace()`, `press_delete()`

- Modifier keys: `press_control()`, `press_shift()`, `press_alt()`
- Function keys: `press_function_key(n)` for any function key from F1 to F12

These input functions abstract the complexity of low-level interactions, allowing learners to simulate realistic workflows such as navigating dropdown menus, submitting forms via Enter, or scrolling through interfaces using arrow keys. The example in the following Listing Code 6 shows a code example.

**Listing 6. Keyboard example use**

```
keyboard = Keyboard(session)
keyboard.press_arrow_down()
keyboard.press_enter()
```

Beyond automation convenience, the `KeyboardController` plays a key pedagogical role. It reinforces the concept of event-driven interactions within web interfaces—something often abstracted away in high-level tools. By requiring users to explicitly simulate actions like pressing keys, PyAutoTk makes invisible processes (e.g., form submission via Enter) tangible.

This promotes an experiential learning process where users begin to understand how browser interactions are structured and triggered. It also helps in debugging real-world scenarios where keyboard input is essential, such as closing popups, navigating modals, or interacting with single-page applications (SPAs). Although still under development, the framework reserves a placeholder for mouse actions through the `MouseController` class, where additional input modalities (e.g., drag-and-drop, hover with delay, or click-and-hold) may be introduced as learners evolve to more complex automation use cases.

### 4.5. Page Object Model as a Learning Composition

The Page Object Model (POM) is one of the most important design patterns for test automation, helping to improve modularity, scalability, and maintainability. In PyAutoTk, POM is not introduced as a pedagogical resource designed to facilitate structured learning for aspiring QA professionals.

**Listing 7. Top Menu POM example using PyAutoTk**

```
class Arsha:
    def __init__(self, session) -> None:
        self.session = session
        self.home_btn = Widget(self
            .session, class_="nav-link scrollto", text="Home")
        self.about_btn = Widget(self
            .session, class_="nav-link scrollto", text="About")
        self.service_btn = Widget(self.
            session, class_="nav-link scrollto", text="Services")
        self.portfolio_btn = Widget(self.session
            , class_="nav-link scrollto", text="Portfolio")
        self.team_btn = Widget(self
            .session, class_="nav-link scrollto", text="Team")
        self.dropdown_btn
             = Widget(self.session, class_="bi bi-chevron-down")
```

```
        self.contact_btn = Widget(self
            .session, class_="nav-link scrollto", text="Contact")
        self.get_started_btn = Widget(self.session, text
            ="Get Started", class_="btn-get-started scrollto")

    def dropbox_navigate(self):
        self.dropdown_btn.click()
        self.dropdown_btn.hover()
```

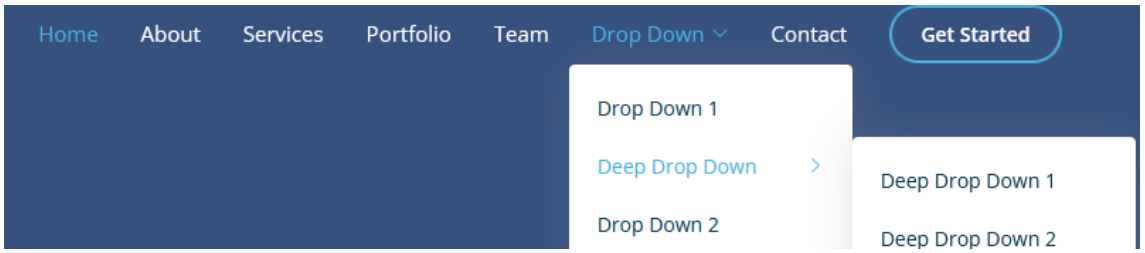


**Figura 3. Top menu button web page example**

By combining abstraction with practical composition, PyAutoTk allows users to build intuitive mental models of automation architecture. The Arsha class (Listing Code 7) demonstrates how to structure a webpage's navigation menu using widgets. Each button—such as `Home`, `About`, and `Contact`—is abstracted as a widget, using identifiable HTML attributes. These are grouped into a class representing the page section, reflecting the POM philosophy of separating locators and actions. The corresponding interface (Figure 3) provides visual reinforcement, linking code to actual UI elements. The method `dropbox_navigate()`, for instance, encapsulates interaction logic into a reusable function. This kind of structure helps beginners grasp the modularity principle behind POM without needing to learn it formally first. It becomes a natural consequence of organizing automation scripts in a readable and reusable way.

### 4.6. Testing of PyAutoTk

To ensure reliability and maintainability during the development of PyAutoTk, a comprehensive suite of unit tests was created. These tests focus on validating critical components, such as the widget and browser functionalities, which are central to the framework's usability.

The widget unit tests ensure that XPath expressions, which are essential for the widget class to interact with web elements, are generated correctly. Similarly, browser unit tests validate the functionality of the browser session decorator for all supported browsers, including tests in headless mode.

An example of browser unit tests, including a headless Chrome session, is shown in Listing Code 8. For additional tests suite available in PyAutoTk, please visit: `https://github.com/brailog/PyAutoTk/tree/dev-icst/pyautotk/tests`.

**Listing 8. Browser Session unit test exmaple**

```
@browser_session(MOCKUP_TEST_URL_FILE)
def test_chrome_browser_headless(self):
    @browser_session(GOOGLE_URL, headless=True)
    def _test_chrome_browser_headless(session):
```

```
        assert session.driver.title == "Google"
    _test_chrome_browser_headless()
```

### 4.7. Exception Handling and Educational Value

A robust exception handling mechanism is critical in any automation framework—not only to ensure reliable execution, but also to guide users through meaningful feedback when things go wrong. In PyAutoTk, exception handling has been designed not only with stability in mind, but also as a didactic resource that encourages deeper understanding of automation flow and failure diagnosis.

All exceptions in PyAutoTk related to widgets and browser behavior inherit from a unified base class: `WidgetException`. This structured hierarchy enables both precise error filtering during execution and improved traceability when debugging complex automation scenarios.

For example, when a user attempts to click on a web element and the action fails, a specific `WidgetClickException` is raised. This exception includes the XPath of the failed element and the original error message, promoting transparency and helping learners trace back to the cause of the issue showing in Listing Code 9

**Listing 9. Handler issues with execption**

```
@browser_session(MOCKUP_TEST_URL_FILE)
def test_navigate_by_search(...):
...
try:
    search_field.click()
except WidgetClickException as e:
    assert False, "Search field did not appear"
```

Each exception is contextually tailored to a specific action or operation within the framework:

- `WidgetClickException`, `WidgetDoubleClickException`, and `WidgetHoverException` address interaction failures.
- `WidgetEnterTextException` and `WidgetScrollException` help identify input and visibility issues.
- `WidgetWaitTimeoutException` ensures that users understand when a timeout condition has occurred.
- `WidgetAttributeException` and `WidgetRetrievePropertiesException` assist in debugging attribute access and data retrieval.
- `BrowserWaitForPageLoadException` is used when a page fails to load fully within a set timeframe.

From an educational perspective, this granularity fosters a culture of hypothesis testing. Rather than presenting a generic error message, PyAutoTk encourages learners to consider why a specific interaction may have failed—whether due to visibility issues, timing delays, incorrect locators, or DOM structure changes. By analyzing exception types and messages, users incrementally build mental models of browser behavior and error recovery.

Additionally, this exception system aligns with the framework's filtering capability. Developers can catch specific exception types when implementing retry logic or conditional handling in their automation scripts. This provides a foundation for more advanced error management patterns such as fallback mechanisms, adaptive waits, or screenshot captures on failure—gradually introducing professional practices into the learning process.

The exception module in PyAutoTk is not merely a defensive coding feature is an integral part of its pedagogical design. It invites exploration, encourages experimentation, and reinforces the critical thinking required to evolve from a beginner to a proficient automation practitioner.

## 5. Technical Depth of PyAutoTk's Abstraction Layer

To demonstrate how PyAutoTk simplifies Selenium workflows while retaining pedagogical transparency, we analyze a side-by-side comparison of **PyAutoTk** (Listing Code 10) and **raw Selenium** (Listing Code 11) code for filling a contact form. This comparison highlights the framework's abstraction mechanisms and their impact on code simplicity and learning curve.

### Code Comparison

**Objective**: Fill a contact form with user-provided data, measure execution time and memory usage.

**Listing 10. PyAutoTk Performance comparation**

```
@browser_session(f"file:///{MOCKUP_TEST_URL_FILE}",
                 browser_type="firefox",
                 headless=True)
def pyauto_fill_contact_form(session, **kwargs):
    start_mem = get_memory_usage()
    start_time = time.time()

    # Click "Contact" using Widget abstraction
    Widget(session, class_="nav-link scrollto",
           text="Contact").click()

    # Dynamically fill form fields
    for key, value in kwargs.items():
        form_control = Widget(session,
                              class_="form-control",
                              name=key)
        form_control.enter_text(value)

    # Measure metrics
    elapsed_time = time.time() - start_time
    end_mem = get_memory_usage()
    return elapsed_time, end_mem - start_mem
```

**Listing 11. Raw Selenium Implementation**

```
def fill_contact_form(name, email, subject, message):
    start_mem = get_memory_usage()
```

```
    driver = webdriver.Firefox(service=FirefoxService())
    driver.get(f"file:///{MOCKUP_TEST_URL_FILE}")
    start_time = time.time()

    # Manual XPath for "Contact" link
    contact = driver.find_element(
        By.XPATH, "//
           a[contains(@class, 'nav-link') and text()='Contact']"
    )
    contact.click()

    # Scroll and locate fields
    form_element = driver.find_element(
        By.XPATH, '//form[@class="php-email-form"]'
    )
    driver.execute_script
       ("arguments[0].scrollIntoView();", form_element)

    # Explicit field interactions
    name_field = driver.find_element(By.ID, "name")
    name_field.send_keys(name)
    email_field = driver.find_element(By.ID, "email")
    email_field.send_keys(email)
    subject_field = driver.find_element(By.ID, "subject")
    message_field = driver.find_element(By.TAG_NAME, "textarea")
    subject_field.send_keys(subject)
    message_field.send_keys(message)

    # Cleanup
    elapsed_time = time.time() - start_time
    end_mem = get_memory_usage()
    driver.quit()
    return elapsed_time, end_mem - start_mem
```

**Key Simplifications in PyAutoTk**

1. **Session Management**: Abstracts browser initialization/cleanup via `@browser_session`, eliminating manual `driver.quit()` calls.
2. **Widget Abstraction**: Replaces verbose locators (e.g., `By.XPATH`) with declarative queries (`class_`, `text`).
3. **Dynamic Handling**: Loops over `kwargs` to map form fields dynamically, avoiding repetitive code.
4. **Implicit Scrolling**: Automatically scrolls elements into view without explicit JavaScript calls.

**Pedagogical Trade-offs**

- **Lower Barriers**: Beginners focus on *what* to automate rather than *how*.
- **Graudal Complexity**: Users can later inspect PyAutoTk's source code to see Selenium mappings.

**Tabela 1. Summary Comparison of PyAutoTk and Selenium**

| Aspect | PyAutoTk | Selenium |
|---|---|---|
| Lines of Code | 12 | 21 |
| Boilerplate | Minimal (decorators/widgets) | High (manual setup) |
| Cognitive Load | Low (focus on intent) | High (syntax-heavy) |
| Flexibility | Limited to abstractions | Full low-level control |
| Performance | 0.8s (with overhead) | 0.3s (raw speed) |

- **Debugging Pedagogy**: Exceptions like `WidgetClickException` provide actionable feedback.

## 6. The User Journey: Hypothesis in Practice

This section presents the learning journey anticipated by the framework's design, which also forms the basis for validating its core hypothesis.

The central hypothesis is that by abstracting automation into widget-based interactions and providing high-level, readable code structures, non-technical users can learn web automation faster and more intuitively. Instead of being overwhelmed by verbose selectors and configuration steps, users can begin with concrete actions—such as clicking a button or filling out a form—without needing to understand the underlying DOM interaction logic.

The expected learning curve begins with structured documentation and simple examples. Users start by running pre-written code snippets and gradually move toward adapting them for their own use cases. Over time, they gain confidence to modify attributes, combine widget interactions, and build more complete workflows.

To validate this learning path, users will be invited to complete short tasks: executing example scripts, navigating browser sessions, and modifying widget properties to achieve specific automation goals. Feedback will be collected both qualitatively (through interviews and open-ended feedback) and quantitatively (through task success rates and time to completion).

One of the key differentiators of PyAutoTk is its commitment to transparency. As an open-source project, all internal components are fully accessible. Users are not only encouraged to use the framework but also to read its source code, understand how widgets are implemented, and explore how sessions are managed. This visibility turns the framework itself into a pedagogical tool, helping learners form mental models of how automation frameworks work internally.

As users evolve in their understanding, they can progressively remove these scaffolds. Since PyAutoTk is built atop Selenium and modular by design, more advanced learners can begin replacing or extending components, or even migrate completely to tools like Selenium or Playwright. This makes PyAutoTk not only a learning resource but also a valid intermediate framework capable of supporting real-world automation needs before transitioning to lower-level solutions.

The design of PyAutoTk is grounded in well-established learning theories (Figure 4). "Learning by Doing," popularized by John Dewey, suggests that learners grasp concepts more effectively when engaging in hands-on experiences. PyAutoTk embraces this principle by encouraging experimentation and immediate feedback—users run real scripts, observe

outcomes, and iteratively build understanding.

Additionally, the framework aligns with the theory of Scaffolding, introduced by Jerome Bruner and based on Vygotsky's educational model. In this approach, learners are supported early on by simplified constructs (e.g., decorators and widgets) that reduce cognitive load. As users gain experience, these supports can be removed, empowering them to work directly with more complex tools and paradigms.

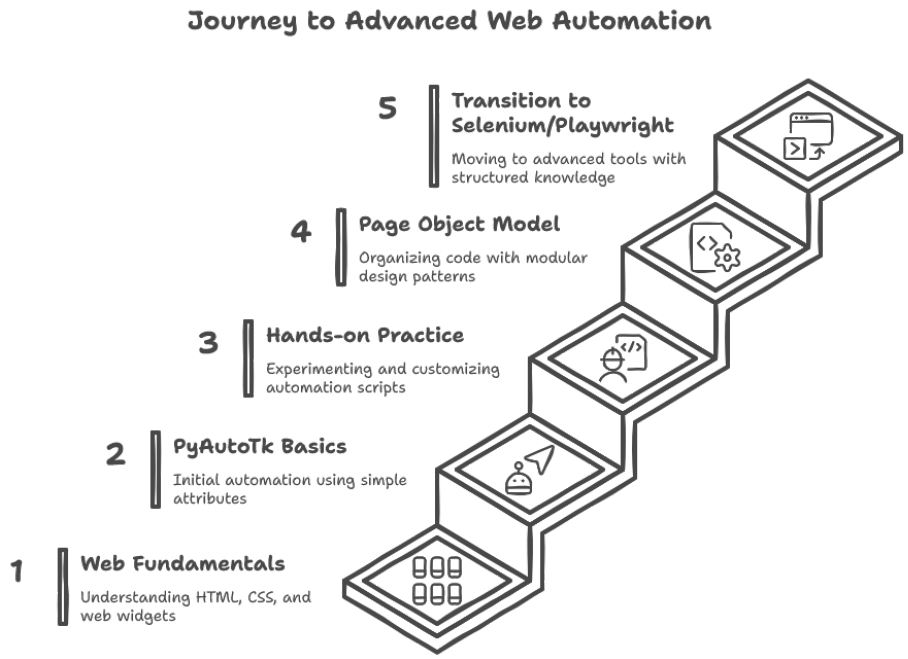


**Figura 4. User Journey in Practice**

## 7. Motivating Example

During the development of PyAutoTk, real-world examples were created not only to validate its architecture but also to demonstrate its educational value through practical use cases. These examples serve a dual purpose: they showcase how automation can be structured using intuitive abstractions, and they give learners an opportunity to apply their knowledge in real environments before transitioning to more complex tools like Selenium or Playwright.

Listing Code 12 presents a typical scenario involving swipe actions in video interfaces such as TikTok and YouTube Shorts. Despite being simplified for educational purposes, the example simulates real interaction patterns and browser behavior, helping users understand key automation concepts such as navigation, keyboard input, and DOM-based element targeting

Moreover, PyAutoTk goes beyond basic examples by offering mechanisms to deal with dynamic and unpredictable interface elements. Listing Code 13 demonstrates how the framework handles conditional popups using exception handling and timeouts—skills essential in production-grade automation.

These examples reinforce the idea that PyAutoTk can be used as a practical stepping stone. Beginners are empowered to write meaningful automation scripts that work in real

websites, giving them confidence and clarity before advancing to lower-level tools. Rather than relying solely on mock interfaces or theoretical explanations, learners experience firsthand how to diagnose and resolve real-world automation challenges.

Thus, PyAutoTk acts not only as a pedagogical framework but also as a bridge between learning and execution. It reduces the initial friction typically encountered in automation onboarding, while still offering the capability and flexibility required for real application testing and prototyping.

**Listing 12. Automation Example: Swiping Through TikTok and YouTube Shorts Videos**

```
@browser_session("https://www.tiktok.com/")
def watch_tiktok(session):
    generical_swipe_for_small_videos_interface
       (session, "For you")

@browser_session(url="https://www.youtube.com/")
def watch_shorts(session) -> bool:
    generical_swipe_for_small_videos_interface
       (session, "Shorts")

def generical_swipe_for_small_videos_interface
   (session, menu_option: str):
    keyboard = Keyboard(session)
    Widget(session, text=menu_option).click()
    session.wait_for_initial_load()
    for _ in range(SWIPE):
        keyboard.arrow_down()
        time.sleep(0.5)
```

**Listing 13. Handling Popups During TikTok Automation.**

```
def swipe_tiktok_with_skip_guest(session, menu_option: str):
    keyboard = Keyboard(session)
    Widget(session, text=menu_option).click()
    session.wait_for_initial_load()
    for _ in range(SWIPE):
        keyboard.arrow_down()
        try:
            Widget(text="Continue as guest").click(timeout=1)
        except WidgetClickException:
            continue
```

## 8. User Feedback and Survey Design

To evaluate the hypothesis that PyAutoTk facilitates the learning process of web automation through abstraction and pedagogical alignment, we developed and distributed a feedback form to potential users of the framework. The primary objective was to understand how real users—particularly those at the beginning of their careers in QA or automation—would perceive the tool's usefulness in learning and applying core automation concepts.

A total of 17 users participated in this feedback round, including university students, junior developers, and QA professionals with varying degrees of prior experience. The feedback collected was crucial in assessing both the usability of PyAutoTk and its effectiveness as an educational aid. The survey was structured into five main categories and summarized in Table 2. The first category, **Background Information**, aimed to contextualize each respondent's experience level with web automation and tools like Selenium or Playwright. This allowed us to calibrate responses according to prior knowledge and assess the framework's impact on different user profiles. The second category, **Learning Experience**, explored how users interacted with the documentation, code examples, and the overall learning flow. It included questions about the ease of understanding the examples and whether users felt more confident in handling basic concepts like HTML structure and the Document Object Model (DOM) after using the tool. In the third category, **Practical Application**, we investigated whether users were able to run and adapt the example scripts, and if they would consider using PyAutoTk in small, real-world projects. While the main intent of the framework is educational, this section helped assess whether its simplicity and abstraction could extend to lightweight practical use cases. The fourth category, **Transition Readiness**, focused on whether the use of PyAutoTk helped users feel more prepared to transition to core automation libraries like Selenium or Playwright. Questions in this section evaluated the confidence gained and understanding developed through the use of high-level abstractions. Lastly, the fifth category, **Suggestions and Perceived Value**, captured qualitative insights on what could be improved and what was most appreciated. Many users highlighted the clarity of the widget abstraction and the usefulness of the examples but also pointed out the need for more visual learning materials and the inconvenience of having to always define a Python function due to the decorator-based session model.

Through this structured approach, we were able to extract rich feedback that informed both the validation of our hypothesis and directions for future development.

### 8.1. Collected Responses

**Tabela 2. User Feedback Summary (n = 17)**

| Question | Response |
|---|---|
| Experience level in automation | 10/17 Beginner, 5/17 Intermediate, 2/17 Advanced |
| Was the documentation and code example easy to understand? | Average rating: 4.6/5 |
| Did you manage to run and edit the examples without issues? | 15/17 Yes |
| Did the framework help you understand HTML/CSS/DOM better? | 14/17 Yes |
| Would you consider using PyAutoTk for small real-world projects? | 6/17 Yes |
| Would you recommend this framework to someone starting in QA or test automation? | 16/17 Yes |
| After using PyAutoTk, do you feel more prepared to learn Selenium or Playwright? | Average rating: 4.3/5 |
| Which concepts became clearer after using the tool? | Structure of HTML; Element identification; How sessions work; Automation flow |
| Suggestions for improvement | More visual examples; Include GUI; More exercises; Need to always use Python function due to decorator |

### 8.2. Insights from User Feedback

The results reinforce the hypothesis that PyAutoTk is effective in promoting a structured and accessible entry into web automation. Respondents reported a high level of satisfaction, particularly around the simplicity of the abstraction (`Widget`, `@browser_session`) and the alignment with educational challenges such as understanding HTML and CSS in relation to automation logic.

Key takeaways include:

- **Cognitive Scaffolding:** Users felt guided by the framework's abstractions, which simplified the learning process and lowered the entry barrier to automation.
- **Didactic Utility:** Most participants indicated that they would recommend the tool to someone starting in QA or test automation.
- **Bridge to Advanced Tools:** The tool encouraged deeper understanding, enabling learners to explore and transition to Selenium or Playwright with greater autonomy.
- **Areas for Improvement:** Feedback highlighted the constraint of always needing to write automation logic inside Python functions due to the decorator-based session management.

## 9. Limitations and Lessons Learned

The evaluation of PyAutoTk revealed important insights into its trade-offs and limitations, particularly when considering its primary goal: to support learning in web automation rather than to compete as a high-performance, production-grade tool.

Performance benchmarks conducted during identical automation tasks—such as form submission and menu navigation—demonstrated that PyAutoTk introduces a measurable overhead compared to native Selenium usage, with an average increase of approximately 0.5 seconds in execution time, more detail in Table 3. However, this performance cost is considered irrelevant in the context of its educational purpose. The abstraction layer, which slightly impacts execution speed, is also what makes the framework accessible and intuitive for beginners, offering users a gentler learning curve when approaching automation for the first time.

**Tabela 3. Performance Metrics Comparison Between PyAutoTk and Selenium**

| Metric | PyAutoTk | Selenium |
|---|---|---|
| **Average Time (s)** | 0.789 | 0.289 |
| **Std Dev (s)** | 0.030 | 0.011 |
| **Min Time (s)** | 0.746 | 0.277 |
| **Max Time (s)** | 0.836 | 0.312 |

This pedagogical trade-off reflects the framework's intent: PyAutoTk was not designed to optimize for speed, but to simplify the understanding of concepts such as DOM interaction, UI element identification, and script modularization. By prioritizing clarity and abstraction, PyAutoTk lowers the barrier to entry for learners who may not yet be comfortable with complex locator strategies or verbose script setups required in tools like Selenium or Playwright.

Several technical limitations remain that could be addressed in future development:

First, PyAutoTk currently supports only Windows 10/11 and Ubuntu (16.04 or newer), with no official compatibility with macOS or distributions like Fedora. This narrows the accessibility of the tool in diverse learning environments and limits its reach among broader audiences. Second, the current version of the source code is tightly coupled to Selenium as the underlying browser automation engine. While this suffices for many educational scenarios, extending the implementation to include Playwright as an alternative backend would enrich the learning experience. This would allow users to understand the differences between engines and experiment with cross-tool abstractions while maintaining the same Widget-based logic. Another technical limitation lies in the lack of hierarchical widget instantiation. Although users can target elements based on attributes like text or class, the framework does not currently support contextual scoping within parent containers, which is often necessary for complex DOM structures. Implementing such functionality would help bridge the gap between simplified learning and more advanced, real-world cases. Finally, while basic actions like clicking and text input are well-supported, advanced user interactions such as drag-and-drop or hover delays are not yet available. Though not essential for introductory learning, these features could be relevant as users progress toward intermediate or advanced automation topics.

In terms of feedback, one recurring observation from learners was the friction introduced by the decorator-based session model. Users must always define a function with the

decorator to execute automation flows, which can feel unintuitive or restrictive in exploratory learning scenarios. Future improvements might explore alternative mechanisms that balance session control with user flexibility.

Overall, the design decisions behind PyAutoTk—from its abstractions to its API design—prioritize the teaching of automation principles rather than replacing conventional tools. Its limitations are acknowledged as opportunities for evolution, with future iterations potentially supporting additional engines and capabilities to further enhance its value as a learning platform.

## 10. Conclusion and Future Work

This paper presented PyAutoTk, a Python-based automation framework developed with an educational perspective to facilitate the learning journey of beginners and QA professionals. By abstracting interactions through high-level constructs such as widgets and session decorators, and by aligning its structure with the Page Object Model (POM) architecture [Ozkaya 2023], PyAutoTk enables users to build conceptual clarity while performing real automation tasks. Its modular and open-source design encourages transparency and allows learners not only to use the tool but also to explore and modify its internal components, reinforcing their understanding of web automation principles.

The results of the evaluation and the feedback collected from 17 users—ranging from university students to junior QA engineers—demonstrated that the tool is effective in promoting structured learning and improving confidence in understanding HTML, CSS, and DOM-related concepts. While some limitations were noted, such as the need to always define Python functions due to the decorator-based session model and the lack of support for macOS or advanced browser interactions like drag-and-drop, these constraints were considered minor in light of the tool's primary goal. The framework's performance trade-off, observed in comparison with Selenium, was also deemed acceptable, given the educational value and clarity provided by its abstractions.

Looking ahead, future developments for PyAutoTk include extending compatibility to macOS and other Linux distributions, integrating Playwright as an alternative backend to support comparative learning, and adding support for more advanced user interactions, such as mouse gestures and contextual element scoping. Another direction involves testing the framework in mobile contexts through the use of tools like `uiautomator` for Android automation [Sinaga et al. 2018]. Based on user feedback, there are also plans to implement visual learning aids and possibly integrate LLM-based solutions for automatically generating Page Object Models.

More than a production-ready framework, PyAutoTk positions itself as a scaffold for education, serving as a stepping stone for those who aim to migrate to more complex tools like Selenium or Playwright. Its open-source nature, simplicity, and alignment with foundational learning theories make it a promising alternative to ease the onboarding process for aspiring automation professionals. By continuing to evolve from user contributions and academic experimentation, PyAutoTk seeks to become a gateway for deeper learning and confident progression in the automation landscape.

## Referências

Ahmed, A., Ahmad, S., Ehsan, N., Mirza, E., and Sarwar, S. Z. (2010). Agile software development: Impact on productivity and quality. In *2010 IEEE International Conference on Management of Innovation Technology*, pages 287–291.

Ijaz, T. and Andlib, F. (2014). Impact of usability on non-technical users: Usability testing through websites. In *2014 National Software Engineering Conference*, pages 37–42.

Ozkaya, I. (2023). Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40(3):4–8.

Rajkumar, M., Pole, A. K., Adige, V. S., and Mahanta, P. (2016). Devops culture and its impact on cloud delivery and software development. In *2016 International Conference on Advances in Computing, Communication, Automation (ICACCA) (Spring)*, pages 1–6.

Sinaga, A. M., Wibowo, P. A., Silalahi, A., and Yolanda, N. (2018). Performance of automation testing tools for android applications. In *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 534–539. IEEE.

Statter, D. and Armoni, M. (2020). Teaching abstraction in computer science to 7th grade students. *ACM Trans. Comput. Educ.*, 20(1).

Wiklund, K. and Wiklund, M. (2018). The next level of test automation: What about the users? In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 159–162.