School of Computer Science and Statistics

Flask from Scratch
- A Learning Platform for the Flask Web Framework

Author: Krishanu Dey
Project Supervisor: Glenn Strong

April 2022

A Dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Computer Science
Abstract

The audience for beginner programming is expanding as more and more people have a growing interest in the subject due to its high applicability and growing job sector in the modern world. Students are being exposed to the world of programming at a younger age than ever before. Thus, the popularity of engaging tools to learn the abstract concepts of programming has grown phenomenally in recent years.

There are various beginner-friendly programming environments; two of particular relevance are: 1. Scratch - from the MIT Media Lab, and 2. Pytch - produced in TCD and is intended to follow on to Scratch for introducing Python. In this research work, we consider a potential follow-on environment to Pytch and provide a new learning platform for users to explore web application development in a platform designed to leverage existing knowledge gained in Scratch and Pytch.

‘Flask from Scratch’ is a fully-browser based online learning platform to teach web app development using the Pythonic web framework called Flask. The proposed service is intended to be used as part of an educational framework for beginners who are moving on from simple introductory programming environments and want to start building more practical web applications but want a low barrier to entry. The new online environment would enable students to quickly jump into the Flask framework without any complex setups and provide guided tutorials to learn the crucial concepts of Flask and web application development in general.
Contents

Title Page 1
Abstract 2
Contents 3
List of Figures 6

Chapter 1 - Introduction 7
  1.1 Motivation 7
  1.2 Research Question 8
  1.3 Research Objectives and the Key Milestones 9
  1.4 Report Overview 11

Chapter 2 - Background 13
  2.1 Importance of Online IDEs as Learning Tools 13
    2.1.1 Easy to Setup and Access 13
    2.1.2 Teach Fundamental Concepts Independently to Reduce Cognitive Overload 14
    2.1.3 Learning Support and Integrated Tools 14
  2.2 Scratch 14
  2.3 Python 15
  2.4 Coding in Browsers 16
  2.5 Skulpt 17
  2.6 Pytch 18
  2.7 Web Apps 18
    2.7.1 Flask 19
    2.7.2 Django 20
2.8 State of the Art - Popular Online Learning Platforms

2.8.1 Replit
2.8.2 CodeSandbox

Chapter 3 - Design

3.1 Why choose a Python-based framework?
3.2 Introducing Web Application Development with the Flask Web Framework
3.3 Entirely Browser-Based
  3.3.1 Scalability
  3.3.2 Security
  3.3.3 Alternative Approaches Considered
3.4 Why Do We Need (Another) Platform?
3.5 Flask From Scratch Design
  3.5.1 Phase I - Flask Library in Skulpt
    1 Detailed Design
    2 Feature Set of the Skulpt-based Flask library
    3 Alternative Approaches Considered
  3.5.2 Phase II - ‘Flask from Scratch’ IDE Web Application
    1 Design of the Platform’s Web Application
    2 User Interface / User Experience
3.6 Tutorials

Chapter 4 - Implementation

4.1 Phase I - Flask Library in Skulpt
  4.1.1 Running the App
  4.1.2 Mapping URL Routes to their respective View Functions
  4.1.3 Handling Routes - including Dynamic Routes with Variables
  4.1.4 Redirecting to other URL Routes Dynamically
List of Figures

1. **Figure 1.1**: Gantt chart summarising the project milestone schedule.

2. **Figure 2.1**: Scratch Development Environment.

3. **Figure 2.2**: Graph of the most popular programming languages based on PYPL data. Graph created by https://darly.solutions/ [10]

4. **Figure 2.3**: Replit IDE running a ‘hello world’ Flask web app.

5. **Figure 2.4**: CodeSandbox IDE running React Js web app.

6. **Figure 3.3**: User interface Mockup for Integrated Development Environment.

7. **Figure 3.4**: Final User interface for the IDE.

8. **Figure 3.5**: ‘Flask from Scratch’ Home Page User Interface Design.

9. **Figure 3.6**: ‘Projects Page’ User Interface Design.

10. **Figure 3.7**: A guided tutorial walkthrough on the ‘Flask from Scratch’ platform.

11. **Figure 4.1**: Embedded browser rendering an HTML login form.

12. **Figure 4.2**: IDE presenting a syntax error to the user.
Chapter 1 - Introduction

1.1 Motivation

The audience for beginner programming is expanding as more and more people have a growing interest in the subject due to its high applicability and growing job sector in the modern world. Students are being exposed to the world of programming at a younger age than ever before.

As studied in “Restart: The Resurgence of Computer Science in UK Schools” [6], N. Brown et al. state that there has been a sudden massive growth in the interest to learn programming through lobbying interest groups and pressure from the technology industry. The force was so compelling that the subject has become mandatory for school students aged five to fourteen as it is an integral part of the UK school curriculum now. Though, as stated in the paper, this has created new problems: there has been a shortage of trained IT teachers and a lack of resources for teaching this relatively new subject in schools to young students.

Thus, the popularity of engaging tools to learn the brand new concepts of programming and computer science technologies has grown phenomenally in recent years. A plethora of these tools exist online and serve as resources for teaching programming to novice students and are made highly accessible and engaging to appeal to the young crowd. They can be either followed as part of the classroom coursework or for individual interests. Two of these resources that could be highlighted include:

1. Scratch [7] - is a high-level block-based visual programming language, and the website is targeted primarily at children aged eight to sixteen.
2. Pytch [8] - serves as a platform to guide students from the block-based programming world to the text-based programming world by acting as a transitory step.
The following research project, ‘Flask from Scratch’, attempts to extend this trajectory to lead the students to the web application development world - following their successful training in Scratch and Pytch. Though this platform focuses on Scratch and Pytch users through a similar user interface and user experience, any user who is comfortable in Python and is interested in learning web application development with the Flask web framework can use this platform.

The new online environment, ‘Flask from Scratch’, would enable students to quickly jump into the framework without requiring them to do any complex system setups. The service also provides tutorials to teach the crucial concepts of Flask and web application development in general.

1.2 Research Question

**Research Question I:** What would be a suitable follow-on programming environment for Pytch users to allow them to continue learning Python skills?

**Research Question II:** What would be a suitable technical design for this system?

**Research Question III:** Explore how to build a prototype for such a system suitable for use with test users.

The ‘Flask from Scratch’ platform is intended to be used as part of an educational framework for beginners moving on from simple introductory programming environments and building more practical web applications with a low barrier to entry.

It is essential to adopt a research-based approach to determine what programming languages and technologies would benefit students on this transitional learning path. Extensive research has already been done on learning platforms, the lack of readily available engaging programming resources and the most instrumental ways of teaching programming to novice users. Hence, there is plenty of information available on the topic to support this research. It also implies that scholars and other researchers have explored the topic in-depth, highlighting the importance of exploring further.
1.3 Research Objectives and the Key Milestones

The proposed objective is to undertake product research to design and implement a new environment that will provide an online learning server for creating introductory Flask web applications.

The research project will be deemed successful if the following design principles are warranted in the final product. Five fundamental design principles were formulated based on the research done by M. Kolling [19, 20] which states the requirements for learning environments. Each of my design principle links to specific requirements set by M.Kolling as identified in the brackets:

**DP1: Simplicity of Access and Setup (“Availability”, “Ease of Use”)**

Ensure minimal need for complex setup and administration for the users of the platform. Build a scalable product to ensure availability to all in the future.

**DP2: Allow the User to Leverage their Existing Knowledge (“Extendibility”)**

Users should be able to use their existing knowledge to connect to their current interests to make the platform appealing to them.

**DP3: Present a Sense of Authenticity (“Consistent mental model”)**

The underlying programming model must be accurately reflected in all representations of the user’s work in the environment.

**DP4: Support User Autonomy via Integrated Learning Materials (“Support for teachers”)**

As concluded by M. Kolling in the study, teachers at secondary schools struggle to have enough time to develop proper teaching materials. Thus, it is essential to provide explicit support for teachers with tutorials and documentation.
**DP5: Potential for continued activity ("Extendibility")**

Ensure users have a clear path for the next steps such that after successful training, they are aware of what they can learn next - either a new technology or master the current one.

Evaluation based on the design goals are described in detail in the Evaluation chapter later in this document.

Overall, the project included the following key milestones that were established as the research was conducted over the academic year:

- **M1** - Conduct product research to understand and detail what present learning resources are lacking and what type of activity could alleviate the issue.
- **M2** - Learn how online integrated development environments can execute code on the browser-side, specifically with the Skulpt project library.
- **M3** - Implement a minimum viable product (MVP) of the Flask library in the Skulpt project, such that the user can create and run a web application that renders at least one HTML page. It should also accept basic parameters through the URL to render information on the page.
- **M4** - Add additional functionality and extend the Flask library API to enable the user to make more diverse and sophisticated Flask web applications.
- **M5** - Perform product research to design the user interface and user experience for the platform to ensure a seamless and intuitive user experience.
- **M6** - Implement the web application for the learning platform which would serve as the front-end of the service.
- **M7** - Create tutorials, walkthroughs and ‘Help’ documentation for users.
- **M8** - Finalise the implementation and write up the dissertation.

The following Gantt chart summarises how the work was planned over the year:
Figure 1.1: Gantt chart summarising the project milestone schedule.

1.4 Report Overview

The report has been divided up into the following sections:

Section I - Introduction: The chapter discusses the current computer science education system's difficulties and highlights the importance of adding more support for new computer science learners. This section also provides a brief description of the solution to be applied and clearly defines what the design principles are.

Section II - Background: This chapter explores the current literature on the state of teaching programming to novice users and describes all the background information needed later to understand the design and implementation decisions of this research project. This chapter also explores state of the art in the online learning platform domain.

Section III - Design: This chapter presents an overview of the platform's design decisions based on the research literature and relevant information presented in the background section. The detailed design and high-level feature set of the overall platform are described in this chapter.

Section IV - Implementation: This chapter provides a more detailed description of the proposed solution and describes the implementation details of the platform.
**Section V - Evaluation:** This chapter evaluates the final version of the learning platform and compares the product with the design principles set initially.

**Section VI - Conclusion:** This chapter concludes the research project with a final reflection of the overall project and describes high priority tasks for any future work.
Chapter 2 - Background

2.1 Importance of Online IDEs as Learning Tools

Code editors and Integrated Development Environments (IDEs) are some of the most popular ways to learn to program and get comfortable with the programming environment. Code editors are text editors that provide extra functionality to assist in the development of software. IDEs are software development environments that offer a more comprehensive collection of features with their in-built code editors.

Though even with the most popular and easy-to-use IDEs, these environments expect you to be proficient, or at least be familiar, with the programming language and frameworks one uses there. Generally, the IDEs would not be designed to guide the users to teach the computer science concepts.

Another set of tools that exist are online learning platforms; there are various free online learning platforms consisting of IDEs with guided support to teach computer science fundamentals. These platforms have gained massive popularity in recent years.

2.1.1 Easy to Setup and Access

Platforms would be highly accessible to inexperienced students if there is a minimal set-up barrier to start using the platform. This could be ensured by an online IDE as it would provide to the user a starting environment with the provided configurations done beforehand to prevent students from becoming overwhelmed with repetitive but complex administrative tasks. Thus, providing a learning platform on the browser would greatly increase the accessibility of the product.

Additionally, the accessibility could be increased further by letting the users work at a high level of abstraction such that the unnecessary details are hidden away. This would enable users to start building on top of the starting states and create their own applications faster.
2.1.2 Teach Fundamental Concepts Independently to Reduce Cognitive Overload

Students can get overwhelmed when they get introduced to many complex concepts at the start. As stated by Miller et al. [21], the cognitive load theory states that the amount of concurrent cognitive problems highly influences the capacity to learn. As students are introduced to more cognitive problems, the learning rate and the memory retention of the concepts drastically reduce.

As students start developing web applications, numerous complex concepts are introduced to them initially, making the learning process difficult. Thus, building learning platforms with well-designed tutorials for each concept is beneficial. These tutorials would ideally introduce the fundamental concepts to the users as independently as possible to let users deal with just one cognitive problem at a time.

2.1.3 Learning Support and Integrated Tools

The learning process could be made more engaging through learning platforms by providing integrated tools and interactive tutorials, making the process more captivating and entertaining to the students, especially the younger students. Well-designed tutorials with detailed walkthroughs would also ensure a low learning curve for inexperienced users.

2.2 Scratch

Scratch, the world’s largest coding community for children, is a coding language with a simple visual interface that allows young people to create digital stories, games, and animations. [7]
Scratch is a programming language designed for children aged eight to sixteen; however, the service is used by many individuals of varied ages to start their journey of learning how to code. The visual interface has helped decrease the abstractness of programming concepts and keep the users engaged in this learning process, especially the young students.

Scratch projects are being created by millions of individuals in various locations, including homes, schools, libraries, and community halls.

2.3 Python

Python is a widely used high-level interpreted language created by Guido van Rossum and was first released in 1991 [11]. Python emphasises code readability, making it an excellent language to teach programming. It is also very versatile, with an extensive standard library that covers a wide range of use cases, as well as a thriving community that creates libraries and tools to fill any gaps.

PYPL (PopularitY of Programming Language) index ranks programming languages based on the number of times tutorials of a language are searched on Google (data provided by Google Trends). According to PYPL, Python is currently the most
popular programming language, as shown in the graph below and has grown the most in the past five years (by 12.8%) [9].

![Bar graph showing the popularity of programming languages with Python as the most popular at 30.34%, followed by Java at 17.23%, JavaScript at 8.65%, and other languages like C#, C/C++, PHP, R, Objective C, Swift, and MATLAB with lesser percentages.](https://darly.solutions/)

**Figure 2.2**: Graph of the most popular programming languages based on PYPL data. Graph created by [https://darly.solutions/][10]

2.4 Coding in Browsers

Before discussing the specifics of the research and the implementation, I would like to discuss the browser mechanics and the functionalities they provide to the users and the developers. This section will also explore the workings of scripts on the browser-side and how the browser cannot interpret any non-javascript language on the client-side directly.

The browser is essentially an interpreter. The only languages that the browser can interpret are HTML, CSS and Javascript (JS) [1].

Different browsers have different engines to compile and execute the javascript code. A few notable engines [2] consist of:

1. Google Chrome, Edge and other Chromium browsers: V8 engine.
2. Mozilla Firefox browser: SpiderMoney engine
3. Safari Browser: JavaScriptCore engine
4. Internet Explorer browser: Chakra engine

Though various companies and groups have developed multiple Javascript engines, they all essentially provide the same support - to enable compiling and executing Javascript code at the client-side on their browsers.

The question may arise: how are other non-JS programming languages, such as Python and C++, compiled in the browser? The short answer is that it is not possible to interpret these languages in the browser directly like Javascript. This creates a problem for online integrated development environments (IDE) for non-Javascript languages. The popular online IDE platforms currently simply forward the code text inputted by the user as a simple string to a backend server that compiles and executes the code in their local runtime, and returns the output to the client. This approach creates scalability, security, and privacy issues that we will discuss later in the Design chapter (Section 3.3: Entirely Browser-based).

### 2.5 Skulpt

Skulpt is an entirely in-browser implementation of Python - with no preprocessing, plugins or server-side support requirements. [3]

Skulpt is a system that transpiles Python code to Javascript code exclusively on the browser. It is an open-source project on GitHub with 863 forks and 3,100 stars with 86 contributors. As discussed later, Skulpt powers the Pytch system and will also power the ‘Flask from Scratch’ system.

It essentially extends the behaviour of the browser to not just to interpret Javascript code (as discussed in the previous section) but also lets us interpret Python code on the browser itself.
2.6 Pytch

Pytch [8] is part of a research project at Trinity College Dublin by Glenn Strong and Benjamin North. Pytch aims to smooth a learner's journey from Scratch to Python by adding an intermediary step in between.

Once a student gets proficient with Scratch, a common next step is Python - a programming language widely used in education and industry. Though Python proves to be a giant leap from Scratch; the student has to make two jumps at once:

- They must correctly type their code into an editor or IDE, getting all syntax details right.
- They also have to leave behind the Scratch world of sprites, costumes, sounds, when this sprite clicked scripts, and so on.

Pytch is a bridge between these two worlds. It has Scratch's learner-friendly sprites, event-driven scripts, graphics, sounds, etc., while introducing the student to the idea of writing textual Python code instead of dragging and dropping blocks. In this way, they keep all the knowledge, intuition and skills that they have built up with Scratch and can focus on the task of learning the Python language.

Pytch executes all Python code on the browser-side using the Skulpt system (Skulpt - section 2.5). The Pytch forked repository of Skulpt was highly modified so that it could be supported in the Pytch system. One of these modifications included adding support for concurrency; Skulpt inherently did not support concurrency, but it was implemented in the Pytch implementation to provide support for concurrent threads to manage multiple actors and actions on the stage.

2.7 Web Apps

A web application is a computer program that utilises web browsers and web technology to perform tasks over the Internet. [13]

There are multiple benefits of using web applications compared to simple desktop applications:
- The web application runs on the browser and hence does not depend on the operating system of the user’s device.
- Everyone accesses the identical version of the application on the browser, which helps avoid incompatibility errors if some users skip critical updates.
- Business costs are reduced due to lower maintenance support required by web applications.

Hence, numerous web application development frameworks have become immensely popularised in the past decade. It has become essential to have experience with web application frameworks to be a successful full-stack engineer in the current industry.

2.7.1 Flask

Flask is a full-stack Pythonic web application micro-framework created by Armin Ronacher in 2010. It allows users to develop and deploy web applications using Python swiftly.

Flask is widely popular as it is easy to learn and does not have a steep learning curve; the fact that a ‘Hello World’ flask web app can be created with just a few lines of code highlights how lightweight the skeletal framework is.

Flask depends on the Jinja template engine and the Werkzeug WSGI toolkit [15], which are described briefly below:

**Jinja:** is a templating engine that is efficient, expressive, and extendable. The template has special placeholders that allow you to write code that looks like Python syntax. Data is supplied to the template in order to render and display the final HTML page. [16]

**WSGI:** The Web Server Gateway Interface (WSGI) is used as a Python web application development standard. The WSGI protocol defines the most common interface between web servers and web applications. [17]
**Werkzeug**: is a WSGI toolkit that implements requests, response objects, and utility functions. This allows a web frame to be created on top of it. [17]

2.7.2 Django

Django is a powerful Python-based web application framework that follows the model-view-controller (MVC) architecture paradigm. Django was developed for a fast-paced news site-based website initially, and its primary goal is to make the building of complex, database-driven websites easier. [18]

Django is an open-source web framework that heavily relies on Python to produce files, settings, and data models. It is built to meet two key challenges: highly skilled Web developers’ stringent standards and a newsroom's tight deadlines.
2.8 State of the Art - Popular Online Learning Platforms

2.8.1 Repl - replit.com

Repl. It is an online coding platform that lets users write code, execute and host apps on their platform.

The Replit platform is hosted on Google Cloud. The Google Cloud load balancer scales up and down the servers based on the demand. Replit uses these server instances to create sandboxes to provide users with playgrounds. The code executed by the user is sent to the sandbox. The sandbox compiles the code, and the output is transmitted to the user from the sandbox.

Replit has two personal plans for its customers. The plan provides the user with basic storage and memory. To upgrade to higher specs, users can pay for premium accounts which start at 7 USD per month.

Replit also has the functionality to run flask web apps in the embedded browser:

![Replit IDE running a 'hello world' Flask web app.](image)

Figure 2.3: Screenshot of the Replit IDE running a ‘hello world’ Flask web app.
2.8.2 CodeSandbox - codesandbox.io

CodeSandbox is an online playground for React and other Javascript-based web frameworks. CodeSandbox provides the users with an embedded browser in their platform where they can run and test their web applications.

CodeSandbox has a similar pricing model as Replit, with the premium plans starting at 7 USD per month for higher specs for the sandboxes.

To be noted, the platform cannot be used to create web applications using Pythonic frameworks such as Flask and Django as CodeSandbox only supports Javascript and thus can only run Javascript-based web frameworks such as React Js and Angular Js. Therefore, there is no functionality to run Flask apps on CodeSandbox.

Figure 2.4: Screenshot of the CodeSandbox IDE running React Js web app
Chapter 3 - Design

The current online programming learning path introduces the basic concepts of programming through Scratch (section 2.2) using their block-based programming language. After gaining familiarity with Scratch, the user moves to the Pytch platform (section 2.6), which provides a transitional step before moving to the text-based programming world; this step helps make the transition smoother and less challenging for the users. This paper attempts to continue this learning idiom and describes a learning environment that would serve as an intermediary step between Python (a text-based environment) and developing real-world web applications with Flask. In this section, the design for such a platform is described in detail.

‘Flask from Scratch’ is an entirely browser-based platform that lets users create and run basic Python Flask web apps. The platform allows users to do simple Flask development without having to do any local setup or leave the browser environment by embedding a Javascript implementation of Python into the frontend web application. Resources to learn Flask are also provided to the users using guided tutorials and documentation.

Considerable research was carried out before the design decision was taken to teach the Python Flask framework on an entirely browser-based platform. Other options were weighed in as well as we shall discuss in this chapter.

3.1 Why choose a Python-based framework?

A thought might arise now to question why we should even bother implementing a Python-based browser IDE which would require an extra step to transpile the Python code to Javascript code (as discussed in Coding in Browser - section 2.4) and why not just implement a Javascript-based browser IDE. To answer this question, we can elaborate on the benefits of learning Python, especially as a beginner; and also why learn web application framework using the Flask framework.
The Python language was carefully chosen based on previous research, particularly a review that examined the different languages to propose the most idiomatic way of writing source code [4]. It concluded that learning programming through Python is relatively easier with its syntax resembling English and indentation to make reading convenient. Python was created with the design philosophy entitled, "There's only one way to do it, and that's why it works."

Python serves as a great tool to teach programming using proper design and coding practices as it strongly embraces the ‘Pythonic’ culture of coding, according to the paper “On the usage of pythonic idioms” [5]. The “Zen of Python” [24] sets out 19 high-level principles that guide Python users on how to write syntactically good code. Thus, the design of the language as a whole emphasises the importance of best practices of programming as well as the proper use of object-oriented programming. This ensures that the beginners learn the correct fundamentals of programming from the start and do not develop any bad practices.

Additionally, as this platform seeks to extend the learning path that initiates with Scratch (section 2.2) and moves to Pytch (section 2.6), it should follow the same Pythonic paradigm as they do and should not introduce a new language for a beginner to learn on that path.

### 3.2 Introducing Web Application Development with the Flask Web Framework

For the reasons discussed in the previous section, the platform intends to use Python as the programming language. Python has numerous web application development frameworks:

1. Flask ([https://flask.palletsprojects.com/](https://flask.palletsprojects.com/))
5. Tornado ([https://www.tornadoweb.org/](https://www.tornadoweb.org/))
Among these frameworks, Flask and Django are the most popular by far compared to the rest. Teaching popular frameworks that could be taken up by the users easily in solving real-world issues is a central objective for this project, and hence, Flask was chosen as the framework.

Though we also have Django, another very popular Pythonic web application framework (as described in section 2.7.2), Flask was determined to be a better framework for beginners to learn the web app development paradigm based on the research. This was the case as Flask is a lightweight framework and this enables beginners to break the ice quickly as they do not have to learn too many complex concepts to get started using vanilla Flask. And in case they do want to add more complex features, Flask apps can be easily expanded with extensions.

A comparative study [25] was carried out for the two Python frameworks, Flask and Django. The study documented the journey of developing the same simple web application in both frameworks. The study argued that both the frameworks were production-ready and they both essentially fulfilled the same goal. The study concluded that “the most significant advantages of Flask were that it provides simplicity, flexibility, fine-grained control and quick and easy to learn. On the other hand, Django was easy to work with because of its extensive features and support for libraries. Another main advantage of Django is its scalability. It is the best fit for a large-scale application.” [25].

Another source also concluded that “Flask is considered more Pythonic than the Django web framework because in common situations the equivalent Flask web application is more explicit. Flask is also easy to get started with as a beginner because there is little boilerplate code for getting a simple app up and running” [14].

Based on the above evidences, it was evident that Flask would provide a better learning curve as compared to Django to teach inexperienced users the basics of developing simple web applications as beginners.
3.3 Entirely Browser-Based

‘Flask from Scratch’ integrates various web technologies to provide an entirely browser-based environment for Python Flask development.

The browser would handle all compilation and execution of the code, and the user would not need to send any data to a backend server - thus potentially eliminating a handful of issues relating to Scalability and Security, as discussed below.

3.3.1 Scalability

The general approach taken by other online IDEs depend of backend servers with a control flow like this:

- When the user clicks the ‘Build’/’Run’ command, the IDE grabs the code from the platform’s code editor and sends it to a backend server.
- The code is compiled and executed in the backend server, and the output is returned to the user’s browser where it gets rendered.

Though this approach has its advantages by making the overall development process considerably simpler and allowing for more functionality in the IDE, it introduces scalability issues. As the audience for the product increases, more backend servers would need to be allotted for handling more requests and provide extra computational power for processing more code in times of peak usage. Also, additional database storage functionalities would be needed as the number of projects would grow proportionally to the growing number of users.

On top of the extra workload of developing these backend servers, considerable effort and production costs would be needed for maintaining and managing the servers to serve growing users all around the world.

The current popular online IDEs (Replit - section 2.8.1, CodeSandbox - section 2.8.2) fund their infrastructure by charging their users under their “premium” plans. As this research project intends to create an accessible product for all, the product would be designed to avoid infrastructural costs as much as possible.
The backend server scalability issue can be avoided in all by simply not having them, and instead, developing the product with a fully browser-based design by compiling and executing the code and storing the projects all locally in the browser.

### 3.3.2 Security

Security and privacy issues can be caused if user data (in the form of user’s code) is sent to other systems (say from the browser to backend servers). Handling these issues would complicate the developing process and the infrastructure as the data would need to be encrypted before it is transferred and stored in backend databases. Also, considerable effort would be needed to ensure compliance with strict privacy laws in some states such as the General Data Protection Regulations (GDPR) in the European Union.

Similarly to the scalability issue, the security and privacy issue could be avoided in full if no data is ever retrieved from the user when all actions are handled in the browser exclusively. Also, as this product will be targeted toward the younger generation, we intend to be appealing to the parents who are highly concerned and conscious of companies using their children’s data. Hence, the concerned people would feel more comfortable with this platform as it does not retrieve any data from its users.

### 3.3.3 Alternative Approaches Considered

Instead of hosting the platform on a browser, these issues can also be avoided by simply shipping a downloadable software platform. However, this approach requires installing an extra software in the learning process which would be ironic as one essential design principle for this product is to enable users to access the platform with no installations required (as discussed in section 1.3). This could also cause compatibility issues with the range of operating systems in the market.

As the target group for this product is the younger student group in schools, installing any softwares in school systems could be problematic as certain school networks
are operated at the regional level that prohibits external softwares if not pre-approved as found in a study by Koelin et al [6]. Thus, teachers cannot incorporate new tools and platforms at a school level independently thus causing bureaucratic delays. Hosting the platform on a website would avoid this problem on the whole as no installations would be required - not even Python or the Flask library would be needed to be installed locally. Thus, the users would be able to easily access the platform through any browser on any machine.

3.4 Why Do We Need (Another) Platform?

As discussed in the State of the Art section, the currently popular online IDEs that support web application development include Replit and CodeSandbox.

Based on the product research discussed so far in this chapter, neither of these two platforms meets the design requirements of our proposed platform. Though both the platforms support web applications, neither of them has functionality for guided tutorials to teach the workings of web app frameworks. The platforms merely provide the users with sandboxes to code in an online IDE, rather than serving as learning platforms for inexperienced users.

CodeSandbox can only support Javascript, thus it can only run Javascript-based web frameworks such as React Js and Angular Js. As we want a platform that supports Flask, a Pythonic web application framework, CodeSandbox cannot be used for that purpose. To be noted though, Replit does support the Flask framework. Replit can be potentially used as the next step in the learning path. After users are comfortable working with Flask and their web app development concepts are clear through our proposed platforms, users can move to Replit to create real-world Flask applications without any guided help.

Additionally, as both of these platforms process the user code in their backend servers, scalability and security issues are introduced (as discussed in section 3.3). This results in high operational costs which ultimately leads to the organisation limiting the user’s capabilities unless subscribed to a premium version. This would be
against our platform’s vision as we want a product that is highly accessible to all for free.

3.5 Flask From Scratch Design

This complete browser-based platform is developed by extending the light open-source transpiler called Skulpt (section 2.4) that transpiles the user-provided Python code to Javascript code. Then, this transpiled Javascript code could be efficiently interpreted by the browser’s Javascript engine (Coding in Browsers - Section 2.4). Finally, The output is then rendered to the user. Thus, providing a full browser-based online IDE to the users.

The design of this platform became complex due to the limitations faced on both sides of the development process. On one side, we wanted a serverless platform, hence we couldn’t process the Python code on the backend and we could only use the browser engine. On the other hand, the functionality of the browser was limited as it does not provide a Python runtime.
How a typical Flask server works (Components: Server and User Browser)

![Diagram of Flask server communication flow]

**Figure 3.1**: Communication Flow of a typical Flask server and user browser.

This diagram demonstrates how a typical Flask server works. Flask app server runs on a server machine that listens for HTTP requests on a specific port for that host URL. The Flask app uses the Python runtime and the Flask library installed in the server's system. The user accesses the Flask web app through the browser which sends an HTTP request to the Flask server. Flask server responds with an HTTP response, which typically consists of the HTML page that the user requested.
How the ‘Flask from Scratch’ Platform Works
(Components: Only User’s Browser)

Figure 3.2: Working of the fully-browser based Flask for the platform.

And this diagram summarises the high-level design of the working of the ‘Flask from Scratch’ platform.

Control Flow: When the user attempts to run their Flask app on the platform, the user’s code is retrieved from the code editor. The Skulpt runtime transpiles the Python code into Javascript compiled code. The embedded browser uses the Javascript compiled code to render the appropriate Flask HTML page based on the URL route requested by the user in the embedded browser’s URL bar.

The overall design process for the project is divided into two fairly isolated phases. Multiple alternative approaches were considered for both phases before determining the current approach. The two major phases are:

Phase I: Adding the Flask library in the Skulpt engine to provide Flask framework functionalities to Skulpt's Python runtime in the browser.
**Phase II:** Designing the User Interface/User Experience (UI/UX) of the product and implementing the web application using the React JS framework and the TypeScript programming language.

The alternative approaches would be discussed at the end of each phase individually where the arguments to not undertake those paths are discussed.

**Phase I - Flask Library in Skulpt**

**Detailed Design**

Skulpt system ([section 2.5](#)) provides the in-browser Python runtime that is used to execute the Python code on the browser code. Though the current version of Skulpt follows the Python 3.7 syntax and grammar, it does not provide all the functionalities of Python 3.7. Numerous core libraries are missing from the implementation currently including the ‘sys’ and ‘os’ library modules. However, new library modules are constantly being implemented for the Skulpt system. As the Skulpt system is limited in its capabilities as it is restricted to the browser runtime, certain libraries are needed to be implemented in alternative ways, diverting from their actual implementations in the Python stable releases. The capabilities are so limited to the point that it is not even plausible to implement various core libraries due to the limitations of the browser runtime.

The current Skulpt repository has no implementation of the Flask library. Hence, the first phase of this project attempted a basic implementation of the Flask library in Skulpt from the ground up. The implementation aimed to provide the majority of the essential core functionalities of the actual Flask library ([section 2.7.1](#)) with the intention that the users would be able to create a diverse range of web applications using the replicated Flask API in the Skulpt engine.

This library would simulate the real Flask library giving the user the impression that an actual server is running in the background which is responding to all the “requests” the user makes in the web app in the browser. While in reality, the route
functions are implemented by tapping into the Skulpt compiled code and keeping track of the state; the Skulpt Flask library is implemented according to this design paradigm.

Though this research is initiated with the consideration that the fully-browser Flask experience might not be possible, this design decision was a huge success - we were successfully able to give the users the real Flask-like experience on the browser using Skulpt. Currently, it supports the following fundamental Flask functionalities. The Flask library in Skulpt can be extended further to add any missing functionality.

**Feature Set of the Skulpt-based Flask library**

1. **Flask app constructor** function to instantiate and set the name of the app.

2. **route() decorator function**: To let users specify which URL/s should invoke a given view function. (Note: Each URL route pattern in a flask app is linked to a view function as will be discussed later in the implementation section)

3. Create **routing architecture** such that users can navigate through the different pages of the application. Also, add functionality to directly navigate to a specific page by searching for it in the embedded browser’s URL bar.

4. **Variable Rules**: Provide functionality to let users create URLs dynamically by including variable values into the URL rule parameter. For example:

```python
@app.route('/user/<name>/<int:id>/<float:weight>')
def user(name, id, weight):
    return f'User {name} with ID {id} has weight {weight}'
```
5. Enable users to build URLs dynamically using the `url_for()` method and also allow functionality to redirect to other URLs using the `redirect()` method.

```python
@app.route('/user/<name>')
def hello_user(name):
    if name == 'admin':
        return redirect(url_for('hello_admin'))
    else:
        return redirect(url_for('hello_guest', guest = name))
```

6. **HTTP protocol methods**: Allow applications to accept GET (default request type) and POST (to deal with form data submissions) requests.

7. **Templating**: Flask uses the jinja2 template engine (section 2.7.1). Jinja2 allows users to pass the HTML with special placeholders which use Python-like syntax. The engine then dynamically generates the relevant HTML page with the parameters passed along with it. We would like to include jinja2 or at least a basic implementation of this templating engine in the Skulpt-based Flask library as well.

8. Passing form submission data using the Request object for POST and GET requests.

9. Functionality to let the application redirect to the relevant error pages based on the error code encountered by the application.

10. **Sessions**: Provide the users with a dictionary that is saved on the client-side. Users could use it as a basic client-side database-like entity, enabling them to create a wide range of web applications.

**Alternative Approaches Considered**

**Alternative Approach I**

Another approach that was considered was to add the actual Flask repository to the Skulpt system and implement the libraries that Flask depends on and were missing from the Skulpt system.
I wrote a small script (below) to list out all the dependencies of the Flask library:

```python
from modulefinder import ModuleFinder
moduleFind = ModuleFinder()
moduleFind.run_script("flask_app.py")
for module_name, module in moduleFind.modules.items():
    print(module_name)
```

The script listed out all the 400+ libraries that the Flask implementation depended on. After comparing them to the Skulpt libraries, it was concluded that most of the libraries listed were not implemented in Skulpt. Implementing those libraries to use the actual Flask library would have been infeasible for the scope of this thesis project and implementing some dependencies were even presumably implausible.

**Alternative Approach II**

Another approach could have been to remove the certain dependencies of the Flask library (instead of implementing them as considered above) for Flask functionalities we did not need for this platform. However, the Flask repository is highly complex and stripping away one library at a time from the 400+ libraries that it depends on would not be feasible in this project's scope.

Additionally, working on network port forwarding would be complex as now we have to work only through the browser instead of the native system.

Hence, due to the limitations of the Skulpt engine currently, it is not feasible for this research's scope to go along with either of these alternative approaches.

**Phase II - ‘Flask from Scratch’ Integrated Development Environment (IDE) Web Application**

**Design of the Platform’s Web Application**

The platform’s web application is designed using React Js.
Home Page

- Short summary of the platform’s goal.
- Display sample tutorials on the dashboard for quick access.
- Display contact details of the developer.

Tutorial Page

- List out a number of sample tutorials for the user to follow.
- The tutorials are to be aimed at beginners and the goal is to leave them with a moderate level of experience with Flask once they have completed all the tutorials.

Project List Page

- Lists out all the current and past projects of the user in that browser.
- The projects would be saved at the client-side using IndexDB to avoid sending any user data to back-end servers.
- Projects would be in storage even if the browser or the system has restarted.

IDE Page

- An Ace Code Editor instance is rendered at the top left side where the user could input their code with syntax highlighting, code folding, highlighting matching parenthesis, indentation support and many more features.
- When the user clicks the 'Run' button, the code from the Ace Code Editor is retrieved. The Skulpt engine transpiles the Python code to Javascript code using the new Flask library in Skulpt (described in the previous section).
- The compiled Javascript code is ported to the embedded browser component at the top right of the IDE. The relevant route function is executed based on the URL route inputted in the embedded browser’s URL bar (if no route is present, the default route ‘/’ is automatically inputted there).
- The executed route function returns the HTML code. This is processed through the Jinja JS tool to dynamically insert data into the HTML code.

- The final HTML is rendered on the embedded browser's window. The embedded browser's “URL bar” can be used to call different app routes defined by the user. In case of errors in the route function code, the embedded browser renders the error in a clean way to the user.

- At the bottom of the screen, there is a console component. The console will notify the user of any errors their Flask web app encounters and also displays the print messages from the code.

**User Interface / User Experience**

![User Interface Mockup](https://www.flaskfromscratch.org/ide/42)

**Code Editor**

This is where the user inputs the code they want to execute.

**Embedded Browser**

Preview of the web app created by the user

**Console**

Prints the error and the console messages.

Figure 3.3: Initial User interface Mockup for the Integrated Development Environment.
Figure 3.4: Final User interface for the IDE.

Figure 3.5: ‘Flask from Scratch’ Home Page User Interface Design.
3.6 Tutorials

The platform is designed to introduce Flask web app development concepts to inexperienced users. One engaging way to teach these concepts to the users is through tutorials to guide them through the Flask web app development process one concept at a time. Research by Kölling, M et al [20] on learning platforms discusses the lack of time and support for the training of computer science teachers and the development of teaching materials at secondary schools in the UK. The study concludes that there is a need for adding explicit additional teaching support through tutorials and teaching documentation in the learning platforms.

The users can follow guided walkthroughs to understand each unit of the tutorial independently with a low cognitive load.
The tutorials have been designed with the following principles in mind:

- Acknowledge the programming concepts the users are already familiar with such that the platform can build on top of that.
- Design a suitable project for an inexperienced Flask user.
- New Flasks concepts that the platform aspires to teach to the users.
- Teach distinct Flask concepts in small steps to avoid a cognitive overload on the user (as discussed in section 2.1.2).
- Teaching the students Flask in sufficient depth for them to be able to use the real Flask framework following their training on this platform.

New tutorials can be easily added. Steps to do so are mentioned in detail in the setting up tutorial section later.

Here is a list of tutorials that have been already implemented to demonstrate the platform’s capabilities and mechanics to provide guided tutorials. These tutorials have been developed based on the TutorialsPoint’s Flask page [26].
Tutorial #1 - Hello World with Flask!

This tutorial will introduce the users to the Flask web application framework by helping them create the 'Hello World' Flask application.

Tutorial #2 - URL Routing in Flask

This tutorial will explain to the users how routing works in the Flask web applications. Routing refers to the mapping of the URLs to the specific view functions (which handle the logic for that URL).

Tutorial #3 - HTTP Methods

This tutorial will teach the users how the Hypertext Transfer Protocol (HTTP) methods work in the Flask web application architecture.

Tutorial #4 - Variable URL Rules

This tutorial will teach users how URLs can be created dynamically by adding variable parts to the URL rule parameters.

Tutorial #5 - Throwing Errors!

This tutorial demonstrates how to throw errors in the Flask web app using the Flask library’s abort() function.
Chapter 4 - Implementation

4.1 Phase I - Flask Library in Skulpt

In order to add a new module for Flask in the Skulpt system, a new folder was created in the 'src/lib' folder of the up-to-date forked repo of Skulpt. A new version of Skulpt can be compiled by executing the command “npm run dist” in the Skulpt folder directory to generate updated Skulpt js files with the new Flask library in it.

A new Flask class was created in this folder with all the relevant attributes and methods to provide a basic implementation of Flask. The code is available at: https://github.com/krishanu-dey/skulpt/blob/master/src/lib/flask/

The important functionalities of the implemented Flask library are discussed below:

4.1.1 Running the App

```python
def run(self) - code
```

This method is called by the user to start the Flask server for the application. All the relevant fields for the new session are initialised here. The method prints the generic Flask server starting message to the user to convey to the user that the Flask code was successfully compiled and that the application can be accessed on the embedded browser now.

4.1.2 Mapping URL Routes to their respective View Functions

```python
def route (self, route, **kwargs) - code
```

The route() function is used as a decorator function to assign the relevant passed route URL to the function. They are set on the line above the function they are supposed to execute. Implementing this function required a deep dive into how Python decorators work. Essentially, the decorator functions serve as logic that wraps other functions and extends the behaviour of the latter function without...
explicitly modifying it. For example, here the view function `about()` will be executed when the user visits the route “aboutus/” on the embedded browser.

```python
@app.route('aboutus/')
def about():
    return 'Hi, this is Kris'
```

I created a `routeTable` dictionary to map the relevant URLs to their view functions with their methods using a `RouteTableEntry` class.

```python
self.routingTable[route] = RouteTableEntry(view_function, methods)
class RouteTableEntry:
    def __init__(self, func, methods):
        self.func = func
        self.methods = methods
```

Now, if the user attempts to visit a particular URL route in the embedded browser, the view function for the URL route is fetched from the `routingTable` dictionary and executed dynamically. The method parameter determines if the route accepts GET and/or POST requests.

Additionally, new URL routes can be added to the view functions by using the `add_url_route` method:

```python
def add_url_route(self, route, endpoint, view_function, **kwargs)
```

4.1.3 Handling Routes - including Dynamic Routes with variables

```python
def handleRoute(self, routeInput, requestData)
```

This function is called by the web app (which serves as the front-end of the platform) to render the relevant route URL. The function accepts the route that needs to be rendered as well as the request message which consists of user submitted information (discussed later in depth in section 4.1.5).

Flask supports certain variable rules to allow creating dynamic URL routing by adding parametric variable values in the routing rule. For example:

```python
@app.route('user/<user>/<int:id>/<float:weight>/')
def dynamic_url(user, id, weight):
    return f'User {user} with ID {id} has weight {weight} kg'
```
Here `<user>`, `<int:id>` and `<float:weight>` are all parametric variable values. Types of the parametric variable values have to be mentioned next to them, except for string variables.

Mapping URLs by simple string matching would not work, as for example the URLs inputted for the above route could look like either (i) `user/kris/21/70` or (i) `user/fer/6/65.8`. Thus, the routes need to be dynamically generated and matched to the correct URL route function by comparing the URL pattern.

The handleRoute() function loops through each route rule and compares them with the user’s inputted URL with a special routeMatch() function to determine if the URL follows the correct pattern and also has the correct type as well.

The two URLs (i) and (ii) match the ‘dynamic_url’ as:

<table>
<thead>
<tr>
<th>Route Rule Elements</th>
<th>URL 1 Elements</th>
<th>URL 2 Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>user</td>
<td>user</td>
</tr>
<tr>
<td><code>&lt;user&gt;</code></td>
<td>kris - ✔️ is of type string</td>
<td>fer - ✔️ is of type string</td>
</tr>
<tr>
<td><code>&lt;int:id&gt;</code></td>
<td>21 - ✔️ is of type int</td>
<td>6 - ✔️ is of type int</td>
</tr>
<tr>
<td><code>&lt;float:weight&gt;</code></td>
<td>70 - ✔️ is of type float</td>
<td>65.8 - ✔️ is of type float</td>
</tr>
</tbody>
</table>

Note - When the variable type is not specified, it means it represents a string.

As all the elements of the URLs match, the helper function would return the boolean value ‘True’ with the parameters as a dictionary mapping from the variable name to the value. A pointer to this dictionary is passed when the view function is called to serve as the arguments to the view function.

Code to the helper functions are available at:

4.1.4 Redirecting to other URL Routes Dynamically

```python
def redirect(newRoute) - code
```

The encapsulation and abstraction created by having a RouteTable dictionary of RouteTableEntry elements was useful for extending beyond the basic features of Flask in a simple and clean way.

Users can redirect and render to another route by using the redirect() method. It looks up the relevant URL route in the RoutingTable dictionary to execute the new URL’s view function.

4.1.5 Building URL routes of View functions Dynamically

```python
def url_for(functionName, **kwargs) - code
```

Using the url_for() method, the users can dynamically build the URL of a specific view function using the function name.

For example, here based on the if statement, the application redirects to the other specific routes by using the redirect() and url_for() functions.

```python
@app.route('/profile/<name>')
def profile_page(name):
    if session['login']:
        return redirect( url_for('hello_user'))
    else:
        return redirect( url_for('login', guest = name))
```

4.1.6 The Request message

A new ‘Request’ was implemented to enable the user to pass arguments to functions using HTML form data and to provide them an overview of the HTTP protocol by letting them send either GET or POST requests through the message. For example:

```python
@app.route("/login", methods=["POST", "GET"])
def login():
```
if request.method == "POST":
    print("login here - POST request")
    return f'Hi, {request.form["name"]}'

elif request.method == "GET":
    print("login here - GET request")
    return ""
    "form id="1" action="/login" method="POST">
    <label for="name">First name:</label><br>
    <input type="text" id="name" name="name" value="John">
    <input type="submit" value="Submit">
    "form>
    return "login here - request method not supported."

In the ‘route’ function decorator, the allowed request “methods” for this view function are mentioned. When a certain request is sent to a route, the supported methods are checked in the RoutingTable for that route. In case the request method is not allowed, an error is thrown.

The Request message passes the submitted form data for GET messages as ‘args’, and while for POST messages it is passed as a ‘form’ dictionary. For example, in the code snippet above, under the “POST” request handling, the form data can be accessed using `request.form["name"]`.

4.1.7 Rendering Templates

def render_template(htmlText, **kwargs) - code

It is essential to support this feature as it is a crucial functionality in Flask. Without this feature, the flask applications created by this website would not be able to dynamically generate the HTML and the user might question the effort in learning new frameworks like Flask. Certain features like rendering templates and dynamically generating URLs give web applications the power for which they are popular for.

Flask uses the Jinja2 library in Python to render templates. However, the Skulpt engine does not have this library implemented as well. Instead, there is an
open-source Javascript implementation of Jinja2 which covers all basic and most advanced functionalities of the actual Jinja2 framework [27]. The Flask library in the Skulpt engine passes the unrendered HTML template with the dictionary of parameters to the React frontend app. The React app imports the JinjaJS library and renders the template using the passed parameters from the Skulpt before displaying it to the user in the embedded browser.

### 4.1.8 Throwing Errors

```python
def abort(code) - code
```

Users can make their web applications throw errors in a convenient way using the Flask’s `abort()` function by simply passing just the error code while calling it. The error codes supported are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>Bad Request</td>
</tr>
<tr>
<td>401</td>
<td>Unauthenticated</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
</tr>
<tr>
<td>406</td>
<td>Not Acceptable</td>
</tr>
<tr>
<td>415</td>
<td>Unsupported Media Type</td>
</tr>
<tr>
<td>429</td>
<td>Too Many Requests</td>
</tr>
</tbody>
</table>

### 4.1.9 Unit Testing

As this library is an attempt at a proper implementation of the Flask library in Skulpt, it is essential to test all the functionality to make sure everything works as intended. Unit tests were written to test out all methods and their base and edge cases.

The tests can be found at:

with the dummy flask code it tests available here.

The structure of the unit tests followed the popular ‘Arrange/Act/Assert’ pattern [28]:

**Arrange step:** This step sets up all the test case data.

**Act step:** This step invokes the function to be tested with the parameters from the test data set up in the arrange step.

**Assert step:** Verifies the function acted properly by comparing the actual output and the expected output (from the test data).

### 4.2 Phase II - ‘Flask from Scratch’ IDE React Web Application

The second phase of the implementation consisted of designing the React web application that would serve as the front-end of the platform.

As this product serves to extend the Pytch learning path, it was desired to keep the user experience as close to that of Pytch as possible. Hence, it would make sense to carry the work forward from Pytch’s React web app and tweak it to make it work with Flask. However, the Pytch web app was heavily coupled with the Pytch system for managing concurrent threads, sounds, visuals and other Pytch specific functionalities.

Accordingly, I made attempts to implement ‘Flask from Scratch’ from scratch. Due to numerous features that we wanted in the IDE, it was not feasible to implement all in the project’s time frame. Hence, I moved back to the initial plan and I was able to make the React app run after considerable work and methodically eliminating all Pytch-dependencies. Now, the app could run with the original Skulpt engine (with the new Flask library), instead of Pytch’s version of modified Skulpt. This lets us inherit the features that were already implemented such as resizing windows, project listing page, storing project data on the user’s browser using IndexDB, Ace code editor for coding support and numerous other useful features as mentioned in section 4.2.7 section later.
4.2.1 Embedded Browser

The ‘Stage’ component of the Pytch app was replaced with the ‘Embedded Browser’ component for ‘Flask from Scratch’.

The ‘Embedded Browser’ component consists of the URL bar and the browser window on the right side of the IDE page.

![Embedded Browser Rendering an HTML Login Form](image)

**URL Bar**: The user can input any route in the URL bar. The web application will render the route that the user’s application supports, otherwise it returns “undefined”.

**Browser Window**: The web application renders the HTML that the view function of the requested URL route returned in the browser window of the embedded browser.

Before the HTML is rendered in the embedded browser, it is dynamically modified such that it can behave properly in the restricted environment. For example, if another web app route link, say “/login” is rendered, the web page would refresh and try to access “file:///login” on a new page; thus, it is vital to modify these HTML tags before rendering for the Flask app to work in the embedded browser.
The script `flask-ide.js` dynamically modifies the HTML being rendered such that the web app properly works in the embedded browser.

### 4.2.2 Rendering a URL Route

```javascript
defunction renderPage(url_requested, requestData) - code
```

This function handles the direct connection between the IDE web app and the Flask library in the Skulpt compiled code. This function accepts the requested URL route and the request message data as the parameters.

The function taps into the Skulpt compiled code to call the necessary view function of the requested URL route. The returned data is either just the HTML string or un-templated HTML with a dictionary for the parameters. The function sanitises the HTML to make it work correctly in the IDE app as discussed below.

### 4.2.3 Modifying Links before Rendering

```javascript
defunction fixURL(jsHtmlString) - code
```

The function `fixURL` retrieves the value of the “href” tag of the link tags (i.e. `<a>`) in the HTML and then sets “href” to “javascript:void(0)” so that the page does not redirect out of the IDE.

To let the Flask web app redirect to a different route, a new attribute is set in the link tag. The “onclick” attribute of the link tag is set with the value `renderPage('{} + hrefValue + '{}')`. Now, when the user clicks the link in the embedded browser, instead of redirecting to a new page, the link calls the function `renderPage` (discussed above) with the requested route as the parameter.

### 4.2.4 Handling Form Data

```javascript
defunction fixForms(jsHtmlString) - code
```
We come across a similar issue with forms as we did with links however this time it is a bit more complex as we need to handle the form data as well.

The default form submission is suppressed so that the page does not refresh and creates a GET/POST request to some route. The form’s request method and action are retrieved from the form. If no method is set, by default it is assumed to be a GET method. A dictionary stores and maps the form ID to what HTTP method it uses. The forms with no IDs are automatically set as unique IDs by the flask-ide script. Instead, the form submission is handled by adding a new “onsubmit” attribute to the form tag which calls the “onFormSubmit” function in flask-ide with the form ID and its requested route.

    function onFormSubmit(formID, url_requested)

When the form is submitted, the “onFormSubmit” function fetches all the inputs of the form using the unique form ID and creates the required request message with all the form data. Finally, it calls the renderPage function (discussed in section 4.2.2) with the target URL route and request message as the parameters. Finally, the renderPage function then handles rendering the correct route with the request message by tapping into the Javascript compiled code which contains all the Flask app view functions.

The flask-ide.js helper file handles this for all the links and forms in the HTML page generated by the Flask app dynamically so that it can handle a web app with any number of links or forms. This connects all the pieces to support URL route handling and form submissions inside the IDE’s embedded browser.

### 4.2.5 Catching User Code Errors

When the user attempts to call a view function with a syntax error, the platform’s engine catches the error and displays it to the user in the embedded browser along with an error message in the console.
4.2.6 Setting up Tutorials

The platform has the functionality to create guided tutorials as discussed in the design section before. The steps to create a new tutorial in the platform are as follows:

1. In the tutorial repository of the platform, create a new branch from the first commit of the ‘initial-empty-state’ branch.
2. Create a new folder in the branch for the tutorial content.
3. In the new tutorial folder, create a new file named ‘code.py’ and add the starting code of the tutorial there. Stage just the code file specifically.
4. Commit the staged change with the comment “{base}”. The platform engine looks for the “{base}” comment in each branch to find the starting code. The tutorial will fail without this commit, thus it is important to ensure each tutorial branch has a “{base}” commit.
5. Then, create the summary.md and tutorial.md file in the tutorial folder. Stage and commit the two files. The links to the sample format for each file can be found here: tutorial.md, summary.md and code.py.
6. The tutorial folder has a specific syntax to create the tutorial tile for each step. The platform engine accepts each subheading of the markdown file as a separate step of the tutorial. The code changes in each step can be tracked by referencing the commit comment (as done in the sample tutorial.md file).

The developer can run the platform locally to test the tutorial. If the new tutorial does not show up in the list, perform an “empty cache and hard reload” of the browser.

4.2.7 Pytch Inherited Feature Set Summary

- **Ace Editor** [22] - The platform uses the ‘Ace Editor’ as its code editor. ‘Ace Editor’ is an open-source project that implements a Javascript-based embeddable code editor. It provides various functionalities to the user such as auto-completion, highlighting matching parentheses and syntax highlighting.

- **Browser Storage with IndexDB** [23] - IndexDB provides a transactional database system to store data on the client-side. The platform uses IndexDB to store user project data on the client’s browser.

- **Tutorial Walkthrough Mechanics** - As discussed in section 4.2.6, the developer can easily create guided tutorials on the platform.

- **Multiple Layouts Options** - The user can select between two layout options for the IDE.

- **IDE Window Resizing Functionalities** - The user can resize each window component of the IDE based on their preference.

- **Functionality to Export and Import their Projects to the Platform** - Users can easily export (also import back) their work to continue working on the project outside the platform using the actual Flask framework. Thus, it would provide extensibility to the system so that the users have a clear idea of what is next in their learning path. This fulfills design principle 5 as it provides the user with the potential for continued activity.
Chapter 5 - Evaluation

5.1 Overall Platform Analysis

As stated in the introduction, the objective of this project was to design and implement a fully-browser based learning platform to teach inexperienced users Flask with an intuitive user experience. When the project started, the plausibility of the project was not apparent. Following research into the browser mechanics and the libraries available for web development, many design ideas were considered (as discussed in section 3.5). After many design discussion meetings about the pros and cons of each approach and even experimenting with a few of them, the design decision taken was to implement a new Flask library as an empty module in the Skulpt library with a React JS web app as the front-end.

The implementation approach taken with that design decision was successful as we were able to fulfil the objective for the proposed platform. Overall, the platform can properly function with no functional errors. The platform sends no data to a backend server and exclusively works through the browser engine. The users can create a wide variety of Flask apps, which can be highlighted by the various implemented tutorials and the demo. The platform requires no complex administration setups. It also provides the users with a teaching experience using the guided tutorials. New tutorials to teach new Flask concepts can be added with ease by following the instructions provided in the Setting up Tutorial section.

‘Flask from Scratch’ fulfils the feature set as proposed in the design section for both - the Flask library in Skulpt and the platform IDE front-end. The Flask library in Skulpt and the platform web app can be easily extended to add missing Flask features as part of future work.

5.2 Product Demonstration

Here is a short video tutorial to demonstrate the usage of the new platform.
The video is available at: https://drive.google.com/file/d/14Dpo2kIReXghQPyyvDmMq0PzU5KsOf4LO

The platform can be run locally conveniently as well by cloning the super-project repository ‘https://github.com/krishanu-dey/pytch-releases’ which consists of five subrepositories for all the components of the platform.

Once cloned, please make sure the following packages are installed in the system:
  - node, tmux, git, virtualenv, python3 and realpath

Finally, run the `./develop` script in the super-project’s main folder to initialise each subrepository and generate the local development server script. The platform will run locally with tmux when the server script is run.

5.3 Evaluation based on the Design Goals

As mentioned briefly in the research objective section, specific design goals are formulated based on research by M. Kolling [19, 20] to determine if the research project is successful or not.

DP1: Simplicity of Access and Setup
  The project ensures this principle by providing access to the platform as a web application that is accessible to all on any browser for free. The product design seeks to lower cost increase with potential scalability growth in the future so that the product always remains free to all.

DP2: Allow the User to Leverage their Existing Knowledge
  The platform accounts for the Python knowledge users have already learned, either independently or through Scratch and Pytch, and helps users to extend on what they have already learned.

DP3: Present a Sense of Authenticity
  Adequate Flask functionality is implemented in the platform to let users create basic Flask web applications seamlessly. Users would be able to use the Flask
knowledge they learn by creating various different types of web applications, thus providing them with a sense of reward for their work. The fact that the user can continue working on their projects (with no changes) locally on real Flask after downloading them highlights the authentic Flask experience users are receiving on this platform.

DP4: Support User Autonomy via Integrated Learning Materials
The platform provides mechanics (as explained in section 4.2.6) in the front-end to create tutorials and guided walkthroughs to aid the user’s learning process.

DP5: Potential for continued activity (e.g. leading on to Flask)
The platform allows users to download their Flask projects so that they can continue developing the app on their local machines with the real Flask library as they extend on what they learned. This principle also ensured the user’s projects are runnable using the actual Python Flask library without any code changes required. Thus, ensuring they do not learn any wrong concepts.

5.4 Further Evaluation Needed using User Case Studies
Following the successful design and implementation of the minimal viable product for the Flask learning platform, ‘Flask from Scratch’, it is essential to properly evaluate the platform through user studies with the targeted user sample. The sample would ideally include inexperienced Flask users familiar with the text-based programming world using Python, preferably through successful training in Scratch and Pytch. A user study would provide crucial feedback to improve the overall user experience and find potential errors in the platform implementation.

Due to the time constraints, conducting a user study was out of the scope of this research project. However, I would recommend conducting user studies in case the platform is further developed in any future efforts.
Chapter 6 - Conclusion

This paper is the first effort in implementing a novel research-based learning platform dedicated to teaching Flask web application development that is entirely browser-based, such that no data is retrieved from the client. This platform will add to the range of learning programming platforms for beginners and will be the first one to provide dedicated guided support to teach web application development with Flask to inexperienced users, specifically extending the learning path created with Scratch and Pytch.

6.1 Future Work

The project's focus was initially to develop a minimum viable product to investigate the plausibility of creating the proposed platform with the strict restrictions that were put on it. Upon successfully implementing the MVP, I proceeded to incorporate additional Flask features into the platform and also worked on the front end to improve the overall user experience. Though numerous features were added, as shown in the video demonstration in section 5.2, there are plenty of features I would have liked to add to develop my envisioned version of the production-ready platform for ‘Flask for Scratch’.

Firstly, if I had two extra months on this research, I would have attempted to conduct user case studies with the targeted audience - which would mainly consist of people familiar with text-based coding but not web app development. The user feedback would provide useful insight before proceeding with any further development as the product still has not been tested on a single member of the target audience yet. This would help fix any fundamental issues of the product before it becomes more complex in the future.

If I had even more time to work on the project, I would add more features to the Flask library and the IDE to provide the user with a richer and a fuller Flask-like experience by adding functionalities that are missing in the platform but are provided
by the actual Flask framework. A high priority functionality that could be added next is a file structure in the IDE to let users add HTML templates and CSS styles as separate files, instead of having them inlined in the code file. This Flask feature is explained in more detail here.

Additional efforts can be made to display the error to users in a more helpful manner to improve the platform’s user experience. Currently, the errors are displayed in the embedded browser as described in section 4.2.5; however, they should be shown in the error tab at the bottom such that it provides details on how the error got triggered and also a link to the specific line that raised the error.

6.2 Overall Research Reflection

As this research comes to a conclusion, I think it is essential to reflect on the work produced by this research based on the knowledge gained over the last academic year.

The process of converting the preliminary idea of creating an online learning platform to a fully-working online IDE with guided tutorials for teaching web application development using Flask involved a process of in-depth research on the literature on computer science teaching environments and the current online IDE state of the art initially, and then an intensive development period to design and implement the final platform.

From the knowledge that I gained from the research conducted and the development efforts of this new platform, it is certain that such an environment is necessary to assist inexperienced users in web application development as there are no other resources available currently, as discussed in the state of the art section, that offer a similar experience as this platform.

I believe that the project has great potential, and further efforts should be put into it, as discussed in the future work section. The platform is easily customisable and extensible to add new features in the online IDE and the Flask library to provide an enhanced experience based on user feedback and further research. The current
implementation of the platform should serve as the foundation for future efforts to create a production-ready learning platform for teaching web applications development with Flask.
Chapter 7 - Acknowledgements

I would like to acknowledge and thank my supervisor, Dr Glenn Strong, who was a vital part of the project’s success. Dr Strong was always supportive of my work and was willing to organise a meeting to help me whenever I was stuck. I would also like to express my gratitude to Dr Benjamin North for his support with the design and implementation of the project. Without their continued assistance and guidance, this research would not have come to fruition.

Finally, I would like to thank my parents and friends for supporting my development throughout my years at university.
Chapter 8 - Bibliography


