

Type Speed Calculator Project with Python

Arife Zeynep Muratoğlu 64220048

Berre Sümeyye Tandoğan 64210036

Zeynep Çalapkulu 64210019

Nefise Buse Uzun 64220041

Ankara Medipol University

Programming Languages (CPE350) Project Assignment

Title

Exploring and Implementing a Modern Programming Language

Objective

The goal of this project is to gain a deeper understanding of one of the recent or notable programming languages from a list provided and explore its syntax, features, strengths, and weaknesses. Through hands-on implementation, the project will allow students to compare their chosen language with others in terms of performance, usability, and scalability. By the end of the project, students should be able to confidently analyze, implement basic programs, and critically evaluate the chosen programming language's applications in modern software development.

CONTENT

INTRODUCTION	5
1. History and Evolution	6
1.1. Creation of Python	6
1.2. Key Milestones in Python's History	6
1.3. Major Python Versions	7
1.4. Reasons for Python's Success.....	7
2. Core Features.....	8
2.1. Readability.....	8
2.2. Syntax	8
2.3. Paradigms.....	9
2.4. Memory Management.....	11
2.5. Concurrency	11
2.6. Indentation	12
2.7. Comments.....	12
3. Ecosystem and Library in Python.....	13
3.1. Python's Standard Library	13
3.2. Popular Libraries and Frameworks	13
3.2.1. Web Development.....	13
3.2.2. Data Science and Machine Learning.....	13
3.2.3. Automation and Web Scraping.....	14
3.2.4. Scientific Computing	14
3.2.5. Game Development.....	14
3.2.6. DevOps and Scripting.....	14
3.3. Tools and Environments	14
3.4. Community Support and Documentation	14
3.5. Versatility and Applicability.....	15
4. Use Cases and Industry Adoption of Python.....	16
4.1. Web Development	16
4.2. Data Science and Machine Learning	16
4.3. Automation and Scripting.....	17
4.4. Scientific Computing.....	17
4.5. Game Development	17

4.6. DevOps and Cloud Computing.....	18
4.7. Cybersecurity	18
4.8. Education and Training	18
4.9. Strengths in Industry Adoption	18
5. Project Implementation.....	20
5.1. Libraries Used in the Code	20
5.2. Variables in the Code	21
5.3. Explanation of Functions in the Code	22
5.4. UI Components.....	30
6. Comparison and Evaluation of Python and LISP	33
6.1. Syntax: Is it easy to learn and read?.....	33
6.2. Performance: How does the language perform compared to others for similar tasks? 34	
6.3. Community Support: Availability of learning resources, documentation, and active community	34
6.4. Applicability: What type of projects is the language most suitable for? What are its limitations?	34
CONCLUSION	36

INTRODUCTION

Programming languages serve as the backbone of modern software development, each uniquely tailored to address various computational needs. Python, in particular, has emerged as a leading choice for developers due to its simplicity, versatility, and extensive ecosystem. This project explores Python through the development of a "Type Speed Calculator," aiming to showcase its capabilities in building user-friendly and efficient applications.

The "Type Speed Calculator" project leverages Python's strengths, including its clean syntax, robust standard library, and GUI development tools like Tkinter. By implementing a real-world application, the project not only highlights Python's ease of use but also compares its performance and features to another language, LISP, known for its symbolic computation and historical significance in artificial intelligence.

Through this hands-on experience, the project underscores Python's adaptability across diverse domains, from web development and data science to automation and education. By analyzing Python's history, features, and industry adoption, the project demonstrates why it continues to be a critical skill in the ever-evolving landscape of software development.

1. History and Evolution

1.1. Creation of Python

Creator: Guido van Rossum

Year of Creation: Late 1980s, with the first release in February 1991.

Motivation: Guido, working at CWI (Centrum Wiskunde & Informatica) in the Netherlands, wanted to create a general-purpose programming language that was simple, readable, and easy to extend.

Inspiration:

Derived many concepts from the ABC programming language, which was also developed at CWI.

Python was designed to address the limitations of ABC while maintaining its simplicity.

Naming: Named after the British comedy series "Monty Python's Flying Circus".

1.2. Key Milestones in Python's History

- 1991: First Public Release (Python 0.9.0)

Included basic features like exception handling, functions, modules, and core data types such as lists, strings, and dictionaries.

The indentation-based syntax, a hallmark of Python, was established.

- 1994: Python 1.0

First official version with features like lambda functions, map, filter, and reduce. Built to prioritize readability and productivity.

- 2000: Python 2.0

Introduced list comprehensions, garbage collection based on reference counting, and the beginnings of Unicode support. Python 2 became widely adopted but contained design decisions that led to long-term issues. Acknowledging these, Python 3 was planned.

- 2008: Python 3.0

Released as a major, backward-incompatible version to address Python 2's flaws.

Key changes included:

Enhanced Unicode handling.

Print became a function (print() instead of a statement).

Division operator (/ vs. //) clarified integer vs. float division.

Initially adopted slowly due to backward incompatibility, but it became the standard over time.

- 2020: End of Python 2 Support

Python 2 reached its end of life on January 1, 2020.

Developers were encouraged to transition fully to Python 3.

1.3. Major Python Versions

- Python 1.x

Release Period: 1994–2000

Focused on establishing Python as a general-purpose, beginner-friendly language.

- Python 2.x

Release Period: 2000–2020

Introduced features like list comprehensions, improved garbage collection, and partial Unicode support.

However, design inconsistencies led to the need for a cleaner version.

- Python 3.x

Release Period: 2008–Present

A major revamp with focus on:

Modernizing the language (e.g., better Unicode handling).

Simplifying code readability and maintainability.

Active development with frequent minor releases like Python 3.11 and 3.12, which bring performance improvements and new features.

1.4. Reasons for Python's Success

Readability and Simplicity: Indentation-based syntax and minimalist design.

Flexibility: Usage in various fields like web development, data analysis, artificial intelligence, and automation.

Community and Libraries: Strong library support with tools like NumPy, TensorFlow, and Django.

Beginner-Friendly: Easy to learn and has extensive documentation resources.

2. Core Features

2.1. Readability

Python was designed with readability as a priority, aiming to make code intuitive and maintainable. The language incorporates elements that resemble English, coupled with mathematical influences, ensuring that code is approachable even for beginners. This emphasis on simplicity reduces the learning curve significantly.

2.2. Syntax

Python features a clean and straightforward syntax that mirrors the English language, allowing developers to write concise and intuitive programs.

Unlike many other programming languages, Python does not require semicolons or parentheses to end statements, using new lines instead.

Python enforces indentation using whitespace to define the scope of loops, functions, and classes, making code visually structured. Missing or inconsistent indentation leads to syntax errors. The standard indentation in Python is four spaces.

```
if 5 > 2:
    print("Five is greater than two!")
if 5 > 2:
    print("Five is greater than two!")
```

-
- **Strings as Arrays:** In Python, strings are treated as arrays of bytes representing Unicode characters. While there is no character data type, single-character strings can be accessed using square brackets.
 - **Variables:**
 - Python does not require explicit declaration of variable types, enabling dynamic typing.
 - Variables can change type during runtime, allowing flexibility in programming.
 - Naming conventions enforce starting variable names with a letter or an underscore, and they must not include reserved keywords.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0

x = 4         # x is of type int
x = "Sally"   # x is now of type str
print(x)
```



```
x = "John"
# is the same as
x = 'John'
```

2.3. Paradigms

Python supports multiple programming paradigms, including:

- **Procedural Programming:** Suitable for scripting and task automation.

```
python

class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

# Calculator nesnesi oluşturun
calc = Calculator()

# Kullanıcıdan giriş alın
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

# İşlemi seçin
operation = input("Enter operation (+ or -): ")

if operation == "+":
    print("Result:", calc.add(num1, num2))
elif operation == "-":
    print("Result:", calc.subtract(num1, num2))
else:
    print("Invalid operation.")
```

- **Object-Oriented Programming (OOP):** Incorporates classes and objects to organize and structure code.

```
python

class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

# Calculator nesnesi oluştur
calc = Calculator()

# Kullanıcıdan giriş al
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

# İşlemi seç
operation = input("Enter operation (+ or -): ")

if operation == "+":
    print("Result:", calc.add(num1, num2))
elif operation == "-":
    print("Result:", calc.subtract(num1, num2))
else:
    print("Invalid operation.")
```

- **Functional Programming:** Provides support for higher-order functions and lambda expressions.

```
python

# Lambda fonksiyonları ile işlemleri tanımla
operations = {
    "+": lambda a, b: a + b,
    "-": lambda a, b: a - b
}

# Kullanıcıdan giriş al
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

# İşlemi seç
operation = input("Enter operation (+ or -): ")

# Sonucu hesapla
if operation in operations:
    result = operations[operation](num1, num2)
    print("Result:", result)
else:
    print("Invalid operation.")
```

2.4. Memory Management

Python employs automatic memory management, leveraging a garbage collector for efficient memory allocation and deallocation. This mechanism includes:

```
python

x = [1, 2, 3]
x = None # Previous list [1, 2, 3] is now eligible for garbage collection
```

- **Reference Counting:** Keeps track of object references and deallocates memory when no references exist.
- **Cycle Detection:** Handles cyclic references that reference counting alone cannot resolve.

2.5. Concurrency

Python supports concurrent programming to handle tasks efficiently. It provides modules such as:

- **Threading:** Allows the creation of lightweight threads for multitasking. Example:

```
python

import threading

def print_numbers():
    for i in range(5):
        print(i)

t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_numbers)

t1.start()
t2.start()
```

- **Asyncio:** Enables asynchronous programming for cooperative multitasking.

```
python
```

```
import asyncio

async def say_hello():
    print("Hello")
    await asyncio.sleep(1)
    print("World!")

asyncio.run(say_hello())
```

2.6. Indentation

Python's use of indentation goes beyond readability, as it is fundamental to defining blocks of code. Missing or incorrect indentation results in syntax errors. Standard practice recommends using four spaces for indentation.

2.7. Comments

Python supports comments to enhance code readability:

- **Single-line comments:** Use the # symbol.
- **Multi-line comments:** Utilize triple quotes (""" or """) for documentation or explanatory purposes.

3. Ecosystem and Library in Python

Python's rich ecosystem and extensive libraries have made it one of the most versatile and powerful programming languages. Its adaptability supports a broad range of domains, from web development to artificial intelligence.

3.1. Python's Standard Library

Python's standard library is a collection of modules and packages that provide ready-to-use solutions for common programming tasks:

- **os** and **sys**: Interact with the operating system and manage system-specific parameters.
- **math**: Perform mathematical operations like trigonometry, logarithms, and factorials.
- **statistics**: Conduct statistical calculations such as mean, median, and standard deviation.
- **json**: Work with JSON data for APIs and configuration files.
- **csv**: Read and write CSV files for data manipulation.
- **logging**: Track application events for debugging and monitoring.
- **re**: Handle complex string matching with regular expressions.

3.2. Popular Libraries and Frameworks

Python's ecosystem includes numerous libraries and frameworks tailored to specific domains:

3.2.1. Web Development

- **Django**: A high-level framework for building secure and scalable web applications.
- **Flask**: A lightweight, modular framework for web applications.
- **FastAPI**: Ideal for building APIs with modern standards like OpenAPI and JSON Schema.

3.2.2. Data Science and Machine Learning

- **NumPy**: Perform numerical computations efficiently.
- **Pandas**: Analyze and manipulate structured data (e.g., tabular data).
- **Matplotlib** and **Seaborn**: Create static, animated, and interactive visualizations.
- **Scikit-learn**: Implement machine learning algorithms such as regression, clustering, and classification.
- **TensorFlow** and **PyTorch**: Build and train deep learning models.

- **Jupyter Notebook:** An interactive computing tool for data analysis and documentation.

3.2.3. Automation and Web Scrapping

- **Selenium:** Automate browser tasks like testing or scraping.
- **Beautiful Soup:** Parse HTML and extract useful data from web pages.
- **Requests:** Make HTTP requests to interact with REST APIs.

3.2.4. Scientific Computing

- **SciPy:** Advanced mathematical and statistical computations.
- **SymPy:** Symbolic mathematics and equation solving.
- **OpenCV:** Computer vision tasks, such as image processing and object detection.

3.2.5. Game Development

- **Pygame:** A library for creating 2D games with sound and graphics support.

3.2.6. DevOps and Scripting

- **Fabric:** Automate server tasks.
- **Ansible:** Manage IT automation efficiently.

3.3. Tools and Environments

Python's ecosystem is supported by a variety of tools to enhance development productivity:

- **Integrated Development Environments (IDEs):**
 - **PyCharm:** Feature-rich IDE with debugging and testing support.
 - **VS Code:** Lightweight editor with excellent Python extensions.
 - **Jupyter Notebook:** Combines code, text, and visuals for data exploration.
- **Package Management Tools:**
 - **pip:** Install and manage Python packages.
 - **Conda:** Manage environments and libraries for data science workflows.
- **Testing Tools:**
 - **Pytest:** Framework for writing simple and scalable test cases.
 - **Unittest:** Built-in module for testing Python code.

3.4. Community Support and Documentation

Python's strong community and support system make it beginner-friendly:

- **PyPI (Python Package Index):** Hosts over 400,000 libraries for diverse use cases.

- **Documentation:** Comprehensive resources for all libraries and tools.
- **Forums:** Active help on platforms like Stack Overflow, Reddit, and Python Discord.
- **Conferences:** Events like PyCon foster learning and networking.

3.5. Versatility and Applicability


Python's libraries and tools are utilized in various industries, including:

- **Web Development:** Frameworks like Django for creating dynamic web applications.
- **Data Science:** Libraries like Pandas and Scikit-learn for analyzing large datasets.
- **Artificial Intelligence:** TensorFlow and PyTorch for building intelligent systems.
- **Automation:** Automating repetitive tasks using tools like Selenium.
- **Finance:** Risk analysis and algorithmic trading using NumPy and Pandas.

In conclusion, Python's ecosystem and libraries make it a flexible, efficient, and powerful tool that aligns perfectly with today's technological requirements. Therefore, learning and utilizing Python is a significant step toward both career growth and project success.

4. Use Cases and Industry Adoption of Python

Python is a dynamic and highly utilized coding language that serves as a foundation across multiple sectors owing to its ease of understanding, clarity, and broad array of tools and utilities. Its adaptability and strong community support have made it a preferred choice across diverse technological landscapes. Here are the primary use cases and areas where Python is predominantly adopted:

 Use Python for... [>>> More](#)

Web Development: Django , Pyramid , Bottle , Tornado , Flask , web2py

GUI Development: tkinter , PyGObject , PyQt , PySide , Kivy , wxPython , DearPyGui

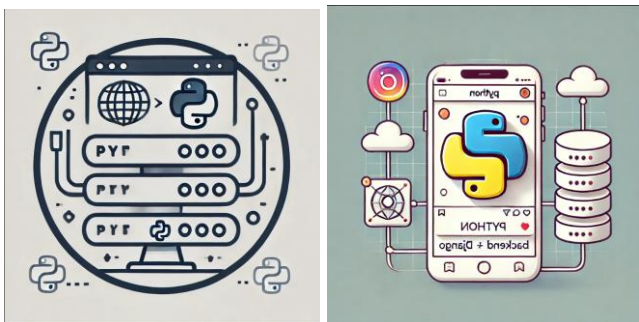
Scientific and Numeric: SciPy , Pandas , IPython

Software Development: Buildbot , Trac , Roundup

System Administration: Ansible , Salt , OpenStack , xonsh

4.1. Web Development

This language is a favored option for crafting online platforms, supported by tools such as Django and Flask. These frameworks simplify the process of creating robust, scalable, and secure web solutions. Companies such as Instagram and Spotify use Python for their backend services. For example, **Instagram** uses Django to handle its massive user base, and **Spotify** relies on Python for personalization and recommendation algorithms.



4.2. Data Science and Machine Learning

The language has transformed the domains of analytics and artificial intelligence. Libraries like Pandas, NumPy, and Matplotlib make data manipulation and visualization easier, while frameworks like TensorFlow, Scikit-learn, and PyTorch enable the creation of complex

machine learning models. Python is the go-to language for data scientists and AI researchers. Examples include **Netflix**, which uses Python to analyze user behavior and offer personalized content recommendations, and **Uber**, which optimizes pricing algorithms and route planning with Python-based data analysis tools. Examples include:

- **Netflix:** Using Python to analyze user behavior and offer personalized content recommendations.
- **Uber:** Optimizing pricing algorithms and route planning with Python-based data analysis tools

4.3. Automation and Scripting

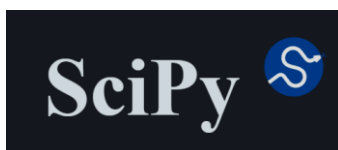
It is commonly employed for handling monotonous operations, including data extraction, organizing files, and validating processes. Its simplicity allows developers to write scripts quickly, improving productivity in various domains. Examples include **Reddit**, which automates post ranking and moderation workflows, and **Dropbox**, which uses Python scripting for file synchronization and automated backups. Examples include:

- **Reddit:** Automation of post ranking and moderation workflows.
- **Dropbox:** Python scripting for file synchronization and automated backups.

4.4. Scientific Computing

In scientific research, Python is used for simulations, data analysis, and visualization. Libraries like SciPy and SymPy provide tools for solving complex mathematical problems, making Python a valuable tool for engineers and researchers. Examples include **CERN**, which analyzes data from the Large Hadron Collider, and **NASA**, which uses Python for conducting space simulations and processing scientific data. Examples include:

- **CERN:** Analyzing data from the Large Hadron Collider.
- **NASA:** Conducting space simulations and processing scientific data.



4.5. Game Development

Python is not traditionally associated with game development, but frameworks like Pygame allow developers to create simple 2D games. It is often used for prototyping and educational purposes in the gaming industry. Examples include **Civilization IV**, where parts of the game logic are written in Python, and **Eve Online**, which uses Python for in-game tools and user interfaces. Examples include:

- **Civilization IV:** Parts of the game logic written in Python.
- **Eve Online:** Using Python for in-game tools and user interfaces.

4.6. DevOps and Cloud Computing

Python plays a significant role in DevOps practices, enabling automation of deployment and configuration tasks. It is also used in cloud computing platforms like AWS and Google Cloud for scripting and creating serverless applications. Examples include **Google Cloud**, where Python scripts are used for API integrations and serverless architecture, and **AWS Lambda**, which relies on Python to build efficient serverless applications. Examples include:

- **Google Cloud:** Python scripts for API integrations and serverless architecture.
- **AWS Lambda:** Building efficient serverless applications with Python.

4.7. Cybersecurity

It finds application in security for crafting utilities to identify weaknesses, study malicious software, and replicate intrusion testing. Libraries such as Scapy and tools like Metasploit are based on Python. Examples include **Metasploit Framework**, which uses Python modules for penetration testing, and **Wireshark**, which extends network analysis capabilities with Python scripts. Examples include:

- **Metasploit Framework:** Python modules for penetration testing.
- **Wireshark:** Extending network analysis capabilities with Python scripts.

4.8. Education and Training

Due to its beginner-friendly syntax, Python is widely used as the first programming language in schools, universities, and coding bootcamps. It helps students learn programming concepts without the complexity of other languages. Examples include **Codecademy**, which offers Python courses designed for beginners, and **MIT**, which leverages Python in introductory computer science courses. Examples include:

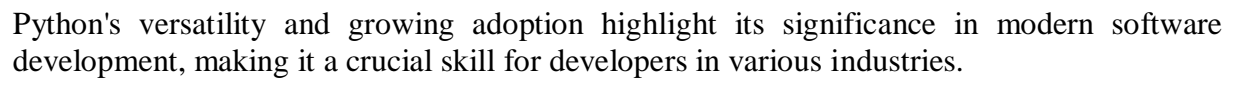
- **Codecademy:** Python courses designed for beginners.
- **MIT:** Leveraging Python in introductory computer science courses.

4.9. Strengths in Industry Adoption

Python's popularity stems from several key strengths:

- **Ease of Use:** Python's syntax is intuitive, making it accessible for beginners and professionals alike.
- **Extensive Ecosystem:** The availability of libraries and frameworks allows Python to be applied in diverse fields.
- **Strong Community:** A large and active community ensures continuous improvement, extensive documentation, and abundant learning resources.

- Python's versatility and growing adoption highlight its significance in modern software development, making it a crucial skill for developers in various industries.



5. Project Implementation

This project creates a "Type Speed Calculator" application that measures users' typing speed and accuracy. The user starts typing the randomly displayed words on the screen and presses the spacebar to move to the next word.

The program provides the user with a one-minute time limit and calculates the following during this time:

- The number of correctly typed words,
- The total number of words typed,
- The accuracy percentage,
- Words per minute (WPM).

When the time is up, the results are displayed to the user in a message box. The program features a modern graphical interface for an easy and user-friendly experience.

5.1. Libraries Used in the Code

```
1  import tkinter as tk
2  from tkinter import messagebox
3  import random
4  import time
5  from tkinter import ttk
6
```

tkinter:

- Python's standard GUI (Graphical User Interface) library.
- Used in this project to create the window, user input fields (Entry), labels (Label), and other UI components.

messagebox (from tkinter):

- Used to display informational messages, warnings, or error messages to the user.
- For example, it is used to notify the user when the word file is missing or when the timer runs out.

random:

- Python's library for random operations.
- In this project, it is used to select random words for the user to type (random.choice()).

time:

- A standard Python library for handling time-related operations.
- In this project, it tracks the game's start time and controls the timer (time.time()).

ttk (from tkinter):

- ttk (Themed Tkinter Widgets) provides more modern and customizable UI components.
- In this project:
 - Used for input fields (ttk.Entry),
 - Labels (ttk.Label),
 - And theme customization (e.g., font, colors, etc.).

These libraries work together to create a user-friendly, dynamic typing speed calculator application.

5.2. Variables in the Code

```
# Constants
WORDS = load_words_from_file("words.txt") # Load words from file

if not WORDS: # Stop program if file is empty or not found
    raise ValueError("Word list is empty or not loaded correctly.")

# Global Variables
correct_word_count = 0
total_word_count = 0
start_time = None
time_left = 60
current_word = None
```

WORDS:

- Purpose: Stores the list of words loaded from a file.
- Words are loaded using the load_words_from_file("words.txt") function.
- If the words cannot be loaded or the list is empty, the program stops.

correct_word_count:

- Purpose: Tracks the number of words the user has typed correctly.
- This variable is incremented when the user types the displayed word correctly.

total_word_count:

- Purpose: Tracks the total number of words typed by the user (both correct and incorrect).
- This variable is incremented every time the user enters a word.

start_time:

- Purpose: Records the time when the game begins.
- A value is assigned to this variable when the user starts typing for the first time (inside the `handle_space_press` function).

time_left:

- Purpose: Tracks the remaining time for the game. It is initially set to 60 seconds.
- This value decreases by one every second when the timer runs.

current_word:

- Purpose: Holds the current word that the user needs to type.
- A random word is selected by the `display_new_word` function and assigned to this variable.

5.3. Explanation of Functions in the Code

```
# Function to load words from file
def load_words_from_file(file_path):
    try:
        with open(file_path, 'r') as file:
            words = file.read().splitlines() # Reads each line as a separate word
            return words
    except FileNotFoundError:
        messagebox.showerror("Error", "Words file not found. Please check the file path.")
        return []
```

```
# Timer Functions
def start_timer():
    global time_left
    if time_left > 0:
        time_left -= 1
        timer_label.config(text=f"Remaining time: {time_left} sec")
        window.after(1000, start_timer)
    else:
        end_game()

def end_game():
    input_field.config(state="disabled")
    accuracy = (correct_word_count / total_word_count) * 100 if total_word_count > 0 else 0
    wpm = correct_word_count
    messagebox.showinfo("Time is up!",
                        f"Correct Words: {correct_word_count}\n"
                        f"Words Per Minute (WPM): {wpm}\n"
                        f"Total Words: {total_word_count}\n"
                        f"Accuracy: {accuracy:.2f}%")
    window.quit()
```

```

def handle_space_press(event):
    global correct_word_count, total_word_count, start_time, current_word

    if start_time is None:
        start_time = time.time()
        instruction_label.destroy()
        start_timer()
        display_new_word()

    user_input = input_field.get().strip()
    total_word_count += 1

    if not user_input:
        return "break"

    event.widget.delete(0, tk.END)

    if user_input == current_word:
        correct_word_count += 1
        input_field.config(foreground="green")
    else:
        input_field.config(foreground="red")

    update_labels()
    input_field.delete(0, tk.END)
    display_new_word()

def validate_partial_input(event):
    user_input = input_field.get().strip()
    if current_word and current_word.startswith(user_input):
        input_field.config(foreground="green")
    else:
        input_field.config(foreground="red")

def display_new_word():
    global current_word
    current_word = random.choice(WORDS)
    word_label.config(text=current_word)

def update_labels():
    correct_label.config(text=f"Correct Words: {correct_word_count}")
    total_label.config(text=f"Total Words: {total_word_count}")

```


load_words_from_file(file_path)

- Purpose: Loads a list of words from a file.
- Steps:
 - Opens the file specified by file_path in read mode.
 - Reads each line as a separate word and creates a list of words using splitlines().
 - If the file is not found, displays an error message using messagebox and returns an empty list.

start_timer()

- Purpose: Implements a countdown timer for the game.
- Steps:
 - Decrements the time_left variable by 1 each second.
 - Updates the timer_label to display the remaining time.
 - If the time reaches 0, calls the end_game() function to stop the game.
 - Uses window.after(1000, start_timer) to repeatedly call itself after 1 second.

end_game()

- Purpose: Ends the game when the timer reaches 0.
- Steps:
 - Disables the input field to prevent further typing.
 - Calculates:
 - Accuracy: Correct words as a percentage of total words typed.
 - WPM (Words Per Minute): Equals the count of correct words.
 - Displays the results using a messagebox.
 - Closes the application with window.quit().

handle_space_press(event)

- Purpose: Handles the event when the user presses the space bar.
- Steps:
 - Starts the timer and displays the first word if it's the first keypress (start_time is None).
 - Increments total_word_count and checks the user's input:
 - If the input matches the current_word, increments correct_word_count and highlights the input field in green.
 - Otherwise, highlights the input field in red.
 - Updates the score labels using update_labels().
 - Displays a new word by calling display_new_word().

validate_partial_input(event)

- Purpose: Validates partial input as the user types a word.
- Steps:
 - Checks if the current word (current_word) starts with the user's input.
 - If valid, highlights the input field in green; otherwise, highlights it in red.

display_new_word()

- Purpose: Displays a new random word for the user to type.
- Steps:
 - Selects a random word from the WORDS list using random.choice().
 - Updates the word_label to show the selected word.

update_labels()

- Purpose: Updates the labels showing the count of correct words and total words typed.
- Steps:
 - Updates correct_label with the count of correct words.
 - Updates total_label with the count of total words typed.

5.4 UI Setup and Component in the Code

```
# UI Setup
window = tk.Tk()
window.title("Type Speed Calculator")
window.geometry("600x400")
window.configure(background="#1c1e24")

style = ttk.Style()
style.theme_use("clam")

style.configure(
    "CustomEntry.TEntry",
    fieldbackground="#2b2d35",
    foreground="ffffff",
    font=("Helvetica", 20)
)

style.configure(
    "CustomLabel.TLabel",
    background="#1c1e24",
    foreground="ffffff",
    font=("Helvetica", 18)
)
```

```

style.configure(
    "HighlightLabel.TLabel",
    background="#1c1e24",
    foreground="#ffc107",
    font=("Helvetica", 22)
)

style.configure(
    "SuccessLabel.TLabel",
    background="#1c1e24",
    foreground="#28a745",
    font=("Helvetica", 20)
)

```

window = tk.Tk()

- Purpose: Initializes the main application window.
- Details:
 - window.title("Type Speed Calculator"): Sets the title of the application window.
 - window.geometry("600x400"): Specifies the size of the window (600 pixels wide and 400 pixels tall).
 - window.configure(background="#1c1e24"): Sets the background color of the window to a dark shade (#1c1e24).

style = ttk.Style()

- Purpose: Creates a style object for customizing the appearance of UI elements using ttk widgets.

style.theme_use("clam")

- Purpose: Applies the "clam" theme to ttk widgets. The "clam" theme provides a modern, clean look compared to the default Tkinter widgets.

Custom Styles

The style.configure() method is used to define and customize specific styles for widgets (like Entry, Label). Each style has a unique name, such as "CustomEntry.TEntry".

CustomEntry.TEntry

- Purpose: Defines the style for input fields (Entry widgets).
- Customization:

- `fieldbackground="#2b2d35"`: Sets the background color of the input field to a dark shade.
- `foreground="#ffffff"`: Sets the text color inside the input field to white.
- `font=("Helvetica", 20)`: Specifies the font type (Helvetica) and size (20) for text in the input field.

CustomLabel.TLabel

- Purpose: Defines the style for standard labels.
- Customization:
 - `background="#1c1e24"`: Matches the label's background color with the application window's background.
 - `foreground="#ffffff"`: Sets the label text color to white.
 - `font=("Helvetica", 18)`: Specifies the font type and size for label text.

HighlightLabel.TLabel

- Purpose: Defines a style for labels that require highlighted text, such as the current word being typed.
- Customization:
 - `background="#1c1e24"`: Matches the label background color with the window background.
 - `foreground="#ffc107"`: Sets the text color to a yellowish hue (#ffc107), giving it a "highlighted" look.
 - `font=("Helvetica", 22)`: Uses a slightly larger font size (22) for emphasis.

SuccessLabel.TLabel

- Purpose: Defines a style for labels that indicate success, such as the correct word count or feedback.
- Customization:
 - `background="#1c1e24"`: Matches the background color with the main window.
 - `foreground="#28a745"`: Sets the text color to green (#28a745), representing success or correctness.
 - `font=("Helvetica", 20)`: Uses a medium-sized font (20) for readability.

UI Components

```
# Instruction Label
instruction_label = ttk.Label(window, text="PRESS SPACE TO START!", style="HighlightLabel.TLabel")
instruction_label.pack(pady=20)

# Word Display Label
word_label = ttk.Label(window, text="", style="CustomLabel.TLabel")
word_label.pack(pady=20)

# Input Field
input_field = ttk.Entry(window, style="CustomEntry.TEntry", font=("Helvetica", 15))
input_field.pack(pady=10, ipadx=5, ipady=5)
input_field.bind("<space>", handle_space_press)
input_field.bind("<KeyRelease>", validate_partial_input)

# Timer Label
timer_label = ttk.Label(window, text="Remaining time: 60 sec", style="HighlightLabel.TLabel")
timer_label.pack(pady=10)

# Correct Words Label
correct_label = ttk.Label(window, text="Correct Words: 0", style="SuccessLabel.TLabel")
correct_label.pack(pady=5)

# Total Words Label
total_label = ttk.Label(window, text="Total Words: 0", style="CustomLabel.TLabel")
total_label.pack(pady=5)

window.mainloop()
```

instruction_label

- Purpose: Displays instructions to the user before the game starts.
- Code Details:
 - `ttk.Label(window, text="PRESS SPACE TO START!", style="HighlightLabel.TLabel")`: Creates a label with the text "PRESS SPACE TO START!" and applies the previously defined `HighlightLabel.TLabel` style (yellow text on a dark background, bold font).
 - `instruction_label.pack(pady=20)`: Adds the label to the window and positions it with 20px vertical padding.

word_label

- Purpose: Displays the current word the user needs to type.
- Code Details:
 - `ttk.Label(window, text="", style="CustomLabel.TLabel")`: Creates an empty label styled with `CustomLabel.TLabel` (white text on a dark background).
 - `word_label.pack(pady=20)`: Adds the label to the window with 20px vertical padding. The text is updated dynamically when a new word is displayed.

input_field

- Purpose: Allows the user to input the words they type.
- Code Details:
 - `ttk.Entry(window, style="CustomEntry.TEntry", font=("Helvetica",15)):` Creates an input field styled with `CustomEntry.TEntry`. A smaller font size (15) is specified for typing.
 - `input_field.pack(pady=10, ipadx=5, ipady=5):` Adds the input field to the window with 10px vertical padding and internal padding (5px) for better spacing.
 - Bindings:
 - `input_field.bind("<space>", handle_space_press):` Detects when the user presses the space key. This triggers the `handle_space_press` function, which processes the input and starts the timer if it hasn't already started.
 - `input_field.bind("<KeyRelease>", validate_partial_input):` Detects when the user releases a key. This triggers the `validate_partial_input` function, which checks if the typed text matches the current word so far.

timer_label

- Purpose: Displays the remaining time for the game.
- Code Details:
 - `ttk.Label(window, text="Remaining time: 60 sec", style="HighlightLabel.TLabel"):` Creates a label showing the initial time (60 seconds) and applies the `HighlightLabel.TLabel` style (yellow text for emphasis).
 - `timer_label.pack(pady=10):` Adds the label to the window with 10px vertical padding. The text updates dynamically as the timer decreases.

correct_label

- Purpose: Displays the count of correctly typed words.
- Code Details:
 - `ttk.Label(window, text="Correct Words: 0", style="SuccessLabel.TLabel"):` Creates a label with the initial count of correct words (0) and applies the `SuccessLabel.TLabel` style (green text to indicate success).
 - `correct_label.pack(pady=5):` Adds the label to the window with 5px vertical padding. The text updates dynamically as the user types correct words.

total_label

- Purpose: Displays the total number of words typed by the user (both correct and incorrect).
- Code Details:

- `ttk.Label(window, text="Total Words: 0", style="CustomLabel.TLabel"):` Creates a label with the initial total word count (0) and applies the `CustomLabel.TLabel` style (white text).
- `total_label.pack(pady=5):` Adds the label to the window with 5px vertical padding. The text updates dynamically as the user types words.

Window Main Loop

- Code Details:
 - `window.mainloop():` Starts the event loop for the application. This loop keeps the window open and listens for user actions (e.g., keypresses) until the application is closed.

6. Comparison and Evaluation of Python and LISP

Python and LISP are two programming languages with distinct features and use cases, each excelling in particular domains of software development. This section provides a detailed comparison of these languages based on the following criteria: Syntax, Performance, Community Support, and Applicability.

6.1. Syntax: Is it easy to learn and read?

Python is renowned for its clean and highly readable syntax, making it an ideal choice for beginners and professionals alike. Its design philosophy emphasizes simplicity and code readability, allowing developers to write concise and intuitive code. In contrast, LISP employs a syntax heavily reliant on nested parentheses, which can be challenging for newcomers to interpret. However, this design lends itself well to symbolic computation and mathematical representations, making LISP a preferred language in domains requiring high levels of abstraction.

Python Example: List Comprehension for Data Transformation

```
python

# Transforming a list of numbers to their squares if they are even
numbers = [1, 2, 3, 4, 5, 6]
squared_evens = [n**2 for n in numbers if n % 2 == 0]

print(squared_evens) # Output: [4, 16, 36]
```

Python's list comprehensions offer a concise way to filter and transform data in a single, readable line of code.

LISP Example: Using mapcar and lambda for Data Transformation

```
lisp

; Transforming a list of numbers to their squares if they are even
(setq numbers '(1 2 3 4 5 6))
(setq squared-evens
  (remove nil
    (mapcar
      (lambda (n) (if (evenp n) (* n n) nil))
      numbers)))

(print squared-evens) ; Output: (4 16 36)
```

LISP requires multiple functions (`mapcar`, `lambda`, `remove`, `if`) and nested parentheses to achieve the same result, making the code less concise and harder to interpret at a glance.

6.2. Performance: How does the language perform compared to others for similar tasks?

While Python is interpreted and generally considered slower for computationally intensive tasks, its integration with optimized libraries such as NumPy and Cython significantly boosts its performance. LISP, on the other hand, has historically been lauded for its efficient execution of recursive algorithms and symbolic computations. Its macro system and compilation capabilities allow for fine-tuned optimizations, particularly in scenarios requiring real-time responsiveness.

6.3. Community Support: Availability of learning resources, documentation, and active community

Python boasts a massive and active community, providing extensive documentation, tutorials, and third-party libraries. This ecosystem greatly facilitates learning and application development. LISP, while supported by a passionate and knowledgeable community, lacks the extensive resources and modern libraries available for Python. As a result, finding solutions to specific challenges or learning LISP might require more effort and specialized guidance.

6.4. Applicability: What type of projects is the language most suitable for? What are its limitations?

Python has emerged as a versatile, general-purpose language with widespread adoption in diverse fields such as web development, data science, machine learning, and automation. For instance, companies like Google and Netflix utilize Python extensively for building scalable applications and data pipelines. Conversely, LISP is predominantly associated with artificial intelligence research and academic settings. Its expressive power and flexibility have made it a foundational language in the development of early AI systems and expert systems. However, its industry adoption remains limited due to its niche focus and smaller ecosystem.

Python Vs LISP Comparison			
	Criteria	Python	LISP
1	Syntax	Clean and highly readable, beginner-friendly.	Heavily reliant on nested parentheses, ideal for symbolic computation.
2	Performance	Slower for intensive tasks but boosted by libraries like NumPy and Cython.	Efficient for recursive algorithms and symbolic computations.
3	Community Support	Massive and active community with extensive resources and libraries.	Smaller but passionate community, fewer modern resources.
4	Applicability	Versatile, used in web development, data science, machine learning, and automation.	Focused on AI research and academic settings; limited industry use.

CONCLUSION

In this project, we explored and implemented a "Type Speed Calculator" using Python, demonstrating the language's versatility and suitability for building modern applications. The project provided insights into Python's history, core features, ecosystem, and wide-ranging applications across various industries. By leveraging Python's strengths, including its simplicity, rich libraries, and extensive community support, we successfully developed a user-friendly program that measures typing speed and accuracy within a one-minute time frame.

Through hands-on implementation, we gained a deeper understanding of Python's dynamic capabilities, such as its GUI-building tools (Tkinter), randomization functions, and time-based operations. Comparing Python with LISP further highlighted Python's readability, extensive use cases, and robust community support, positioning it as an ideal choice for general-purpose programming.

The "Type Speed Calculator" project exemplifies how Python's features can be utilized to create efficient and interactive applications. This experience underscored Python's flexibility in adapting to various domains and reaffirmed its role as a critical skill for modern software development.