UNIVERSITÀ
DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Interactive Code Playgrounds

A slides system to make University programming classes more active and inclusive

Supervisors
Lorenzo Angeli
Maurizio Marchese

Student
Luca De Menego

Academic year 2022/2023

Acknowledgements

I would like to take this opportunity to express my deepest gratitude to the individuals who have contributed to the completion of this dissertation, making this journey a memorable experience. First and foremost, I am grateful to my supervisor, Lorenzo Angeli, for his support and guidance. His genuine passion for the subject has truly inspired me. Thanks should also go to Professor Maurizio Marchese, for his valuable feedback and contributions. Finally, to my family, who has been a constant source of love and support throughout this journey, thanks for your sacrifices, understanding, and patience.

Contents

Abstract

This dissertation explores the potential of Interactive Code Playgrounds (ICPs) as a tool for enhancing University programming education, particularly introductory courses. It promotes an engaging, inclusive and active learning environment, consisting of slides with execution capabilities and exercises. Through a series of interventions conducted in introductory programming courses, the effectiveness and limitations of this tool were evaluated. The findings revealed that while certain programming languages and browser compatibility issues exist, students appreciated the interactivity and user-friendliness of ICPs, and found the resulting pedagogy more productive. The research lays the groundwork for future advancements in University programming education through the refinement of ICPs.

1 Executive Summary

1.1 Problem Statement

As digitalization becomes more complex, the proliferation of programming languages and development environments has led to an increasingly higher systemic fragility. This presents a significant challenge in providing a standardized and accessible environment to students in the University programming education setting, especially when considering the need for compatibility in outdated computers or offline environments.

Existing solutions developed to tackle these challenges primarily rely on Virtual Machines (VMs), Docker Containers, or in-browser cloud environments. Each of these alternatives comes with its own set of trade-offs. VMs demand high computational power, Docker or similar technologies may be hard to install, and browser environments typically rely on a persistent internet connection.

This dissertation aims to address these challenges by introducing and evaluating the effectiveness of Interactive Code Playgrounds (ICPs). ICPs represent an alternative didactic material combining interactive slides, code execution capabilities and exercises. This new form of slides promotes active learning, and being entirely browser-based, it facilitates the installation process and tries to reduce technical difficulties. Furthermore, ICPs-based material can be distributed as single-file web servers that can be run on Windows, MacOS and Linux. This feature reduces the reliance on constant internet connectivity, making ICPs suitable for offline usage.

1.2 Development

The ICPs project is composed of several modules, each designed to provide a specific feature. The most important ones are `icp-bundle`, `icp-create-server` and `icp-editor`.

`icp-bundle` is a plugin that offers code playgrounds exposed as web components. It supports various programming languages, including Javascript/Typescript, Python, Java, Processing, StandardML, C/C++ and SQL. The module was developed using the Svelte framework, and it leverages the power of `WebWorkers` or `SharedWorkers` to handle resource-intensive tasks such as code compilation and execution. The communication pattern used to pass messages between components in a child-parent relation is based on `CustomEvents`, capable of propagating across Shadow DOM boundaries. The code editor is built on CodeMirror 6, and it has been customized with plugins to fulfill our requirements. The build process relies on Vite, Gulp and a set of configuration files that allow to maintain modular outputs. The resulting files can be imported and used within web pages, allowing the creation of interactive code playgrounds using custom HTML tags such as `<python-editor />`.

The `icp-create-server` project automates the creation of single-file distributable web servers, which are essential for providing students with didactic material that can be used offline. The base file we start with is an executable ZIP archive that contains all the resources that will be served on `localhost` when executed, and this project simply inserts into this archive an `index.html` file containing the slides. To achieve this functionality in Javascript, we developed a script that operates at byte-level and adheres to the ZIP standard. The resulting NPM package exposes a function that takes a hexadecimal representation of the original ZIP file and the HTML content of the slides as input, and returns a `Uint8Array` with the updated ZIP archive.

The `icp-editor` module is a user-friendly *"What You See is What You Get"* (WYSIWYG) editor that enables the creation of slides compatible with ICPs. It uses Reveal.js as the underlying presentation framework, and it provides a visual interface for designing the slides. The module is built with Svelte, and it uses Vite for efficient development and bundling. The editor is publicly served on Github Pages, allowing users to easily access it.

Together, these modules form the foundation of the ICPs project, allowing to create interactive learning material for programming courses.

1.3 Interventions

The interventions conducted to evaluate the effectiveness of ICPs consisted of a pilot study in July 2022 and the main research activities carried out during the second semester, starting from February 2023. The research employs a mixed methods approach, combining qualitative and quantitative analysis techniques. The interventions involved the integration of ICPs into selected lectures of introductory programming courses.

To gather feedback and insights from students, two surveys were administered, which aimed to collect qualitative data through open-ended questions and quantitative data through Likert scale-based questions. In addition to surveys, participant field observations were used as an additional form of data collection. This allowed to triangulate results, ensuring a comprehensive understanding of students' experiences.

The data analysis process involved several steps to transform raw data into meaningful insights. The collected data was cleaned, organized, and examined to identify patterns, trends, and conclusions. For the qualitative data, we employed a coding process to categorize responses to open-ended questions and field notes. These codes were then grouped into broader themes. The quantitative data, derived from Likert scale-based questions, was converted into numerical values, grouped into categories and organized through a series of measurements of central tendency, namely average, median and mode. While we employed some descriptive and inferential statistics to a certain extent, the analysis was primarily qualitative.

1.4 Experimental Contexts

This dissertation primarily focuses on the three interventions conducted during the second semester, as the first pilot study was mainly conducted by my supervisor. As a result, the experimental contexts of this research involved three introductory programming courses at the University of Trento. These courses were from different disciplines, namely sociology, psychology, and computer science. A total of 95 students participated in these courses, and the majority of them were beginners with a high level of motivation to learn programming.

The classes were conducted in a setting that provided a persistent and high-speed internet connection. During classes, students had access to their own personal computers or the computers provided by the University. Prior to the introduction of ICPs, approximately 17% of students reported difficulties installing the Integrated Development Environment (IDE) required for the course.

During the interventions, the majority of the students opted to use the proposed alternative didactic material based on ICPs, and a small minority chose to stick with the original didactic material in PDF format.

1.5 Key Findings

The key findings of this research highlight both the strengths and weaknesses of ICPs as an alternative didactic material in programming education.

One of the main challenges encountered with ICPs was the compatibility with certain programming languages. Some languages, such as Processing or Java, exhibited issues related to incorrect outputs. Furthermore, while ICPs functioned effectively in Google Chrome and Firefox, compatibility issues were observed with other web browsers. As a last shortcoming, students complained about some missing features, such as the possibility to take notes directly on the slides or navigate to a slide by entering its number.

Despite these limitations, students expressed a preference for interactive material, as also proved by the study tools they used. They found ICPs to be more engaging and productive compared to traditional lecture materials, and its interactive nature allowed them to better understand programming concepts and code snippets. Additionally, ICPs were found to be generally more familiar than the course's IDE, and the slides quality was positively evaluated.

Although some technical issues were encountered by students when using ICPs, they reported fewer problems compared to the course's IDE. Most of the issues students faced with ICPs were related to its limited compatibility with some programming languages, suggesting that investing more development

time to address these language-specific challenges could further improve the tool's effectiveness.

1.6 Final Considerations

This dissertation serves as a stepping stone towards the advancement of University programming education, and it aims to create a more effective and engaging learning environment that gives importance to accessibility and inclusiveness. We acknowledge that this research did not comprehensively evaluate the accessibility of ICPs or their effectiveness in challenging contexts with restricted internet access or outdated hardware. Nonetheless, we believe it provides valuable insights into the strengths, limitations, and potential improvements of ICPs. In the selected educational settings the findings highlight some bugs and compatibility issues, but also a positive reception of ICPs and a more pleasing resulting learning experience.

2 Introduction

2.1 Background Information

The process of digitalization has transformed many aspects of our lives, changing the way we live, work and learn. In the field of education, digitalization has opened up new opportunities for students to access knowledge and learn in innovative ways. At the same time, however, this process has created various kinds of issues that need to be addressed.

From a **technical** perspective, digitalization is becoming increasingly complex, fragile and heterogeneous. As more systems are developed, making them interact correctly with each other becomes harder. In education, this issue is particularly evident in the field of programming, where the proliferation of programming languages and development environments has led to an increasing systemic fragility. This poses a challenge to students, who would instead need platform-independent tools and easier set-up processes to learn effectively.

Another issue that arises from digitalization is **pedagogical**. In education, it is essential to provide students technologies and tools that look and behave the same on all environments. However, this can be complex to achieve in today's digital landscape, where there are multiple devices, operating systems, and software versions. This issue becomes particularly important in a blended-learning setting, since in an online environment it can be challenging to troubleshoot differences among Operating Systems (OSs) and devices. Furthermore, in the field of computer science, where students need to use complex development environments, providing a consistent learning experience is tough.

Socio-technical issues also arise from digitalization. The assumption that everyone has access to a persistent and high-speed internet connection is misplaced. In reality, not everyone has the same level of access to technology. This can clearly create a divide between students with different resources, leading to inequalities in education. Additionally, it is also complex to keep up with the accelerated pace of change, that can become overwhelming. This is particularly evident in Higher Education, where the teaching pedagogy has been slow to adapt to new digital opportunities.

Finally, when building digital resources a too often unconsidered feature is accessibility, which means making resources usable by as many people as possible. Providing equitable access to users experiencing visual or hearing impairments is critical, especially in education.

2.2 Goal Setting

To solve the aforementioned issues, this dissertation proposes an alternative didactic material for programming classes, which also integrates a simple development environment.

2.2.1 Research Gap

Technical Aspect

A typical problem instructors face when preparing programming laboratories is the setup of the students' development environment. Students have machines that differ in terms of hardware specification, computational power, operating system, software versions etc. Integrated Development Environments (IDEs), compilers and interpreters may behave differently when used under different settings, and programs working on the instructor's computer may fail to work on other machines.

The proposed solution is platform independent. In fact it uses a web technology, which is inherently compatible with any device having access to a browser. This greatly reduces technical issues, and the time spent on troubleshooting.

Additionally, we expect our solution to be long-lasting. It is a completely front-end technology, requiring no back-end server performing computations. Furthermore, most of the updates regarding internet communication protocols and browser APIs - that our solution relies on - are typically introduced with backward compatibility in mind.

Pedagogical Aspect

A standardized environment is often a critical feature in programming classes. It reduces possible technical issues, while helping instructors and students compare their results. However, most of the current technologies that are based on software run natively on computers appear diversely on different operating systems.

All web browsers are very similar to each other in terms of HTML/CSS interpretation and exposed APIs. This means any web technology, including our solution, looks and behaves the same on all kinds of environments, despite the OS.

Socio-Technical Aspect

One of the most widely used standards for document exchange in education is the Portable Document Format (PDF). PDF files, however, have not been originally designed with accessibility in mind, which means screen readers encounter a number of problems when following their structure. But when talking about programming education, this is not the only problem: most of the development environments currently available - such as Visual Studio Code or Eclipse - have accessibility limitations.

The proposed solution is based on HTML pages, which can be built with a high level of accessibility thanks to features like semantic elements and ARIA attributes. The development environment our solution gives access to is itself web-based, hence it inherits the same level of accessibility.

2.2.2 Proposed Solution

The catchphrase of technologies disrupting education has been around for more than a decade. The status of the use of technologies in Higher Education shows that pedagogy has not really been revolutionized during these years. As a result, we propose a solution that goes against the idea of creating disrupting technologies: Interactive Code Playgrounds (ICPs).

ICPs represent a new slides system, resembling PowerPoint presentation. It is, however, far more interactive. Thanks to the use of web technologies, it employs Javascript and WebAssembly-based (WASM) software to create self-contained development environments accessible within the slides themselves, working locally without communicating with external servers.

To make slides created with this system independent from an internet connection, we propose to distribute them through a single-file distributable web server that can seamlessly run on the most widely used OSs.

2.3 Intervention Context

The main context in which we think ICPs would be useful is introductory programming courses. Students approaching the world of programming are often frightened by the complex set up of a development environment, and the time instructors spend troubleshooting is substantial. Offering a hassle-free experience, at least for the first few lectures, may motivate the students to continue, while allowing them to start coding from the very beginning.

At the same time, ICPs may also boost teaching. The instructor could take advantage of the code playgrounds available in the slides to show how the output of a certain code snippet changes when editing specific sections of the code. This enables a more dynamic approach to pedagogy, and possible on-the-spot small exercises for students. From this specific standpoint, the context in which ICPs could be applied can also be more advanced.

2.4 Significance of the Study

The proposed product focuses around four goals, which also represent its main benefits:

- **Educational Value**: provide a more engaging learning experience for students approaching the world of programming, while helping them develop essential coding skills.
- **Streamlined User Experience**: offer a hassle-free experience, eliminating the need for complicated installations or setups.

- **Universal Compatibility**: prepare didactic material that looks and behaves the same across all devices, enhancing instructors and students experience.
- **Inclusiveness**: beyond its technical contributions, the study aims to send a message to the educational community about the importance of inclusivity and accessibility, as a call to action.

2.5 Workflow

This dissertation presents a study that started as a Research Project. The Research Project consisted of 12 CFU, and it centered around the implementation of the alpha version of the bundle and its integration in HTML slides that can also work offline. It also focused on the development of a Minimum Viable Product (MVP) for a WYSIWYG slides editor. It finished with a first experimentation in a Summer School.

Part of the Research Project was co-financed by the European Commission in the framework of the Erasmus+ OpenU project (KA3 - 606692-EPP-I-2018-2-FR-EPPKA3-PI-POLICY). This work represents only the opinion of the authors and neither partners nor the European Commission can be held responsible for any use that may be made of the information contained herein.

The work for this dissertation increased the robustness of the product by solving issues encountered during the pilot study and introducing new features and languages. The WYSIWYG editor has been finalized, and the slides system has been assessed during various lectures.

2.6 Signposting

In this dissertation, we propose Interactive Code Playgrounds as an alternative slides system for programming education, while explaining its development process and examining its effectiveness. Chapter 3 is a Literature Review that describes how to exploit digitalization, focusing on Higher Education contexts. It also presents the main State-of-the-Art adopted technologies, while examining their limitations and principal challenges in digital transition, namely digital divide and accessibility. Chapter 4 describes the technical aspects related to ICPs, consisting of the technical goals we achieve with this product and its development process. Chapter 5 describes the research methodology followed to evaluate ICPs, based on a mixed methods approach involving surveys and field observations. Chapter 6 shows the results of our research, with an objective description of the collected data and a final discussion. Chapter 7 focuses on the current limitations of the proposed tool, and a general description of how it might be enhanced in the future. Finally, Chapter 8 concludes the dissertation by summarizing the main contributions of the research, highlighting its implications for computer programming education, and suggesting directions for further research.

3 Literature Review

3.1 Exploiting Digitalization

Early attempts to integrate Information and Communication Technology (ICT) into education during the 2000s often resulted in a medium change: didactic material is proposed in digital format, while the teaching paradigm stays the same. Clegg *et al.* (2003) observed that ICT-based learning typically only mimicked basic information giving functions, and even a few years later critical voices such as Selwyn (2007) stated that in university courses students' exposure to ICT was primarily limited to PowerPoint usage. [8, 39]

Some authors, such as Prensky (2008), argued that technology can improve teaching and the learning process only if the pedagogical practices also change, substituting the old "telling"-based teaching methodology. [33]

Nowadays the notion of technology disrupting and revolutionizing education has become somewhat common place. Another critical voice, Teräs, calls this notion a catchphrase, while questioning why education is considered broken, and whether technology is indeed the solution [43]. Henderson *et al.*'s analysis (2017) indicates that digital technologies *"are clearly not transforming the nature of university teaching and learning, or even substantially disrupting the student experience"*. As a result, their final suggestion is that universities should simply continue to develop and refine digital resources. [19]

Didactic material can be improved in various ways, such as through increased accessibility, availability and reliability [19]. Another approach is to make it more interactive. Studies have shown that students rarely learn by only receiving information; they learn through a process of reflection and interaction with the provided resources. [19, 27]

To summarize, the key to exploit digitalization lies not in the introduction of a disruptive technology, but in using digital advancements to enhance and upgrade educational material, by making it more robust, accessible and interactive when possible.

3.2 Teaching Programming in Higher Education

Over the past two decades, pedagogy in programming classes has undergone significant changes. While K-12 education has seen a pedagogical revolution, with a shift from studying technical aspects to developing computational thinking, programming pedagogy in Higher Education has been comparatively less considered. In practical terms K-12 has seen the introduction of visual programming languages such as Scratch that, coupled with an active approach, can significantly improve learning while maintaining student motivation and commitment [36]. Instead, in Higher Education the technologies employed for the last two decades have remained almost the same, grounded on slideshows and code snippets. This has resulted in a limited development of educational technology at the university level [39], reflected in the state-of-the-art solutions currently used and the obstacles students and instructors face when interacting with technology.

In many introductory programming courses students need to set up a development environment and learn how to use several components, such as code editors, compilers or interpreters, and run or debug interfaces. The challenges related to these processes have been widely documented: installing compilation tools and Integrated Development Environments (IDEs) can be cumbersome, especially for someone who is approaching programming for the first time; managing privileges in computer labs is problematic, since students typically need to install some tools, but they cannot be automatically given administrative privileges; requiring students to install the development environment on their personal computers leads to issues too, as slight variations between machines can easily result in compatibility complications. [13, 18, 22, 23, 26, 27, 45]

Providing a standardized programming environment to students is often a critical feature. As identified by Boroni *et al.* already in 1998: *"The elimination of platform dependence as a hurdle to*

producing good educational software systems that can be widely used without hassle is indeed a major achievement of profound import". However technical difficulties and differences in students' machines make this objective quite complex to achieve [4, 3, 23, 26]. For instance, the `C++` programming language has different compilers for different OSs, all of which have slightly different standards. The use of divergent tools or versions can result in issues popularly referred to as *"it works on my machine"*, which may negatively impact the students' experience as code examples working on the instructor's computer may not work on the students' ones. [45]

3.2.1 State of the Art Solutions

Various approaches have been attempted to address the aforementioned issues, each with different trade-offs.

Virtual Machines

Virtualization solves the problem of standardizing the students' development environment. There are two ways in which instructors can decide to provide a virtualized environment: through server-side Virtual Machines (VMs) or client-side VMs.

As documented by Malan (2013) and Laadan *et al.* (2010), cloud-based VMs introduce several complications. While students may have administrative access to a VM, they typically lack the same privileges on the host machine, which prevents them from configuring the virtualized environment. Furthermore, a persistent internet connection is necessary, and latency issues are often encountered.

Consequently, various universities have transitioned to client-based VMs, which grant students full control over allocated resources. Although a strong internet connection is required for the initial VM download, this solution works offline. However, VMs demand computational power, and the setup process remains complex. [18, 23, 25, 45]

Docker Containers

Docker is a platform that enables developers to build, share and deploy applications while eliminating all *"it works on my machine"* issues. Docker offers similar benefits as VMs, but it does not use full virtualization, as its containers share the host OS kernel. As a result, it is faster, and more resource-efficient. Additionally, instructors can easily design Docker Containers using a text-based configuration file. The main drawback of Docker is its potentially complex installation process, especially on Windows systems that require enabling the Hyper-V functionality, which is unsupported by the Home edition [26, 45]. Thus, setting up a container-based enviornment is even more complicated than installing a complete virtual machine. Additionally, the containers metaphor may be somewhat difficult to grasp, whereas virtual machines are typically easier to comprehend in comparison. In fact, the containers metaphor requires an understanding of OS-level virtualization and the concept of a shared kernel, which can be a bit more abstract than the idea of a complete virtual computer that VMs offer.

In-browser Environments

The Web has enabled the development of platform independent tools, which have proven to be valuable assets in education. In fact, the Boroni's argument about the importance of platform independence we previously mentioned was actually an extract of a study regarding a browser-based technology. Additionally, nowadays a web browser is a standard fixture for nearly all OSs. How many people have access to a browser, then? According to statista, the total number of mobile devices in 2021 was approximately 14 billion[1], and in 2019 almost half of private households worldwide were estimated to have a computer at home (reaching nearly 80% in developed countries)[2].

The current landscape of web-based educational tools is vast and continues to grow, particularly in blended-learning situations. We already have successful cases of integrating container-based systems within the browser, as proved by the use of GitHub Codespaces in Harvard's CS50 course [26]. Another

[1] `https://www.statista.com/statistics/245501` - Last Accessed 20/04/2023

[2] `https://www.statista.com/statistics/748551` - Last Accessed 20/04/2023

typical use case we find in various university courses is related to the use of web server-based code playgrounds that allow students to access an online development environment with predetermined exercises and collaborative editing features. Among the most widely used services, we find Replit[3] and Python Tutor[4]. [14, 15, 16, 48]

The possibility to easily create these platform independent tools, however, comes at a cost: all the web applications we have described require a persistent internet connection. Can we afford to introduce such a requirement? Furthermore, when utilizing third-party provided services, we become reliant on external actors, namely the service provider, which may for instance impose restrictions in the future.

3.3 Challenges in Digital Transition

The integration of ICT in education holds significant potential for shaping the future of learning. However, improper implementation could result in *'a further widening of the knowledge gap and deepening of existing economic and social inequalities among the developed and the developing countries'*. [30]

What steps can be taken to address these challenges and ensure equitable access to education? In the next Subsections we focus on analyzing two peculiar issues: digital divide and accessibility.

3.3.1 Digital Divide

The recent surge in web-based learning environments, driven in large part by the Covid-19 pandemic, has exposed the digital divide that exists when internet connectivity is a prerequisite for education.

Surveys conducted during lockdowns have revealed that many students lack access to a stable internet connection. For example, in one of the regional colleges of the University of the Philippines, 41% of undergraduate students did not have access to an internet connection, while 51% of them had data caps due to due to mobile network-enabled internet. [43, 44]

Apart from students-related challenges, some universities may also face infrastructure problems. As observed by Mikre in 2011, for instance, only about 40% of schools in Ethiopia had computers, and most of them still experienced limited or low internet access. [30]

One last issue, easily forgotten, is the link decay phenomenon. The web is not a true library, and the literature has often reported how links tend to become inaccessible after several years. Projects like the Internet Archive (IA)[5] try to address this issue by storing snapshots of pages currently available on the internet and providing an alternative way to access them, but they cannot guarantee a completely exhaustive archive. Hence, any didactic material only accessible through the internet may become unavailable in the future, especially if it is not under the direct control of the university providing it. [12, 20]

3.3.2 Accessibility

Accessibility is the practice of making resources usable by as many people as possible. It embodies the idea that everyone has the right to be included in society, no matter what their ability or circumstances [40]. According to the World Health Organization (WHO), approximately 16% of the world's population experience significant disability[6]. In detail, the *MDN Web Documentation* identifies four categories of disabilities that should be considered when building resources: visual, hearing, mobility and cognitive impairments[7].

The Portable Document Format (PDF) has been one of the most widely used standards for document exchange for several years. PDF files, however, have not been originally designed with accessibility in mind, and screen readers encounter numerous problems while following their structure. The PDF/UA standard takes into account accessibility, but it still has various pitfalls, and it is not widely used yet. [41]

From the education in Computer Science (CS) perspective, studies have shown that most of the technologies used in coding classes, such as Visual Studio or Eclipse, present significant challenges for

[3] `https://replit.com/` - Last Accessed 20/04/2023
[4] `https://pythontutor.com/` - Last Accessed 20/04/2023
[5] `https://archive.org/` - Last Accessed 20/04/2023
[6] `https://www.who.int/en/news-room/fact-sheets/detail/disability-and-health` - Last Accessed 20/04/2023
[7] `https://developer.mozilla.org/docs/Learn/Accessibility` - Last Accessed 20/04/2023

students with visual impairments. In fact, the percentage of visually-challenged students in CS-related courses is low and calls for concern. [2, 28, 32, 42]

In recent years, most of the attempts to create accessible material have employed the HTML language, which provides semantic elements and Accessible Rich Internet Applications (ARIA) attributes for screen readers. Hence, browser-based educational tools represent a promising solution to enhance accessibility in programming courses. [9, 32, 41, 47]

4 Development

In Chapter 3 we have seen the main State-of-the-Art attempts to enhance programming classes by exploiting digitalization, together with their limitations. To solve these limitations, we propose Interactive Code Playgrounds (ICP), an interactive slides system built for students learning programming languages.

4.1 Achieving Technical Goals

The Literature Review in Chapter 3 shows how the setup of a development environment in an introductory programming class typically leads to **technical issues**. We expect didactic material created with ICPs to drastically reduce the amount of problems and troubleshooting time needed during hands-on programming lectures. The development environment offered by this new kind of didactic material is a web technology, which makes it inherently compatible with any device that has access to a browser. Additionally, no setup or installation process is needed: students are simply required to open a link or execute a local file containing the slides.

When developing software, **sustainability** is a too often overlooked problem. Currently, computing emissions make up almost 4% of the world's total emission, and they are predicted to increase significantly. By 2040, emissions from computing alone are projected to account for more than half of the emissions budget necessary to keep global warming below 1.5°C. In addition, the energy consumption of all communication and computation technology currently used worldwide is around 11% of the world's electricity consumption, and is expected to increase by 3-4 times by 2040, based on a conservative estimate. To address these concerns, Wim Vanderbauwhede (2023) proposes "*Frugal Computing*", which aims to *"achieve the same results for less energy by being more frugal with our computing resources"*. By adopting more sustainable computing practices, such as optimizing energy efficiency and useful life of computing resources or reducing network utilization, we can help reduce the environmental impact of computing, and contribute to global efforts to mitigate climate change. [46]

Our proposed solution fulfills most of the major requirements identified by Vanderbauwhede. Firstly, ICPs works efficiently on older devices, as it uses minimal computational power and only requires a relatively up-to-date browser. Secondly, we anticipate ICPs to be a long-lasting technology. The system will only be impacted if specific browser APIs - such as `SharedWorkers` - are excluded from future updates. However, historical trends indicate that web-related updates are usually introduced with backward compatibility, both in terms of communication protocol and browser implementation, which makes such changes quite unlikely. Thirdly, ICPs is not a full-fledged IDE, but rather a simplified version that demands fewer computational resources, thus reducing the overall energy consumption of the system, compared to similar software. Last but not least, ICPs requires minimal network data, as the compilation and execution of code snippets occur client-side, requiring no interaction with external service providers. Therefore, a network connection is only necessary to download the slides.

Developing a sustainable and scalable software in an academic setting is complicated. There are multiple constraints in terms of workforce, time and resources. To manage these issues Philip J. Guo (2021) has identified 10 design guidelines [14]. Among them, we feel ICPs particularly excels in the following ones:

- **user experience** - `walk-up-and-use`: no installation process or login is needed, it just works;
- **software architecture** - `be stateless`: the project does not maintain any state, in fact a *backend* server is not even required;
- **development workflow** - `single developer`: having a single developer working on ICPs helps maintaining continuity.

A single guideline is not met, which de facto represents one of the main problems of ICPs: dependencies are not really minimized. The project relies on a lot of external dependencies, that limit its robustness. However, thanks to the use of version control systems and packet managers, even breaking changes in outer packages would not disable ICPs.

4.2 Project Architecture

The system's development relies on the use of web technologies, which constitute what we call *frontend*. This particular branch of Computer Science manages everything happening on the client-side of web pages, including user interfaces and code executed in the scripting engine of the user's browser. In simple terms, this means that the developed slides system only depends on HTML templates and scripts downloaded at startup, and after the initialization no additional communication with external entities is required for ICPs to work.

The source code of the Interactive Code Playgrounds project is split in different modules, each providing a specific feature. These modules are openly available on my GitHub profile at `github.com/lucademenego99`:

- **icp-bundle**: a plugin that offers code playgrounds exposed as web components;
- **icp-editor**: a web application that allows users to create ICPs slides through a WYSIWYG user interface;
- **icp-create-server**: an NPM[1] package that automates the creation of ICPs slides not requiring an internet connection;
- **icp-slides**: a repository containing all the ICPs-based slide decks created by me as demos, examples or didactic material for actual lectures;
- **icp-tiddlywiki**: a TiddlyWiki notebook that provides in-depth information about the project's development.

The main feature offered by ICPs is the ability to edit and execute code in a simple browser-based development environment. This requirement is fulfilled by the `icp-bundle` project, which allows to create code editors that include a set of buttons to configure the environment and execute the provided script.

The creation of slides in HTML format is entrusted to `reveal.js`, a widely used HTML presentation framework that offers all the typical features needed in slideshows. Code playgrounds created using `icp-bundle` can be put into `reveal.js` slides through an HTML template representing the editor and some custom Cascading Style Sheets (CSS) that fill the compatibility gap between the two technologies.

The ability to use didactic material built with ICPs without an internet connection is a fundamental feature we want to offer. This is achieved by using `Redbean`, a single-file distributable web server that automatically exposes the content put inside it in a local web server accessible via `http://localhost:port`. So, an instructor may offer didactic material built with ICPs in two ways:

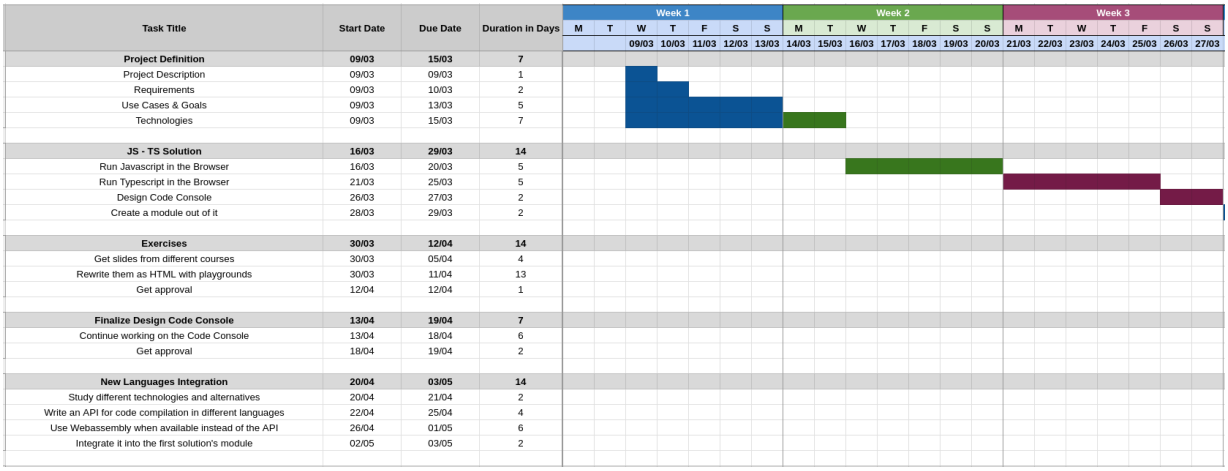
- by exposing it online, as an HTML page;
- by providing a `Redbean` file containing the slides.

We understand that creating slides using HTML and CSS may be a cumbersome process. We developed a web *"What You See is What You Get"* (WYSIWYG) editor that instructors can use to create slides via a visual user interface. Even if it is quite limited in terms of functionalities, it completely eliminates the requirement of knowing HTML, and automates slides export.

[1] A Software Registry for Javascript packages

4.3 Development Process

The development process follows some principles of the Agile Manifesto, by delivering software quickly and talking frequently with my supervisor. Figure 4.1 shows part of a timeline followed during development, which enforces continuous delivery and a fixed pace throughout the entire process.



Task Title	Start Date	Due Date	Duration in Days
Project Definition	**09/03**	**15/03**	**7**
Project Description	09/03	09/03	1
Requirements	09/03	10/03	2
Use Cases & Goals	09/03	13/03	5
Technologies	09/03	15/03	7
JS - TS Solution	**16/03**	**29/03**	**14**
Run Javascript in the Browser	16/03	20/03	5
Run Typescript in the Browser	21/03	25/03	5
Design Code Console	26/03	27/03	2
Create a module out of it	28/03	29/03	2
Exercises	**30/03**	**12/04**	**14**
Get slides from different courses	30/03	05/04	4
Rewrite them as HTML with playgrounds	30/03	11/04	13
Get approval	12/04	12/04	1
Finalize Design Code Console	**13/04**	**19/04**	**7**
Continue working on the Code Console	13/04	18/04	6
Get approval	18/04	19/04	2
New Languages Integration	**20/04**	**03/05**	**14**
Study different technologies and alternatives	20/04	21/04	2
Write an API for code compilation in different languages	22/04	25/04	4
Use Webassembly when available instead of the API	26/04	01/05	6
Integrate it into the first solution's module	02/05	03/05	2

Figure 4.1: Extract of the timeline followed during the development process

As mentioned in Chapter 2, part of ICPs was developed for a 12 CFU *Research Project*, finished with the pilot study described in Chapter 5. This first version represents a Minimum Viable Product (MVP) meeting our requirements, consisting of a simple code editor compatible with few programming languages. The dissertation development work centers around the modularization through a web framework, the integration of new features and the improvement of the product's robustness.

The following Sections do not consider all intermediate steps followed during the development phase. Rather, they provide a description of the complete project, showcasing how the final version has been developed. Hence, a large part of the initial work completed during the Research Project is not taken into consideration.

4.4 Bundle Setup

The ICPs Bundle has been built with the Svelte framework, and it relies on different dependencies to provide a complete code editor and compilation/execution features. Each available playground is exposed as a web component, allowing developers to easily import it into any web page. The following Subsections describe in detail the overall structure of the project, and how different modules communicate between each other.

4.4.1 Svelte

Most traditional frameworks used nowadays to develop web applications perform most of their work directly in the browser, client-side. For example, React's primary algorithm for rendering page components is based on the use of a Virtual DOM tree, which is a lightweight copy of the actual DOM. When there is a change in the application's state, the Virtual DOM is recreated and compared with the real DOM. The differences between the two trees are reconciled using a heuristic algorithm called *"Diffing algorithm"*, which involves iterating through the children recursively. Despite being based on heuristics and achieving a `O(n)` complexity, there are still unnecessary computations that can hinder page loading and interaction.

Svelte shifts most of the computation to compile-time and produces highly optimized vanilla Javascript code. This code surgically manipulates the DOM when the state changes, thereby avoiding the use of a Virtual DOM and the overhead of creating and reconciling DOM trees. As a result, Svelte's bundles are much more compact since no additional code is required to maintain the Virtual DOM.

When developing code playgrounds, bundle size and execution time can present significant challenges, especially for languages with large compilation dependencies or lengthy runtime initialization

times (Subsection 4.6.2). This is why choosing a framework that takes these complications into account is critical. Thanks to this decision, the project has achieved an average bundle size of `3Mb` per language (not including external dependencies).

4.4.2 Web Components

Web Components is a suite of technologies that allows developers to create reusable custom elements ready to be used in web applications and HTML pages. Thanks to this technology, we can embed all the business logic of a particular module in a simple element tag, integrable within HTML pages by simply using its name:

```
<my-web-component-name></my-web-component-name>
```

Each web component is composed by three main elements:

- custom element: Javascript APIs that allow to define custom HTML tags with their behavior;
- shadow DOM: Javascript APIs that allow to attach an encapsulated DOM tree into an element, privately handled by default;
- HTML template: gives developers the possibility to write markup templates not displayed in the rendered page and usable multiple times on-demand.

The creation of web components is typically a bit cumbersome. In fact, it requires to:

- integrate the component's functionality into a class that extends an HTML element;
- register the new custom element with `CustomElementRegistry.define()`;
- attach a shadow DOM to the element, with `Element.attachShadow()`;
- define an HTML template with `<template></template>` and optionally `<slot></slot>`.

Svelte eases this process by asking to only use one single line to define a web component:

```
<svelte:options tag="web-component-tag-name" />
```

This line will tell the compiler to translate the Svelte component into a web component, but only if the build configuration defines the following option:

```
compilerOptions: {
    customElement: true
}
```

Web Components have been the chosen technology since they make the integration of ICPs into web pages a trivial process: one just needs to import the script and use the exposed component as any other HTML element.

4.4.3 Components Communication Pattern

The Interactive Code Playgrounds this bundle offers have been implemented as Svelte components. Simple Svelte components, however, cannot be easily imported into webpages, therefore during the build process these components are translated into web components.

When performing this translation and using an architecture based on nested components, a few issues must be handled, derived by the fact that the CSS styling of imported components is lost due to the inner mechanics of Shadow DOM. Importing them as web components instead of Svelte components is a possible solution. A solution that, however, leads to another concern: child-parent message passing becomes challenging as the typical communication pattern Svelte uses stops working. `CustomEvents` represent the resolution of these complications, as they are particular events that can be used as an alternative communication pattern.

The code snippet below shows an example of `CustomEvent` inizialization:

```
const event = new CustomEvent("hello-world-event", {
    detail: {
        message: "Hello World!"
    },
    bubbles: true,
    cancelable: true,
    composed: true
});
```

This event has a name (*"hello-world-event"*), it contains some information (in *"detail"*) and it has a set of additional options. The `bubbles:true` property allows for event listeners to catch and handle the generated event. The `composed:true` option allows the event to propagate across the Shadow DOM boundaries, which enables easy message exchange between child and parent belonging to independent DOM trees. The `cancelable:true` option allows developers to prevent the browser's default behavior by calling the `event.preventDefault()` method.

Apart from providing a simple channel for child-parent communication, `CustomEvents` also allow messages to be sent directly to the main application that is using ICPs. The `changedout` CustomEvent, for instance, is fired every time the user changes the code inside the editor. Despite being important also ICP-side, any website using this tool can listen to code changes by exploiting this CustomEvent, which is what happens in the WYSIWYG editor described in Section 4.13.

4.5 Code Editor

Using a basic text input for a code playground is clearly not ideal, as it lacks the essential features we typically expect when coding, such as syntax highlighting and keyboard shortcuts. Therefore, a complete in-browser code editor has been one of the first components integrated into ICPs.

There are various open-source products that offer the main features of a code editor within web pages, and the most popular ones are Ace, CodeMirror and Monaco. **Ace** was one of the first code editors implemented for the browser, hence today it is very stable. However, its API seems outdated, and its three-clause BSD license is quite restrictive, compared to for instance the MIT license. **Monaco** is the editor that also powers VS Code, it has a modern and advanced UI and excellent documentation. However, it has a huge bundle size and it is not as fast as Ace or CodeMirror. **CodeMirror 6** was recently rewritten to provide a new documentation, API and UI. It is now distributed as a collection of modules, which makes its integration straightforward and highly customizable. CodeMirror is accessible, and it is the only web code editor that can be smoothly used on all devices. Additionally, it is available under the MIT license. Despite being less stable than Ace, thanks to all these features it is starting to get adopted in production environments by many companies, such as Repl.it and Adobe. As a result, CodeMirror was selected as the code editor for Interactive Code Playgrounds.

4.5.1 Integration into ICPs

CodeMirror 6 is entirely modularized, so it comes with a range of NPM packages, each providing specific extensions. However, the main npm package, `"codemirror"`, already includes a curated set of extensions needed to create a basic editor, such as default command bindings, line numbers, undo history, bracket matching and closing, search text and a fold gutter. Therefore, this package serves as the starting point for CodeMirror integration. The upcoming Subsections provide further details on how external extensions have been integrated to personalize the environment.

4.5.2 Languages

One of the key features typically expected from a code editor is syntax highlighting. Fortunately, for most of the programming languages integrated into ICP, CodeMirror offers a dedicated NPM package that provides the corresponding language support and a parser. Installing the NPM package and integrating the language as an additional CodeMirror extension is usually sufficient to make it work.

The only language for which we used an external module is Typescript. Despite being part of the CodeMirror's Javascript language plugin as a simple component offering syntax highlighting, we decided to include a plugin[2] that also offers linting and autocompletion features. This plugin relies on

[2]The module's source code has been taken from the `prisma/text-editor` repository.

a virtual file system created within the browser, used for a local Typescript language server based on the `vfs` NPM package.

4.5.3 Tabs Handling

Due to the W3C Web Content Accessibility Guidelines' "No Keyboard Trap" criterion[3], CodeMirror does not handle the *Tab key* by default. While most users navigate the internet with a pointing device, some users rely instead on keyboard navigation. It's crucial to ensure that they can use the Tab key to move between page elements for accessibility reasons.

However, in a code editor setting, the Tab key is also necessary to handle indentation. As a solution to this peculiar problem, a button has been added in the playgrounds that allows users to activate and deactivate the Tab key handling on-demand.

From a development standpoint, CodeMirror already provides an extension from the NPM package `@codemirror/commands` that allows to set up indentation with the Tab key. However, this feature must be completely under the user's control. To achieve this, a Compartment object was used. **Compartments** are extensions containers that can be dynamically reconfigured using a specially crafted `Compartment.reconfigure()` function that takes the new extensions the Compartment should provide as input.

In this case, the compartment related to tabs handling starts with an empty list of extensions. A globally exposed function `setTabsHandling` takes a CodeMirror editor, the compartment and a boolean indicating whether the tabs handling should be enabled or not, and automatically reconfigures the provided extensions. This update is finally applied by dispatching the created transaction to the CodeMirror editor, which updates the editor's state and view accordingly.

4.5.4 Editable Filter

In programming classes, instructors often need to guide students through exercises where they are required to modify or complete only certain parts of the provided code. As stated by McIntyre et al (1998):

> "this would create a pleasant, interactive environment in which the student is unencumbered with the implementation details of the code and instead can focus upon the intended properties of the code." [27]

To address this need, a new CodeMirror extension has been developed that allows to restrict the editable portion of the code. This is accomplished through a `StateEffect` that designates which parts of the text should be made non-editable. `StateEffects` are used to represent additional effects associated with a transaction, and in this case, we want to apply a non-editable filter running within a transaction every time the user attempts to modify the code. Additionally, the `StateEffect` is linked to a specially-crafted `Decoration` that gives editable and non-editable texts specific CSS classes, providing the user with visual feedback about which parts of the code are read-only.

Teachers can designate which parts of the code are editable by using the tag `<EDITABLE>`. For example, the following code creates a Javascript playground where only the log message is editable:

```
<javascript-editor code='console.log("<EDITABLE>Hello World!</EDITABLE>");' />
```

When creating the code editor, the ICP bundle detects the editable tags, adds the read-only transaction filter to the list of extensions, and sets the editable and non-editable parts based on the inserted tags. This is achieved through an exported function that, given the editor and the range, dispatches an event on the editor that applies the `StateEffects` representing editable and non-editable fields.

4.5.5 Light/Dark Mode

When creating an editor, it is possible to choose between two available themes: `light` and `dark`. As most of the extra features, the theme is implemented using a `Compartment` object which allows dynamic theme changes by dispatching an effect on the editor that reconfigures the applied extensions.

[3] `https://www.w3.org/WAI/WCAG21/Understanding/no-keyboard-trap.html` - Last Accessed 07/04/2023

The dark theme extension is provided by the NPM package `@codemirror/theme-one-dark`, while the light theme is enabled by default and requires no additional extensions.

4.5.6 Additional Features

In addition to the main features, several small extensions have been added to the code editor. These include:

- an extension enabling line wrapping in the editor by setting CSS white-space to pre-wrap in the content;
- an extension that fires a DOM `CustomEvent` "execute" when the user presses `CTRL+Enter`, which will eventually be caught to execute the code;
- an extension that fires a DOM `CustomEvent` "changedcode" whenever the code changes.

4.6 In-browser Compilation

WebAssembly (Wasm) is a binary instruction format designed as a portable compilation target for programming languages. One of its key features is that it can be used within the web platform, where it can access browser functionalities through the usual web APIs, and where Wasm-generated modules can be accessed via Javascript. This has expanded the capabilities of web browsers, and led to new possibilities. In this Section we investigate the use of WebAssembly to compile programming languages directly in the browser's internal scripting engine without relying on external services. Since the compilation happens locally, this eliminates the need for a constant internet connection. In an educational setting, this allows for the creation of didactic material that can be accessed offline, which is essential for students who have limited internet access or who frequently travel.

4.6.1 WebWorkers and SharedWorkers

The Javascript scripting engine employed by browsers can only execute code in a single thread, and no parallel execution of Javascript code is possible without the use of external browser's APIs. Although asynchronous idioms are available in Javascript, such as AJAX requests, these do not represent a true parallel execution. The term asynchronous only indicates the fact that multiple instructions may be executed in scrambled order. In fact this is usually implemented as the so called event loop, which resembles the following code:

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

Apart from Javascript, within a browser also UI updates and interactions are handled in the same main thread. This means that heavy Javascript-side tasks can block the main thread and cause UI to become unresponsive.

In our case most of the compilation and execution tasks can be considered heavy, and it is unacceptable to prevent the user from interacting with the web page while testing some code. To address this issue `WebWorkers` and `SharedWorkers` have been used, which are Javascript APIs that enable multithreaded processing. Code executed within a `Worker` will in fact run in a thread different from the main one, hence UI will not suffer of performance issues. Moreover, messages can be easily passed between the main thread and the `Worker`. The following snippet, for instance, demonstrates how to perform message passing between the main thread and a `WebWorker`.

Main thread:

```
worker.onmessage = (e) => {
    console.log("Received message from worker ${e.data}");

    worker.postMessage({
        request: e.data,
        response: "Pong"
    });
}
```

Worker:

```
onmessage = (e) => {
    console.log("Received message from main thread ${e.data}");

    postMessage({
        request: e.data,
        answer: "Ping"
    });
}
```

By moving all tasks related to execution and compilation in `Workers`, the browser's main thread won't be busy, and the web page will be usable even when running some code snippets.

To improve readability and maintain separation of concerns in the actual implementation, the same message-passing protocol has been used for all `Workers`. In this way the code responsible for asking the compilation of a code snippet and receiving the responses will not depend on the programming language in use. In detail, this protocol defines:

The request for compilation as:

```
type Message = {
    language: Language;
    script: string;
    input: string;
    offline: boolean;
    baseUrl: string;
};
```

The response as:

```
type WorkerResponse = {
    debug: string;
    result: string;
}

type WorkerError = {
    error: string;
    line: number;
}
```

In the request, the language and script keys define the programming language and the code that should be executed. Offline tells whether the execution should happen without the requirement of an internet connection. If this is the case and the compiler needs additional dependencies, these dependencies must be provided by the web server exposing the slides under the path `baseUrl`. Input is still not used, but it has been inserted as a possibility for future updates: if a code snippet handles I/O, the input should be taken from this field instead.

The main response contains debug, a string exposing all logs, and result, the final result of the code snippet. If there is an error during the execution, instead, an error message is passed to the main thread, together with - if available - the line number in which the error occurred.

What is the difference between a `WebWorker` and a `SharedWorker`? Apart from the dissimilar APIs that need to be used to access these tools, the most important difference is given by the fact that while a `WebWorker` is only accessible by the script that called it, a `SharedWorker` can be shared between multiple scripts, even if they are being accessed by different windows. A `SharedWorker` becomes vital for the compilation of all languages that require a high computational cost during the initialization of the environment. In fact, by sharing between multiple playgrounds a single worker responsible for compiling a language, the initialization of the environment happens only once, drastically improving the user experience.

4.6.2 Available Languages

Javascript/Typescript

Javascript and Typescript can easily be executed within a browser context without relying on Wasm-based solutions. Given a Javascript code snippet stored as a string, it can be run by using one of the

following constructs:

```
eval(source);       // First alternative
Function(source)(); // Second alternative
```

Executing user-provided code using `eval` is discouraged as it poses various security risks. Therefore, the `Function` constructor is preferred as it parses the given code into a function object, and the executed code runs in a separate scope.

Typescript code just needs to be transpiled before calling the `Function` constructor, by using the function `transpile` exported by the official `typescript package`.

The only problem that may arise when trying to execute user-provided code in this way is the fact that all `console.log()` calls will simply be put inside the browser's Javascript console. In order to catch these logs, all main console functions have been rewritten into a new version that instead stores the inputs inside an array. Following is a code snippet that shows how it has been managed:

```
const messages = [];
const logRewrite = (msg, ...optionalParams) => {
    messages.push(msg + ' ' + optionalParams.join(' '));
};
console.log = logRewrite;
console.info = logRewrite;
console.warn = logRewrite;
console.error = logRewrite;
console.debug = logRewrite;
console.assert = msg => {
    if (!msg) {
        messages.push("Wrong Assertion!")
    }
}
```

This allows us to eventually get the output and show it to the user by simply sending to the main thread a message containing:

```
messages.join("\n")
```

To catch errors a simple `try/catch` construct has been used, that sends the caught exception to the main thread if needed.

Python

Python has been integrated into ICPs through `Pyodide`, which is a port of CPython to WebAssembly performed through Emscripten, a famous compiler toolchain for Wasm. This makes Pyodide a Python distribution for the browser that allows to run Python code in the browser. Thanks to `micropip`, any pure Python package can also be imported, even if for the scope of our project when using Pyodide offline only a subset of packages is actually available.

When using ICPs with an internet connection, Pyodide will be automatically imported by the web worker from the CDN; when a connection is not available, instead, all Pyodide dependencies must be exported by a local web server, and their path needs to be provided by the initialization message sent by the main thread to the worker.

As in Javascript's case, it is essential to catch all logs and error messages. When initializing Pyodide, some custom `stdout` and `stderr` functions are provided, that push all printed messages to a specially-crafted array. Following is an example of the `stdout` function redefinition:

```
stdout: (text) => {
    if (pythonInitCompleted) {
        messages.push(text);
    } else if (text.includes('Python initialization complete')) {
        pythonInitCompleted = true;
    }
}
```

After the initialization, Python code can be executed in the following way:

```
pyodideObj.globals.clear();
await pyodideObj.loadPackagesFromImports(message.data.script);
let output = await pyodideObj.runPythonAsync(message.data.script);
```

The first line re-initializes the Python environment, removing all previously defined variables; the second line handles all imports, and the last one executes the Python code.

Java

The two commonly used alternatives for compiling Java code into WebAssembly are CheerpJ and TeaVM. CheerpJ is a solution allowing you to convert Java applications into Webapps that rely on a CheerpJ runtime exposed through a proprietary `HttpServer`. It may be the most production-ready solution, but it has two big disadvantages: it is not open-source, and any application using it must connect to the internet and download some dependencies only available in CheerpJ servers.

TeaVM is a similar open-source technology that allows the compilation of Java bytecode emitting Javascript and WebAssembly code that runs in browsers.

Our use case, however, is quite different: we don't want to compile a pre-made Java application and expose it through a web server; we want to compile and execute Java code directly within a browser. This task in general is not production-ready yet, regardless of the chosen technology. There are some solutions available, but creating an entire JVM in Wasm is challenging, and typically some bugs can be encountered when executing relatively complex applications.

Both CheerpJ and TeaVM provide a solution to this problem, but due to the previously explained limitations of CheerpJ, we will just focus on TeaVM's one. TeaVM's team has developed a project, called `teavm-javac`, which is essentially Javac from OpenJDK compiled with TeaVM and exposed as a WebWorker. This WebWorker, briefly explained, takes as input a message with the code snippet to be executed and returns a message with the code execution's result.

This project is however abandoned, so it is not up to date with TeaVM's latest versions and some bugs are still present. Nevertheless, for simple applications the compiler works, and it is usable for introductory courses.
The project can be compiled with Maven, and the outputs we need are:

- `classes.js`: Javascript code exposing a WebWorker that has the capabilities of compiling and executing Java Code;
- `classlib.txt`: a zip file containing all `.class` Java files, needed for the runtime initialization.

Due to the previously discussed limitations of `WebWorkers`, we forked the `teavm-javac` project and modified it to export a `SharedWorker` instead of a `WebWorker`. Briefly explained, this has been achieved by recreating in Java the `SharedWorkers` APIs structure and modifying the code responsible for message passing.
Finally, ICPs-side, three `SharedWorkers` are needed:

- `JavaWorker`: communicate with the main thread, load the `classlib` file and initialize the other workers;
- `JavaTeaWorker`: run the `classes.js` generated file to compile code;
- `JavaRunWorker`: run the Java code compiled into Javascript and listen for stdout/stderr messages.

SQL

In Computer Science courses, databases are a key topic of study. As for programming languages, it may be difficult at first to set up a local environment, which in this case means creating a database and connecting to it. Moreover, it can be useful to quickly test SQL commands and queries during a lecture without having to manually set up and clear a database.

For this reason we decided to integrate a SQL environment into ICPs. The integration has been performed using `sql.js`, a compilation of SQLite to WebAssembly performed through Emscripten.

In detail, this project uses as database a virtual file stored in memory automatically cleared when the user exits the page.

To maintain consistency between different executions, the database is initialized each time the user runs a code snippet. The initialization and code execution are handled by the following code:

```
let db;
const SQL = initSqlJs({ locateFile: (file) => wasmUrl }).then((SQL) => {
    db = new SQL.Database();
    loadedSql = true;
});
await SQL;
const output = db.exec(script);
```

When performing statements that select data, the output is a table of the form:

```
type Table = {
    columns: string[];
    values: any[][];
};
```

To show this information to the user, the output is then translated into an `HTMLTableElement`.

Processing

Processing is a Java-based software sketchbook used to create graphics. As described in Chapter 5, sometimes it is promoted as a language for learning how to code, so it has been integrated into ICPs.

Incorporating it into our Java solution would be complex, since we would need to map the entire Processing framework with TeaVM. Due to this difficulty, instead of compiling user-provided Processing code we decided to translate it into P5.js. Most P5.js APIs, in fact, are easily relatable to the Processing ones. Additionally, being P5.js a Javascript framework, the translated code can be easily executed in a browser context.

During the development phase, we employed a set of functions derived from the *processing-p5-convert.js* project to automatically map Processing code into P5.js. The mapped code may create a canvas that does not fit into the ICPs' output container. Hence, we designed some regular expressions that, bounded with replace functions, update the user code with the correct canvas size.

The P5.js execution in global mode cannot happen inside a `Worker`, which brings some complexities when infinite loops or runtime errors are encountered. To solve the infinite loops issue we integrated a babel plugin called `loopProtection` that automatically wraps all loops into conditional breaks. Runtime errors, instead, are caught by overriding the `onunhandledrejection` function of the `window` object.

StandardML

Standard ML is a functional programming language, popular in the development of theorem provers, and among compiler writers or programming language researchers. The integration of this language has been straightforward thanks to the `SOSML` open-source project that gives access to an online StandardML interpreter written in Typescript and compatible with most of the StandardML features. This product is already used in a freshman class at Saarland University, and now it is completely integrated also into our code playgrounds.

From the development perspective, the worker responsible for executing a StandardML snippet splits the scripts each time a semicolon is found and tries to evaluate the result. If the given code part is not a complete valid script, the SOSML interpreter returns an `IncompleteError`. If an `IncompleteError` is thrown and some splits are still available, the Worker automatically appends to the script the next split, until the interpreter returns `OK`.

C/C++

C and C++ are widely used general purpose programming languages.

The local compilation of C/C++ language is based on the work by Ben Smith showcased during the CppCon 2019 talk, in which the developer used WASI SDK tools to compile the LLVM source (a toolkit for the construction of compilers, optimizers and runtime environments). The result is clang running on WebAssembly, usable with any WASM runtime (e.g. Wasmtime). Running it in a browser is a bit more complicated and requires some work: WASI implementation is not currently provided by the browsers yet. Ben Smith solved this issue by manually implementing a polyfill for the required imports using Javascript and WebAssembly. He developed an in-memory file system (MemFS) that implements all imports required for clang, and he wrote a small script that at startup automatically untars inside MemFS a sysroot[4] containing all WASI SDK libraries (includes) [1].

As an general overview, the compilation of a C/C++ program into WASM follows these steps:

- untar the sysroot into MemFS;
- copy the program source code into MemFS;
- when the run button is clicked, write the same program into the clang module, which will output an object file into MemFS;
- after the link call, the object file will be linked, producing a wasm output put in MemFS;
- pull the wasm file from MemFS and compile it using the normal WASM compilation tools, producing a runnable WASM module;
- run the WASM module.

4.7 Offline Version

Interactive Code Playgrounds are based on HTML and Javascript, so they can be used within a browser when exposed by a web server. However, this means that an internet connection is required to access the didactic material provided by professors, which is a strong limitation for students without internet access. How can we solve this issue?

Since all languages are compiled and executed client-side, we may think that simply downloading the slides will allow us to get access to the didactic material offline. However, while this may be true for languages like Javascript and Typescript, there are some languages with a lot of dependencies that cannot be directly included inside an HTML file.

Asking students to download multiple files beyond the HTML file is not an option, as it leads to CORS errors. Historically local files from the same directory were treated as having the same origin. Nowadays, however, most browsers treat all local files as having opaque origins. The reason is described in the advisory CVE-2019-11730[5]: if a user opens an HTML file locally, a malicious script available within it may use `file://` URIs to access other local files through name guessing, leading to important security implications.

The only solution to this problem would be to start a local web server on the students' computers. But this would clearly make the entire process of opening didactic material quite complex, considering also the fact that students would have to set up their system by installing an HTTP server. The Redbean technology described in the next Subsection solves the issue.

4.7.1 Redbean

Redbean is an open-source single-file distributable web server based on the concept of zip executable and compatible with six operating systems: Linux, Mac, Windows, FreeBSD, NetBSD and OpenBSD. By default, when executed, Redbean starts a web server that exposes everything contained within itself. It is designed not to be installed: after having downloaded the file it is possible to run it without additional dependencies. This is possible thanks to Cosmopolitan Libc, which makes C a build-once run-anywhere language that does not even need an interpreter or virtual machine.

[4] A sysroot is a directory tree that contains the system headers and libraries that are used by the compiler and linker to build software for a specific target platform

[5] CVE-2019-11730: `https://www.mozilla.org/en-US/security/advisories/mfsa2019-21/#CVE-2019-11730`

Why should we use Redbean for Interactive Code Playgrounds? As previously mentioned, to make it possible for ICPs to work offline we need a web server simple to start and with no external dependencies. Redbean allows us to create distributable web servers that contain, within themselves, the entire ICPs runtime environment together with the slides system. Moreover, being Redbean a zip file, all ICPs dependencies can be compressed.

Hence, to distribute a slide deck built with ICPs, we just need to:

1. download the base Redbean zip executable;
2. insert into Redbean the slides and ICPs dependencies;
3. distribute the file to the students.

The students will eventually be able to execute the program and access the local web server automatically started by Redbean.

As a last step, we can configure Redbean to exactly meet our needs. Redbean provides various files and APIs to manage the started web server and the zip executable itself. For instance, the `.init.lua` file is a script run at startup that allows the state of the Lua interpreter and various server options to be modified. In our specific case it has been configured with the following functions:

- `ProgramCache(0)`: disable caching. This is essential as, without this configuration, when opening different Redbean files we may visualize wrong cached slides.
- `ProgramHeader("Access-Control-Allow-Origin", "*")`: disable all CORS-related problems, that may arise when performing requests from the WebWorkers.
- `LaunchBrowser('/')`: automatically launch the system's default browser at startup, with the URL location already set to the exposed web server. This makes the entire process of accessing the slides easier, as students are not expected to understand under which port the server is exposed.

4.7.2 Automating the Server Creation

When the bundle's build phase finishes, all ICPs dependencies are automatically inserted into a Redbean file. This means that all we need to prepare a distributable Redbean web server is to insert the slides inside the resulting zip. This is simple to achieve from a Linux environment, where a command like the following:

```
zip redbean.com slides.html
```

will put inside the zip file `redbean.com` the HTML slides `slides.html`. However, doing the same from a browser with plain Javascript brings some complications.

Standard ZIP handling libraries built with Javascript typically create a new ZIP instead of modifying in-place an already existing one. By creating a new ZIP all the business logic powered by Cosmopolitan that allows these files to be executable is lost. This means that trying to add the slides to the redbean file with Javascript will probably lead us to the generation of a normal ZIP that does not have any webserver-related functionalities.

As a hacky way to solve this problem, we implemented a package that manages the ZIP file at byte-level. This package updates an empty `index.html` we assume to be already available within the Redbean file with new content. But before diving into the implementation, it is essential to understand how a ZIP file is constructed.

Zip Structure

As shown in Figure 4.2, a ZIP has three main sections:

- a section containing all file headers and content;
- the central directory;

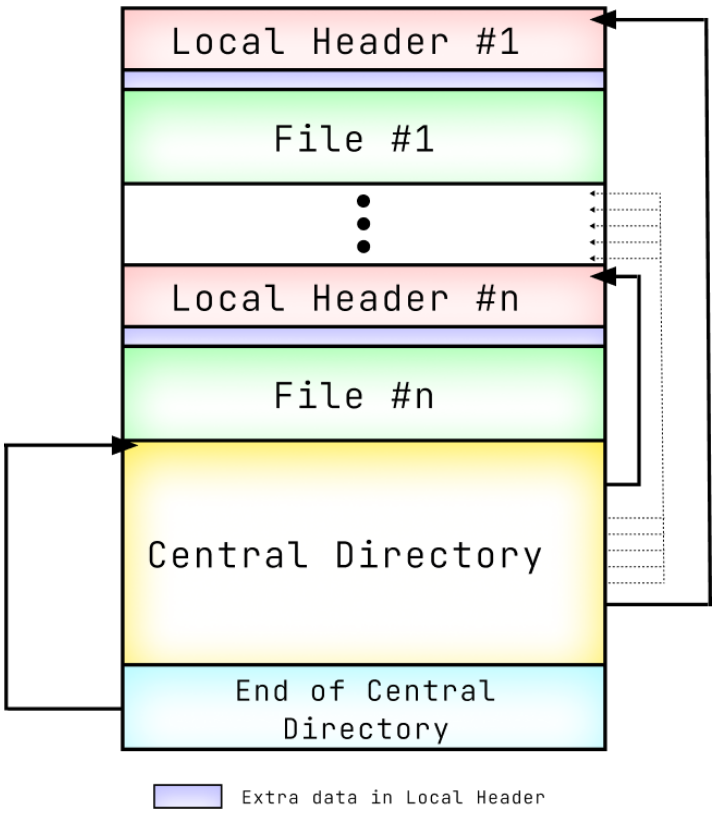


Figure 4.2: Schema describing the ZIP file format

- the end of central directory.

Each file placed in the ZIP file must be preceded by a **"local file header"** record that provides basic information about the file.

In the local file header at offset 6 there is a general purpose bit flag. If its bit at offset 3 is set, then there also must be a **data descriptor** section providing:

- the CRC-32 value of uncompressed data, where the CRC-32 is an error-detecting code that allows to detect changes to digital data.
- compressed and uncompressed file size.

After the local file header, the actual content of the file is found, which can be either compressed or not.

For each file contained in the ZIP there should also be a **"central directory header"** record in the central directory section that provides the location of the corresponding local file header and additional information about the file itself.

Finally we have the **"end of central directory"** record, which mostly gives information to quickly read the ZIP file and find what is needed. For instance, it provides the exact position of the start of central directory, and the total number of files stored within the ZIP.

Each section is uniquely identified via specially-crafted signatures. In detail:

- `0x04034b50` identifies the local file header;
- `0x02014b50` identifies the central directory file header;
- `0x06064b50` identifies the end of central directory.

These signatures can be used to find information about the ZIP file without needing to perform a complete search from top to bottom.

Modify a file within a Zip

Our objective is to modify the ZIP file at byte-level, by inserting into an already present `index.html` the HTML code representing the slides. As a general overview, to perform this update it is needed to:

- add the new content after the local file header;
- re-calculate the CRC-32 value of the file;
- re-calculate compressed and uncompressed size, which in our case will be the same;
- re-calculate the offset to the start of central directory header.

The position of each piece of information can be easily found by exploiting the previously mentioned signatures and by following the ZIP format documentation, which provides the exact offsets for each value. Therefore, given the ZIP string in hexadecimal format, we can find the position of the last local file header record by searching the last occurrence of `504b0304`, and the information we will need to update is found at fixed offsets:

```
const startOfIndexFile = zipString.lastIndexOf('504b0304');
const startOfCRC32 = startOfIndexFile + 28;
const startOfUncompressedSize = startOfCRC32 + 8;
const startOfCompressedSize = startOfUncompressedSize + 8;
const startOfFileContent = startOfCompressedSize + 92;
```

Notice we are taking for granted the `index.html` file is the last one inserted into the ZIP.

At this point we can easily update the file content with the string we want, converted in hexadecimal format, and update the compressed and uncompressed sizes with simple calls to the `replace()` function. To calculate the CRC-32 value of the new string, the algorithm from JSZip and pako - released under the MIT license - has been integrated.

At this point we can get the position of the information we need to update in the central directory header by following the same approach:

```
const startOfCentralDirectoryHeader = zipString.lastIndexOf('504b0102');
const startOfCentralDirectoryCRC32 = startOfCentralDirectoryHeader + 32;
const startOfCentralDirectoryCompressedSize = startOfCentralDirectoryCRC32 + 8;
const startOfCentralDirectoryUncompressedSize =
    startOfCentralDirectoryCompressedSize + 8;
```

and set the new already calculated values.

As a last step, we need to compute the new offset to the start of central directory header, whose calculation is based on the difference between the new and old content. The position of this information can be found as follows:

```
const startOfEndOfCentralDirectory = zipString.lastIndexOf('504b0506');
const startOfOffsetToStartOfCentralDirectory = startOfEndOfCentralDirectory + 32;
```

Finally, the new offset is calculated with the following code snippet:

```
const startOffset = zipString.substring(
  startOfOffsetToStartOfCentralDirectory,
  startOfOffsetToStartOfCentralDirectory + 8
).match(/.{2}/g).reverse().join("");

let newOffset;
if (newContentSize > sizeOfFileContent) {
    newOffset = parseInt(startOffset, 16) + (newContentSize - sizeOfFileContent);
} else {
    newOffset = parseInt(startOffset, 16) - (sizeOfFileContent - newContentSize);
}
```

Obviously, before inserting the newly found value it must be converted to the hex format. The conversion from decimal to hex in little endian is performed in the following way:

```
function decimalToHex(d, padding) {
    if (d < 0) {
        d = 0xFFFFFFFF + d + 1;
    }

    var hex = Number(d).toString(16);

    padding = typeof (padding) === "undefined"
                || padding === null ? 2 : padding;
    while (hex.length < padding) {
        hex = "0" + hex;
    }

    return hex.match(/.{2}/g).reverse().join("");
}
```

Here the first condition deals with negative numbers. Then `d` is converted to hex, padding is applied to make the hex string always be in the correct format and it is finally transformed to little endian.

After this series of step, we have a correctly constructed ZIP file in hexadecimal format. What reimains to do is to convert this representation into a `Uint8Array` that contains the actual binary data. This last step is performed through the following code snippet:

```
return new Uint8Array(zipString.match(/[\da-f]{2}/gi).map(function (h) {
    return parseInt(h, 16)
}));
```

This code finds all occurrences of hexadecimal pairs, converts all these pairs into the decimal equivalent and creates the `Uint8Array` object using its constructor.

At this point the ZIP file will be ready to be downloaded and used as a single-file distributable web server.

Npm package creation

Npm is a package manager for the Javascript programming language. It contains nearly 1 million code packages, and it is completely free to use. When exposing services based on Javascript code, npm is typically the most straightforward way to do it, so we decided to offer the findings of Subsection 4.7.2 as a self-contained npm package called `icp-create-server`.

The only function exposed by the package is `generateRedbeanFile`, which takes as input a string representing the redbean file in hexadecimal format and the slides in HTML and returns a `Uint8Array` with the newly updated ZIP file. The following snippet shows how this package can be used:

```
import { generateRedbeanFile } from 'icp-create-server'

const slides = "<section><h1>Hello world!</h1></section>";

// Get the redbean.com file returned from the build of icp-bundle
const response = await fetch(redbeanUrl);
const file = await response.blob();

// Get the hex string
let zipString = bufferToHex(await file.arrayBuffer());

// Call generateRedbeanFile
const generated: Uint8Array = await generateRedbeanFile(
    slides, zipString
);
```

4.8 Build

Interactive Code Playgrounds is a tool thought to be integrated into HTML pages with a simple script import of the form:

```
<script src="https://path-to-icp-bundle.minified.js"></script>
```

This means ICPs should not be exported as a module: they should be exported as *Immediately Invoked Function Expressions* (IIFE) that prepare the developed web components detailed in Subsection 4.4.2 to be used. Apart from this output, as explained in Subsection 4.7.2 a Redbean file should also be automatically prepared, already including all ICPs dependencies and an empty `index.html` file ready to be updated with actual content.

The following Subsections explain how this is achieved.

4.8.1 Vite

Vite is a tool providing a development server with hot reload features and a build command that bundles the input code with Rollup, a famous module bundler. Vite also provides predefined template presets for multiple framework, including Svelte.

Vite's build process can be configured with various options. First of all, pre-made plugins can be included, and in our case it is essential to use the `svelte` plugin, developed by the Svelte team and fundamental to compile applications built with the Svelte framework. Build options instead can be used to declare additional properties for input and output. The following code snippet shows how editors are typically exported as web components in the ICPs bundle:

```
export default defineConfig({
    build: {
        outDir: 'dist/base/',
        emptyOutDir: false,
        lib: {
            entry: 'src/exports/{language}.ts',
            formats: ['iife'],
            fileName: '{language}',
            name: '{Language}CodePlayground',
        },
        rollupOptions: {
            output: {
                inlineDynamicImports: true,
            }
        }
    },
    plugins: [svelte({
        compilerOptions: {
            customElement: true
        }
    })]
})
```

As we can see the entry point used by this configuration is another configuration file available in the `src/exports` folder. In this folder we have a file for each possible build flow, which simply exports the editor corresponding to the chosen language. For instance, the file `src/exports/java.ts` simply contains:

```
    export * from '../lib/JavaEditor.svelte';
```

This makes the build configuration codebase more modular, and it also allows to easily export multiple editors at the same time.

4.8.2 Gulp

Gulp is a toolklit that allows to automate time-consuming tasks in the development workflow. In our case Gulp is used to perform the build process and prepare the final output folder. The main Gulp script, that can be run with `npm run build`, expects as input a parameter with the language that needs to be built. Gulp will automatically check for the corresponding editor existence and start the build process by using the vite configuration seen before. Some languages need additional dependencies to work offline. For these languages Gulp automatically puts all of them inside the output folder. Finally, it also zips all dependencies inside the Redbean file, while deleting all files that won't be useful for ICPs.

4.9 Optimization Steps

After having decided the target programming language, the build process will create an `iife` minified Javascript file that contains the source code of the editor and of all the dependencies the main file imports. This means that if the code is not properly modularized and differentiated, the build process will automatically include within the generated output the dependencies of the entire project, instead of only the ones of a specific programming language. So it is essential to understand which modules are in common between all editors and which instead are only required for some specific build options.

All the common dependencies are directly included in a specially-crafted web component called `base-editor` that contains:

- the CodeMirror editor;
- the user interface, which is the same for all programming languages;
- the communication with the Workers that happens through message passing.

Language-specific dependencies, such as the Worker responsible for executing code or CodeMirror extensions, are instead imported in the component created for the chosen programming language. For instance, the Javascript Editor's main file has the following structure:

```
<svelte:options tag="javascript-editor" />
<script lang="ts">
    ...
    import BaseEditor from "./BaseEditor.svelte";
    import JavascriptWorker from "../modules/workers/javascriptWorker?worker&
    inline";
    import { javascript } from "@codemirror/lang-javascript";
    ...
</script>
<base-editor
    syntax={javascript()}
    {webworker}
    language="javascript"
    ...
/>
```

This architecture allows the build process to generate outputs strictly consisting of the needed dependencies for each programming language, and nothing more. Moreover, the source code that instead represents the elements in common between all the developed editors is not duplicated, hence a major update would be automatically inherited by all the exported components.

4.9.1 Bundle Output

The build phase of the first bundle version creates the following web components:

- `<javascript-editor></javascript-editor>`;
- `<typescript-editor></typescript-editor>`;
- `<python-editor></python-editor>`;
- `<java-editor></java-editor>`;
- `<sql-editor></sql-editor>`.
- `<processing-editor></processing-editor>`.
- `<standardml-editor></standardml-editor>`.
- `<cpp-editor></cpp-editor>`.

Every component has a set of properties that can be passed inside the tags:

- **code**: the initial code snippet the playground should contain;

- **theme**: light or dark;
- **downloadable**: if set to true, a download button will appear allowing to download the code snippet;
- **type**: type of the user interface, which can be either normal or vertical;
- **id** and **save**: if save is set to true, the code will be cached in local storage with key equal to the given id.

To access the components in a web page, one just needs to include the corresponding script:

```
<script src="{language}.iife.js"></script>
```

and then use it as a normal HTML tag. The editor will automatically get the size of its parent, so it should be created within an HTML element with a fixed size. For instance, the following snippet will create a javascript editor that takes the entire screen:

```
<div style="width: 100vw; height: 100vh;">
  <javascript-editor code='console.log("Hello World!");' />
</div>
```

4.10 Bundle Result

This section provides some screenshots of the final result of ICPs bundle. Figure 4.3 shows an example of a Python playground with default theme, created with the following HTML code:

```
<python-editor contenteditable="true" code="# This program adds two numbers
num1 = 1.5
num2 = 6.3
sum = num1 + num2
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))">
</python-editor>
```

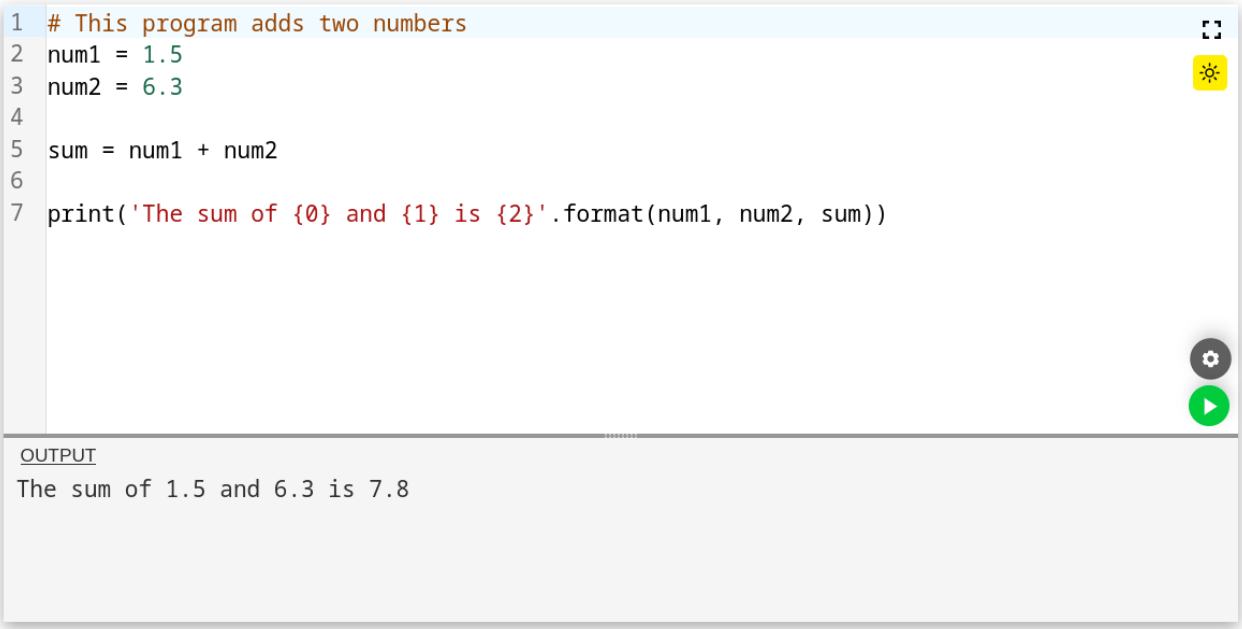


Figure 4.3: Example of a Python Interactive Code Playground

As described in Subsection 4.5.4, it is also possible to define within a code editor which parts should remain read-only, and which parts should instead be editable. Figure 4.4 shows the result of a playground where only the printed message can be modified by the user.

Figure 4.4: Example of a dark and vertical Java Interactive Code Playground, where only some parts of the code are editable

As shown in Figure 4.5, the settings button provides few additional options. You can unlock the read-only code, cancel any modifications applied to the code, enable/disable Tabs handling and copy the snippet to the clipboard.

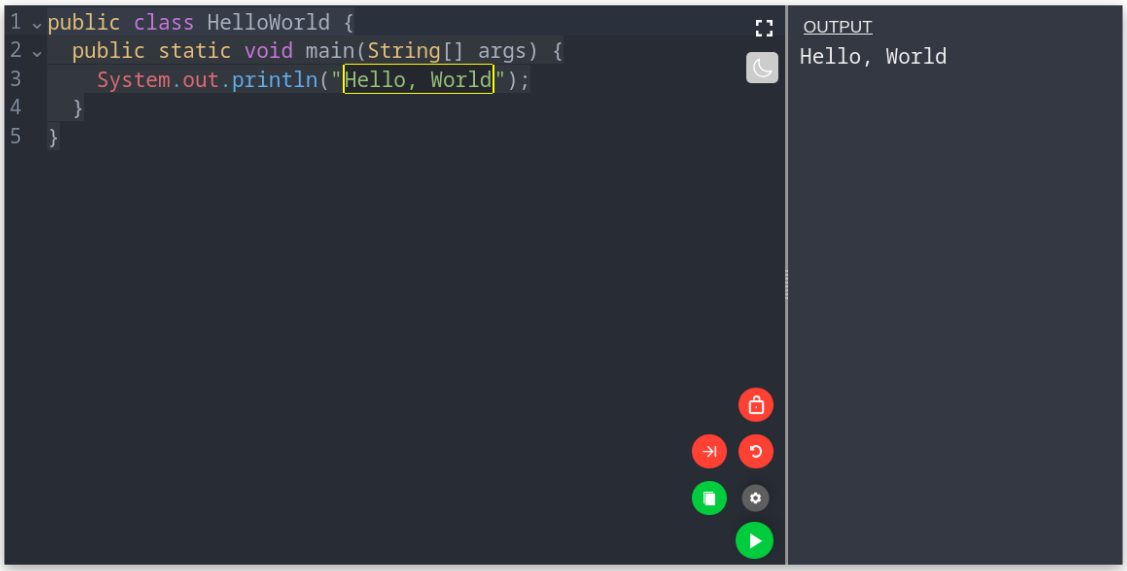


Figure 4.5: Java Playground with opened settings

Subsection 4.6.2 describes how results from a `SELECT` statement are automatically mapped to an HTML table. Figure 4.6 shows a SQL Interactive Code Playground containing an example of a resulting table.

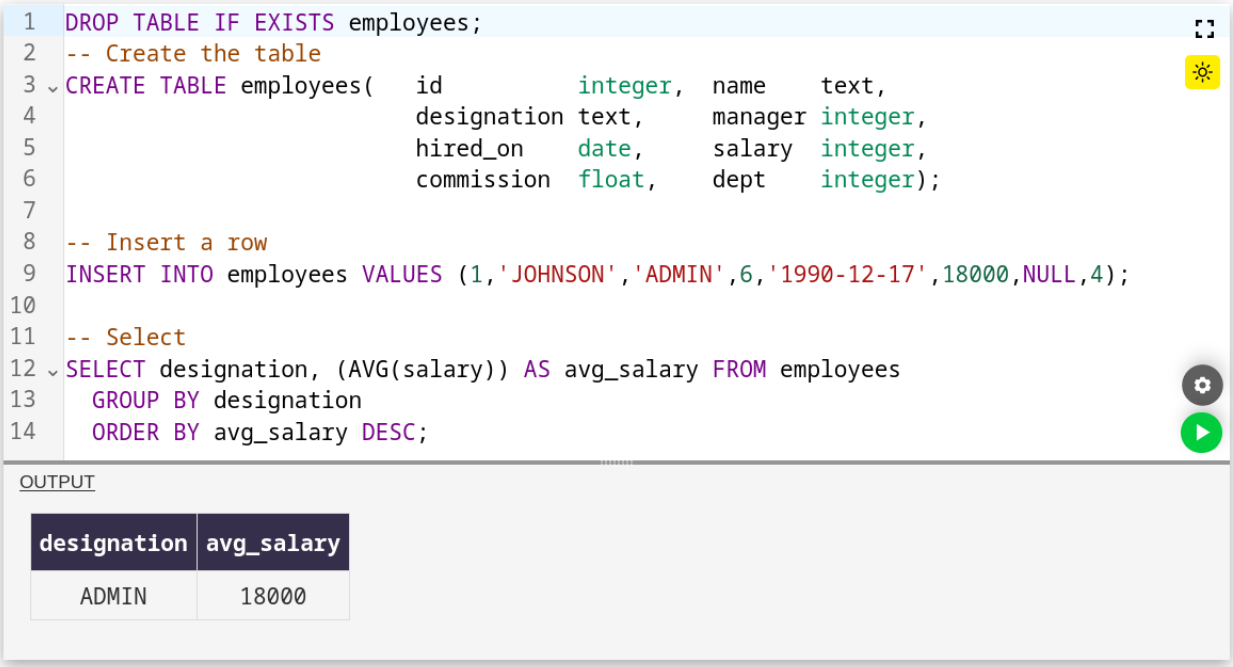


Figure 4.6: Example of a SQL Interactive Code Playground

4.11 HTML Slides System

The slideshows are one of the main types of material exploited during oral presentations, used as a way to fit images or to mark important concepts. In particular, they are often vital to teachers for instructional purposes, where they not only accompany the explanation, but they are proposed as a form of didactic material. Slideshows have actually been around for nearly a century, and they originally consisted of a series of individual photographic slides projected onto a screen. Nowadays they quite evolved in terms of type of storage, but not in terms of interactiveness: they are still static by definition.

With the Interactive Code Playgrounds project, we would like to propose an enhancement of slides specifically created for programming classes, in which each slide can become interactive, in the sense that a student accessing it will be able to edit, compile and run code snippets without ever leaving the provided environment. We have just described a bundle offering playgrounds easily integrable into any web page. How can we use it to create interactive slides? The solution relies on the use of an HTML slides system.

4.12 Presentation Framework

One of the most widely used HTML presentation frameworks currently available as open-source projects is `reveal.js`. Thanks to a specially-crafted bundle, a page importing this framework can access a set of APIs and CSS styles to create fully featured slideshows inside a web page, by simply using HTML code. The main exposed features are:

- **fragments**: highlight or incrementally reveal individual elements on a slide;
- **links**: create links from one slide to the other;
- **code**: present static code with syntax highlighting;
- **vertical slides**: expand slides both horizontally and vertically;
- **PDF export**: export presentations as PDFs.

Figure 4.7 shows an example of a typical reveal.js slide.

Figure 4.7: Example of a Reveal.js slide, containing a code snippet

4.12.1 Integration Problems

The integration of CodeMirror editors inside reveal.js has proven to be quite cumbersome. Without any fix, it is impossible to correctly navigate the code within the editor by using the keyboard or the mouse. All these complexities mainly derive by the fact that reveal.js applies by default some CSS transformations to the HTML elements put in the slides, which are not compatible with CodeMirror[6]. I managed to solve the problem by disabling the default reveal.js slide layout, which consists of styles used to scale and center slides content. Without it, slides will not be responsive by default, hence a custom specially-crafted style must be implemented.

The custom CSS code developed to center all elements inserted into the slides simply applies a `flex` display with `justify-content` and `align-items` properties set to true. Handling responsiveness, instead, is far more challenging: we want slides to be usable on any device, with a fixed ratio approximately set to `16:9`. Using a unit measure dependent on the screen width would be a working solution for all devices having a landscape orientation. However, it is essential to also take into consideration all devices - such as smartphones - that use a portrait configuration. The solution found to solve this issue for all kinds of screens is to use a unit measure that considers both the screen width and screen height. In detail, each measure will be dependent of the minimum between the value expressed in `vw` (viewport width) and the value expressed in `vh` (viewport height).

For instance, if we wanted an element to take the full available height, it should have the CSS `height` property set to `min(55.5vw, 100vh)`. The second parameter is 100vh since we want the element to take the entire viewport height. The first parameter is calculated based on the wanted fixed ratio. In our case, the ratio should be `16:9`, so its value becomes: `(100*10/18)vh = 55.5vw`. On a screen with landscape orientation the minimum between the two values will clearly be `100vh`, and the height of the element will be the expected one. On a screen with portrait orientation, instead, the minimum between the two values will be `55.5vw`, hence the element will be much smaller, and slides will appear as a `16:9` rectangle on the center of the screen, with all elements appropriately scaled. Figure 4.8 shows how the scaling system works when using this kind of unit measure.

Creating slides while using this unit measure may be a bit cumbersome, considering all additional calculations that must be performed. However, as a trade-off, we can think of using an aspect ratio of `16:8`: despite being a bit unusual, it allows us to base all calculations on simple divisions or multiplications by 2, which makes the entire process more straightforward.

After having defined this system, we created a CSS file with custom styles that exports the definitions of the most important elements typically used within a web page following the designed unit measure, namely: `p`, `a`, `hx`, `ul`, `ol`, `li`, `table`, `code`. Additionally, a set of useful CSS classes have been defined that may help creating typical components, such as titles, subtitles, code containers, rows, columns, etc. As an example, following is the override of `p` elements:

[6]CSS Transform Issue: https://github.com/codemirror/dev/issues/324; Integration into reveal.js Issue: https://github.com/hakimel/reveal.js/issues/1735

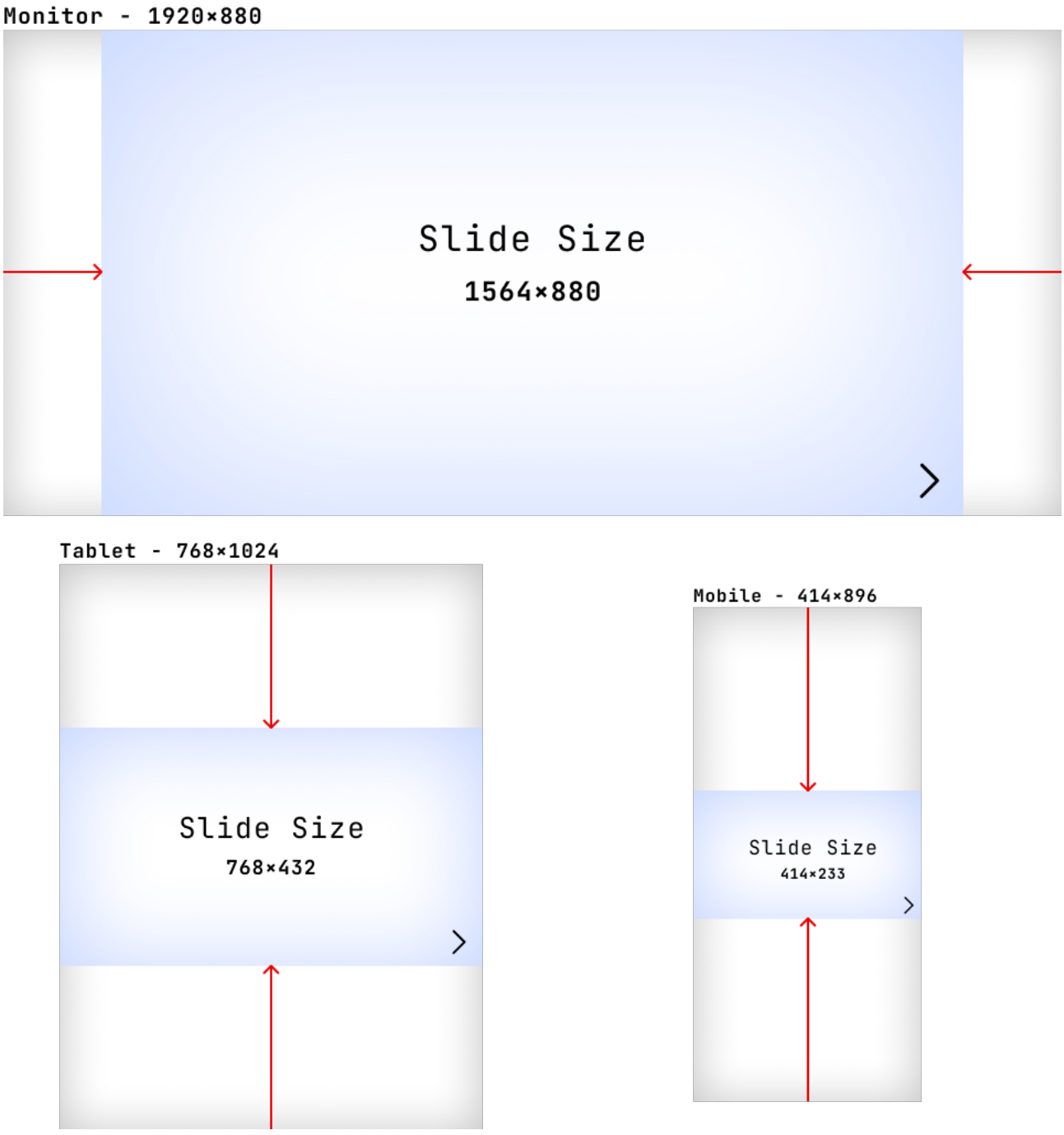


Figure 4.8: Overview of how the developed scaling system works on different screen sizes

```
p {
  font-size: min(1.6vw, 3.1vh) !important;
  margin: min(0.5vw, 1vh) 0 !important;
}
```

4.12.2 Slides Preparation

Thanks to the various APIs and functionalities `reveal.js` offers, most of the PDF slides can be converted into this system. The only limitation is given by the fact that elements cannot be easily dragged and dropped where we want, since HTML works a bit differently. In those cases, however, it is possible to include Scalable Vector Graphics (SVGs), which can also be shown and hidden by using the `fragments` feature, leading to an effect similar to the typical animations you find on PDF slides. Some examples of slides manually converted by myself can be found on GitHub at `https://github.com/lucademenego99/icp-slides`.

As previously explained, Interactive Code Playgrounds can be added into any web page by importing the bundle and using the constructed web components. The process to include them into `reveal.js` slides is quite similar, but it is essential to set these frameworks up correctly, in order to make them compatible between each other. Apart from the ICPs bundle, the custom CSS styles presented in Subsection 4.12.1 must be imported into the page. Additionally, `reveal.js` must be also imported and initialized by including at least the following options:

```
Reveal.initialize({
  help: false,           // Do not display an help overlay when ? is pressed
  history: true,         // Push each slide change to the browser history
  keyboard: true,        // Enable keyboard shortcuts for navigation
  width: "90%",          // Set the width of the slides
  height: "90%",         // Set the height of the slides
  center: true,          // Vertical centering of slides
  touch: true,           // Enables touch navigation on devices with touch input
  backgroundTransition: 'fade', // Slides transition style
  transition: 'fade',    // Select the type of slides transition
  disableLayout: true    // Disable the default layout (centering and scaling)
});
```

The last option, in particular, is quite important: it allows us to disable the default reveal.js layout that would break CodeMirror.

The last update that needs to be applied to CSS styling regards the PDF export feature offered by reveal.js. Even in this case, since we disabled the default layout we need to add some specially-crafted CSS properties, but only when the user asks for this slides version. To do this, the following script must be included at the end of the HTML file:

```
<script>
    if (window.location.href.indexOf('?print-pdf') > -1) {
      // Create a new <style> HTML element
      var css = document.createElement("style");
      // Set the custom style to the CSS element
      css.innerHTML = " ...CUSTOM CSS STYLE... ";
      // Append the css to the document head
      document.head.appendChild(css);
    }
</script>
```

With these configurations and the correct imports, ICPs can finally be included into reveal.js slides. For instance, a slide with some text on the left and a playground on the right can be created with the following HTML template:

```
<section>
  <h3 class="title">Exercise 1</h3>
  <div class="row">
    <div class="col">
      <p>You are given two variables a and b.</p>
      <p>Write a function that swaps their values.</p>
```

```
      </div>
      <div class="col small-code">
        <python-editor contenteditable="true" code='a = 3
b = 5
# Write here...'></python-editor>
      </div>
    </div>
  </section>
```

Figure 4.9 shows the slide resulting from this template.

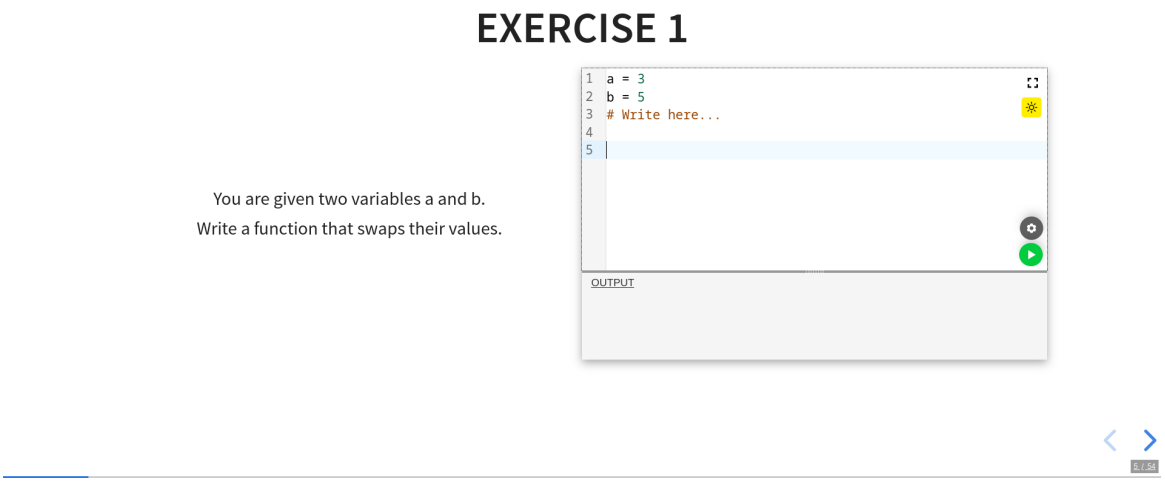


Figure 4.9: Example of a reveal.js slide coupled with ICPs

4.13 WYSIWYG Editor

We realize that creating didactic material with HTML code may be complex, especially for instructors who have no experience in web development. To address this complexity we developed a web *"What You See is What You Get"* (WYSIWYG) editor that allows to create ICPs-compatible slides by relying on a visual interface.

This editor provides a set of possible layouts to choose from for each slide. In each section of a given layout users can insert text, an image or an Interactive Code Playground.

The **text editor** is powered by Quill, an open-source project used by many top-tier companies, such as Microsoft, Linkedin, Slack and Grammarly. We configured Quill's toolbar to provide various options to modify the selected text: increase its size, make it bold/italic/underlined, change its color and background, make it a link, format it as a code snippet or a quote.

When inserting an **image**, the editor will automatically ask the user to put an alternative text that describes what the image is about. Most people do not even know how important an alternative text is for someone, and by automatically requesting it we are enforcing anyone using the slides to create accessible material.

The **Interactive Code Playgrounds** put within a slide will appear empty, but functioning. This means that while completing a slide with some custom code, instructors will be able to quickly test whether the presented snippet works or not. The programming language is chosen through a selection box available in the navigation bar.

A typical feature needed when preparing a slide deck is the possibility to reorganize the slides order. We developed an **overview mode** that allows users to see all slides in the current order and to modify the ordering via a drag-and-drop system. Since slides in `reveal.js` can be presented both horizontally and vertically, users are not limited on the x axis: slides can be reordered as in a grid system.

Last but not least, a **theme button** can be used to toggle between the dark and the light mode, which affect both the slides graphics and the code playgrounds.

4.13.1 Editor Setup

We developed this web editor with Svelte, the same technology used for building the ICPs bundle. Styles are mainly configured with `Tailwind`, a CSS framework built to substitute typical CSS styles with pre-determined CSS classes.

To manage the application's state we used Svelte store, which allows to define a set of variables accessible by all components. The store mainly contains information about the slides written by the user: when the user requests an export, from the information provided by the store the application automatically generates the HTML code representing the slides.

4.13.2 Reveal.js Integration

We integrated `reveal.js` by using its official NPM package. After the usual initialization performed through the exported `Reveal` instance, slides are dynamically created based on the store information. The following snippet shows how this is achieved with Svelte:

```
<div class="slides">
    {#each $revealSlides as verticalSlides, index (index)}
        <section>
            {#each verticalSlides as slide (slide.id)}
                {#if slide.layout == Layouts.BODY}
                    <LayoutBody bind:slide />
                {:else if slide.layout == Layouts.MAIN}
                    <LayoutMain bind:slide />
                {:else if slide.layout == Layouts.COLUMNS}
                    <LayoutColumns bind:slide />
                {/if}
            {/each}
        </section>
    {/each}
</div>
```

Each layout component shown in the snippet can be rendered in two ways by the application. If the user has already designed the slide, it is simply shown. Otherwise, the user is presented with a selection screen where to choose which elements to add. As soon as the slide object available within the store is modified, Svelte automatically updates the user interface appropriately.

To create a new slide a user can use two specially-crafted buttons: one adding a slide horizontally and one adding it vertically. Even in this case, from a development standpoint, the button simply adds an empty slide in the store, and the Svelte framework takes care of the update `reveal.js`-side.

4.13.3 ICPs Integration

The ICPs Bundle described in Section 4.4 was integrated into this project using NPM. Since the bundle is exposed in `IIFE` format, we cannot directly import the required web component as a module. We managed to install all the available web components by importing the full bundle and inserting it into a global script:

```
<script lang="ts">
    import bundle from 'icp-bundle/dist/base/full.iife.js?url';
    ...
</script>
...
<svelte:head>
    <script src="{bundle}"></script>
</svelte:head>
```

As described in Section 4.11, a custom CSS styling must be integrated to make ICPs work in reveal.js. This is achieved in the same way: the CSS code is imported and appended in the `<svelte:head>` section.

4.13.4 Slides Export

As previously mentioned, all slides information is stored in a variable containing a 2D array of objects composed by:

- `ID`: slide identifier, expressed as UUIDv4;
- `indexH` and `indexV`: slide indices;
- `layout`: chosen layout for the slide;
- `template`: class used to generate the slide HTML.

Every template has a set of functions used to generate the HTML code for a particular element. For instance, `buildText`, `buildImage` and `buildCode` are functions inherited by all templates, used to create the main slides components. The entire slide HTML code is instead returned by the function `generateHTML`, common to all templates. When a user requests the slides export, the template's `generateHTML` function is called, and the complete HTML is created by composing a fixed header and the constructed body.

There are three export possibilities:

- export a single HTML file that requires an internet connection to properly work;
- export a Redbean executable file, which works without an internet connection;
- export the slides and all the files required to self-host the didactic material.

Exporting the content following the first and third options is quite trivial: it is just a matter of manipulating variables and generating HTML code. To export the Redbean file instead, as explained in Section 4.7, we need to use the `icp-create-server` NPM package and its exported function `generateRedbeanFile`. In detail, this is achieved with the following code, run from a `WebWorker`:

```
// Fetch the redbean file
const response = await fetch(redbean);
const file = await response.blob();

// Get the hex string
let zipString = bufferToHex(await file.arrayBuffer());

// Generate the new redbean file exposing the slides
const generated = await generateRedbeanFile(
    e.data.slides,
    zipString
);

// Send the generated redbean file to the main thread
postMessage({
    "generated": generated,
});
```

where `bufferToHex` is a function converting an `ArrayBuffer` to a hex string:

```
function bufferToHex(buffer) {
    return [...new Uint8Array(buffer)]
        .map((b) => b.toString(16).padStart(2, "0"))
        .join("");
}
```

4.13.5 Editor in Action

The final version of the editor can be seen in Figures 4.10 (light mode) and 4.11 (dark mode). This single-page web application contains a navigation bar with various configuration options and an embedded `reveal.js` configurable slide deck.

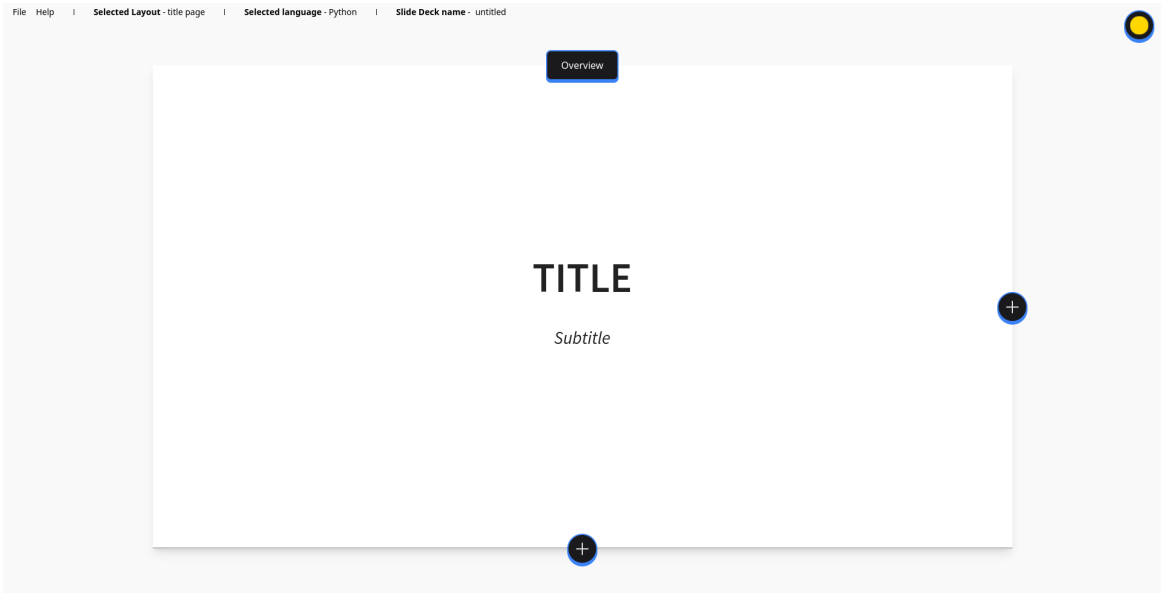


Figure 4.10: Main page of the editor, light theme

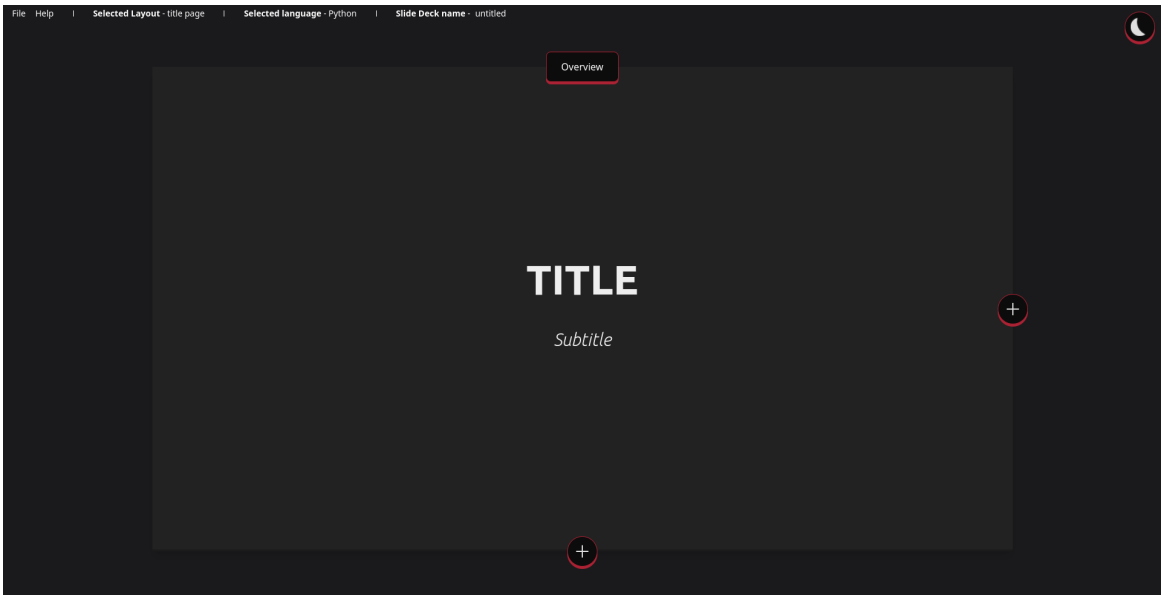


Figure 4.11: Main page of the editor, dark theme

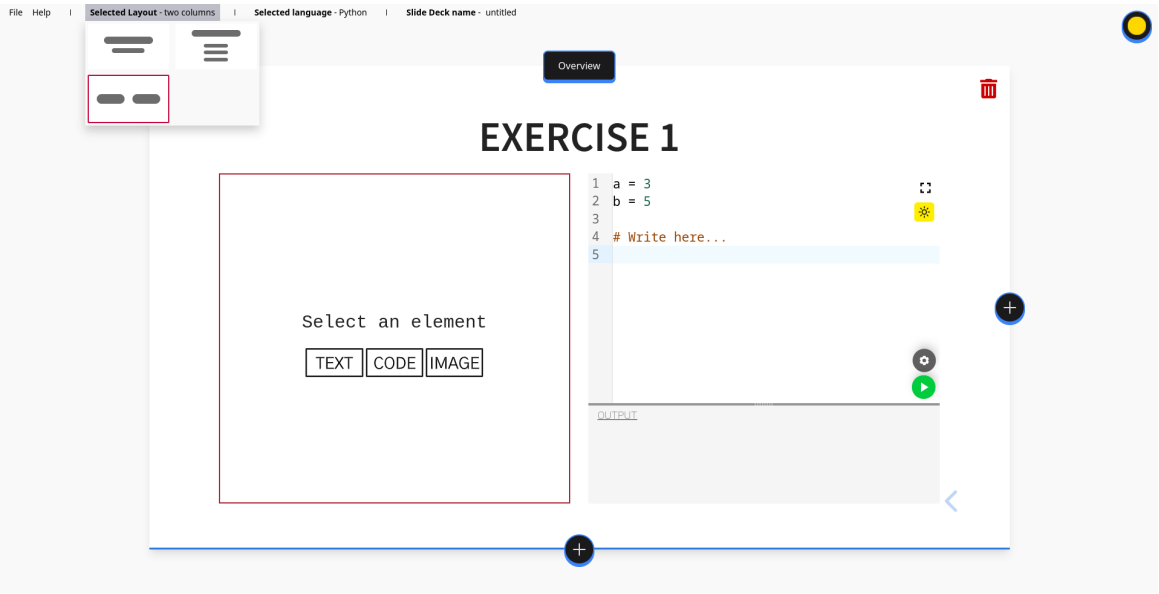


Figure 4.12: Example of how layouts can be used to style slides

The *"plus"* buttons can be used to create new slides, and for each slide the user can choose from the navigation bar which layout to use. For example, Figure 4.12 shows a two-columns slide being built, with a Python code playground on the right.

The overview mode can be activated by clicking on the button above the slides. This mode shows all the created slides, and by dragging and dropping slides can be reordered.

Figure 4.13: Slides overview offered by the editor

Finally, Figure 4.14 shows the three previously discussed possibilities for exporting the slides.

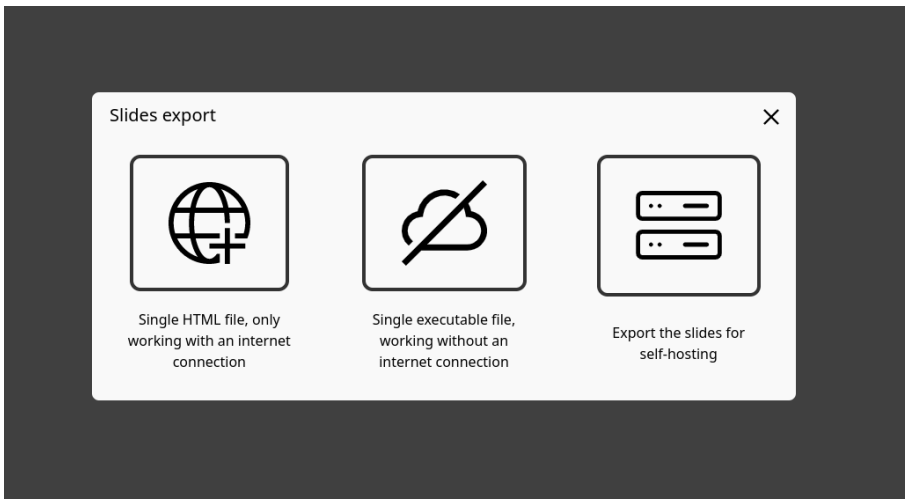


Figure 4.14: Slides export alternatives

5 Methodologies and Methods

The ICPs slides system provides students a simple, ready-to-use development environment, without requiring set up or installation steps. Our goal is to minimize the amount of technical difficulties encountered by students, especially those learning to program for the first time. Hence, introductory programming courses are ideal contexts in which to evaluate this tool. Checking whether students with no prior IDE experience can autonomously understand how to use this simplified environment is fundamental to our research. At the same time, evaluating the tool in more advanced settings could allow us to deepen and generalize our research.

5.1 Research Contexts Selection

The alpha version of ICPs was ready on July 2022, and we first tested it during a pilot study for an intensive Python introductory course taught at the Department of Sociology and Social Research. This was a relatively small context, where we only followed a few lectures.

After a second development cycle, a new, bigger batch of field tests has been conducted in February 2023. We first identified the intervention scale, by limiting our research to approximately ten lectures. Afterwards, we had to determine whether to focus all the slots on a single course, or to perform a horizontal study by spreading them across multiple courses. We ultimately opted for the latter horizontal approach for several reasons. First of all, by choosing it we are maintaining the structure of the preliminary study. Moreover, this approach allowed us to diversify the intervention contexts. This is particularly important in a technology education setting, where research typically has a limited replicability in different contexts [6].

To identify possible courses in which ICPs could be employed, we searched course offerings related to computer science using the "Didactic Activities Search" functionality provided by the *esse3* system[1]. For logistical reasons, we limited our scope to the University of Trento, so we could observe students interacting with the tool in person.

Upon examining the course offerings for the second semester, it became clear that there is a scarcity of programming-related courses: Table 5.1 shows the complete list. Furthermore, two of them had to be excluded from consideration. Firstly, "Data Visualization Lab" cannot employ ICPs due to our Python runtime's lack of graphical output options. Secondly, although "Informatica ed elementi di

[1] `https://www.esse3.unitn.it/Guide/PaginaRicercaInse.do` - Last Accessed 21/04/2023

Course	Department	Programming Language
Laboratorio di Programmazione	Mathematics	Java
Informatica ed Elementi di Programmazione I	Psychology and Cognitive Science	Processing
Informatica ed Elementi di Programmazione II	Psychology and Cognitive Science	Javascript
Introduzione alla programmazione con Python per le scienze sociali	Sociology and Social Research	Python
Computer Programming 2	Information Engineering and Computer Science	Java
Functional Programming	Information Engineering and Computer Science	PolyML
Programmazione Funzionale	Information Engineering and Computer Science	StandardML
Data Visualization Lab	Information Engineering and Computer Science	Python

Table 5.1: List of programming-related courses we found taught during the second semester at the University of Trento

programmazione II" offers a brief introduction on Javascript, a language compatible with ICPs, it may not be the most suitable choice as Javascript can be inherently executed in a browser, so our tool does not really shine in this scenario. After all, many Javascript playgrounds and similar technologies already exist.

Despite the limited number of available courses, we believe we have more than enough room for manoeuvre, as these courses align well with the kinds of programming classes we aimed to study. Specifically, they cover both introductory courses and a few advanced ones for experienced students.

5.2 Contacts

Our contact strategy was to email the main professor of each course, introducing ICPs and providing a link to a demo[2]. The email's main purpose was to arrange a brief introductory meeting where we could showcase the tool and answer any questions.

We reached out to six leads initially. We did not receive a response from one of these contacts, while another was not yet ready to use our tool as they were still preparing the course and changing its structure would have been too complicated. However, we reached an agreement to use the tool for a few lectures of the following courses.

The **"Functional Programming"** course uses PolyML, an independent implementation of StandardML with additional features that typically produces slightly different outputs. We could not quickly integrate PolyML into ICPs, so we used StandardML instead. The professor told us from the very beginning that using another language would not be ideal, even if it is very similar. However, they decided to give it a try. Unfortunately, our experience with this course was limited to only one lecture where ICPs were not even fully exploited. As we expected, the existence of differences between these languages was clearly an issue for an introductory course on functional programming. Thus, we will not present any results deriving from this experiment.

Regarding the **"Informatica ed Elementi di Programmazione I"** course, the professor was concerned about whether ICPs could be effectively used with the Processing language. To address this concern, we integrated Processing - as described in Chapter 4 - and provided a demo a few days later showing how a lecture with our tool might look. The professor then accepted to meet with us online, and after asking for more details about our research approach, he agreed to use this alternative slides system.

The professor teaching **"Introduzione alla programmazione con Python per le scienze sociali"** actually contacted us first, after being recommended to do so by the professor who allowed us to conduct our pilot study. We had an online meeting to talk about the potential use cases for ICPs, and as a result, the professor agreed to use our tool for two laboratories.

The professor for **"Computer Programming 2"**, who was also a supervisor for my Research Project, was already aware of ICPs. After reviewing some demo material we prepared, consisting of various exercises that could benefit from the interactivity this project provides, the professor agreed to use it for some lectures.

The literature review in Chapter 3 shows how the development of a non-disruptive tool such as ICPs may be the solution to introducing ICT into education. ICPs is a simple slides system that does not significantly alter teaching materials or require changes to teaching methodology. Therefore, most

[2] `https://lucademenego99.github.io/icp-slides` - Last Accessed 25/04/2023

professors agreed to use it without much consideration, as it resembles familiar slides but with minor changes.

5.3 Performed Interventions

In total, we conducted four interventions as part of our research, with me being responsible for all of them except for the pilot study. These interventions involved attending lectures for several courses, including the pilot study, "Informatica ed elementi di programmazione I", "Seminario: Introduzione alla programmazione con Python per le scienze sociali" and "Computer Programming 2". Unlike the intervention regarding the "Functional Programming" course, these interventions proceeded smoothly, allowing us to gather all the necessary information for our research. Figure 5.1 shows a picture of a lecture employing our tool.

Figure 5.1: Picture of a lecture of "Informatica ed elementi di programmazione I" using ICPs

5.3.1 Research Methodology

This study employs a mixed-methods research approach, which integrates both qualitative and quantitative methods. Halcomb *et al.* state that mixed methods research *"capitalises on the strengths of both qualitative and quantitative analysis, whilst ameliorating their weaknesses to provide an integrated comprehensive understanding of the topic under investigation"*. Merriam (1998) further argues that field observations can function as a form of data triangulation to substantiate the findings. However, Scammon *et al.* (2013) caution that employing mixed methods is not always the optimal choice, and the decision should be based on the added value gained from combining qualitative and quantitative methods over using a single method alone. [17, 29, 37]

In our case, we believe that firsthand observation of students' experiences using ICPs would add a significant depth to the research. Technical difficulties and obstacles students face when learning to use the slides would be better documented through field notes than with numerical data collected from surveys. However, another form of data collection, such as surveys, remains necessary to triangulate results. In fact, field observations conducted by a single researcher may not exhaustively represent the context reality, or may even be biased, despite efforts to remain impartial and nonjudgemental. Furthermore, Johnson *et al.* (2003) note that students may be influenced by the researcher's presence in the classroom, making the integration of another methodology essential. [21]

Consequently, the overall methodology we employed incorporates two methods: surveys and field observations, in a form of micro ethnography.

Surveys

In designing the surveys, we followed Richards *et al.*'s recommendation (2002), by striving for valid, reliable and unambiguous questions [34]. Section 5.4 provides more information about the proposed

questions.

There are various types of surveys one could employ, each with its strengths and weaknesses. Specifically, three main survey formats can be identified: closed-ended or structured, open-ended or unstructured, and a combination of both. Seliger *et al* (1989) argue that closed-ended surveys are more efficient and easier to analyze, while Gillham (2000) contends that open-ended surveys offer *"a greater level of discovery"*. In light of these perspectives, Zohrabi (2013) comes to the conclusion that any survey should include both open and closed questions, to complement each other [11, 38, 49]. Following this last piece of advice, we chose a mixed format survey, including both open-ended and closed-ended questions.

Brown (2001) distinguishes between two primary methods of administering surveys: self-administered, usually mailed to respondents, and group-administered, in which the survey is presented to all respondents at one time and place [5]. The latter method is often considered superior, as it allows the researcher to clarify any unclear questions, gain a better understanding of the environment in which the survey was filled, and it commonly results in a higher return rate [49]. Consequently, this study adopts the group-administered approach, asking students to complete the survey before or after a designated lecture.

Micro Ethnography

A participant field observation, akin to an ethnographic study, is adopted, wherein the researcher enters the classroom and directly engages with students. Burns (1999) notes that in this setting *"the researcher becomes a member of the context and participates in its culture and activities"* [7]. Hence, this approach may help to alleviate the potential issue of students altering their behaviour due to the researcher's presence.

This form of analysis based on classroom observations is no stranger to educational settings, and more importantly, combined with surveys, it is identified by Johnson *et al.* (2003) as a way to collect *"relatively objective firsthand information"* [21, 31, 49]. In fact, in our research field observations are used to triangulate results: while in surveys students report their own experience and comments, we use field notes to cross-reference the data and see whether what can be observed in class matches the students' narrative.

Fraenkel *et al.* (2003) recommend choosing between "narrow focus" and "broad focus" field observations [10]. A broad focus approach is selected in this study, as it helps to catch as much information as possible. Additionally, recording everything and carefully reviewing it later may improve the reliability of data by reducing personal bias. Thus, the followed methodology involves a first and broad-focus note-taking step performed directly in class, and a complete review performed afterwards.

During the field observations, the four categories of classroom activities identified by Ross (1992) are documented: student activities, sources of input to students, student behavior and the distribution of classroom time [35]. Having a clear understanding of what the researcher has to record supports the note-taking process performed on the spot, which Johnson *et al.* (2003) emphasize as critical [21].

LeCompte *et al.* (1982) suggest that clarifying the researcher's status can increase the external reliability of the study, which pertains to its replicability [24]. Therefore, we want to explicitly state that in-class observations were conducted solely by me, the developer of ICPs and a Master's student in Computer Science at the University of Trento, working on his dissertation.

5.3.2 Data Analysis

After the data collection is completed, the next step is data analysis. The data analysis process involves cleaning, organizing and examining the collected data to identify patterns and trends, while drawing meaningful conclusions.

The cleaning process involved the creation of an anonimized table containing all results from the initial and final surveys, joined based on the students' emails. All Likert-scale questions' results were converted to numerical values, while open questions have been transcribed. Students who have not given their consent on the first survey have not been taken into consideration.

The second step involved a qualitative analysis of the results collected from open questions and of the field notes taken while observing students interacting with ICPs. We first broke down the

surveys' answers into smaller units, or codes, representing recurring themes. Afterwards these codes have been grouped into categories, based on their similarity in meaning. Sometimes the questions were too general, and students did not completely understand where to insert a consideration. For instance, in some cases students talked about missing technical features in the comments section instead. Consequently, during this process we also moved all answers in the correct sections. Finally, all codes and categories have been inserted into tables and sorted based on their occurrences.

A similar procedure has been employed for field notes. By using Taguette[3], an open-source qualitative analysis tool, we identified the main recurrent themes by organizing them into categories. After having organized and commented the most recurring ones, we performed a cross-reference by comparing these results and the codes identified in the surveys.

Once the coding process was completed, we analyzed data using a combination of descriptive and inferential statistics. We employed descriptive statistics to summarize the data and identify trends. Inferential statistics allowed us to test our hypotheses and draw conclusions about the relationships between variables.

All closed questions were grouped into categories and organized through a series of measurements of central tendency, namely average, median and mode. The standard deviation of the population was calculated with the excel function `STDEV.P`, which uses the formula:

$$STDEV.P(x, N) = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N}}$$

The confidence interval, instead, was calculated by using the excel function `CONFIDENCE.T`, which uses the Student's t distribution to return the confidence interval for a population mean, such that a value picket at random from the data has 95% probability of lying within the mean plus or minus the calculated width of half the confidence interval. By calculating the confidence interval in this way we are assuming the population distribution is normal, which may not be true in this case. However, due to the central limit theorem, the calculated values should still be approximately correct when the sample size is greater than 30. Hence, when the sample size was smaller, we did not take into consideration the resulting value.

In some cases we tried to identify patterns and correlations between variables. In these cases, we first employed the `CORREL` excel function to find the correlation coefficient between two variables, and then we performed a `T.TEST` to calculate the p-value and evaluate whether the correlation was statistically significant. In detail, the `CORREL` function calculates:

$$CORREL(X, Y) = \frac{\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \overline{x})^2 \sum_{i=1}^{N}(y_i - \overline{y})^2}}$$

while the `T.TEST` returns the probability associated with a Student's t-Test.

Finally, we merged all information coming from both qualitative and quantitative analysis, by performing data triangulation. This allowed us to substantiate the findings and discover the most recurring themes.

5.3.3 Research Contexts Description

This Subsection aims to provide a detailed description of the research contexts that were followed in the study.

Pilot Study - Introduction to Programming

This course of 6 CFU is an intensive course taught at the Department of Sociology and Social Research, consisting on multiple lectures conducted in a brief time frame. This course introduces to programming using the Python programming language, following the didactic material proposed by the SoftPython

[3] `https://www.taguette.org/` - Last Accessed 23/05/2023

website[4]. Being an introductory course, students approaching it typically do not have prior experience in programming.

Approximately twenty students were enrolled on the course. The lectures were mainly theoretical, but they also had various interactive examples shown from a Jupyter Notebook[5].

Informatica ed elementi di programmazione I

This course of 12 CFU is part of the Bachelor's in Interfaces and Communication Technologies, taught at the Department of Psychology and Cognitive Science. It is followed during the second semester of the first year. This course is a simple introduction on programming and algorithms, focusing on the Processing language. Students are only required to have a basic maths knowledge, and nothing more. Hence, students following this course will not probably have coding experience.

Approximately fifty students are enrolled on the course. The lectures merge theory and exercises, by introducing concepts and asking students to immediately start solving some problems and writing code. ICPs will be used for four lectures and a half, where the instructor will talk about conditional statements and iterations.

Seminario: Introduzione alla programmazione con Python per le scienze sociali

This course of 3 CFU is a seminar of the Department of Sociology and Social Research, and students can decide in which year to take it. It is an introduction to programming taught in Python, presenting the programming language syntax and its main constructs. Being the concepts so basic, we expect students following this course to have no experience in programming.

Approximately twenty students are enrolled on the course. The lectures are mainly theoretical, but they also have various interactive examples shown from a Jupyter Notebook. There are however two laboratories in which students will have the ability to test their knowledge. ICPs will be used specifically during these laboratories.

Computer Programming 2

This course of 6 CFU is part of the Bachelor's in Computer Science and in Computer, Communications and Electronic Engineering, taught at the Department of Information Engineering and Computer Science. It is followed during the second semester of the first year. Hence, as in the case of Functional Programming, students attending it already have experience in programming. This course provides fundamental concepts characterizing Object Oriented Programming, and the chosen programming language is Java.

Approximately twenty students are enrolled on the course. Lectures are divided in two formats: theoretical and laboratories. ICPs will be used during two theoretical lectures about inheritance and polymorphism.

5.4 Methods

We developed two surveys: one administered before the experiment, and one right after. The following Subsections provide in-depth information about each one. The complete list of questions, instead, can be found in the Attachments.

5.4.1 Pre-experience survey

The pre-experience survey begins with a consent form that explains how respondent's data will be analyzed and presented in external settings. It also emphasizes that the use of ICPs is optional, and that participating in this experiment only means providing consent for us to observe the class and collect survey responses.

The second section of the survey mainly collects demographic information. Name, last name and study course are required to match data with the follow-up survey. It also asks about respondents'

[4] `https://it.softpython.org/` - Last Accessed 25/04/2023

[5] `https://jupyter.org/` - Last Accessed 25/04/2023

feelings toward computers and programming, their motivation level, any difficulties installing required tools, perceptions of the most challenging aspects of learning programming, and which aspect they expect they will spend most time on.

The third section briefly presents the ICPs tool, showing screenshots and verifying whether respondents can understand at a glance how it works. It also asks about how important respondents consider various features that an interactive slides system might have.

5.4.2 Post-experience survey

The post-experience survey follows up on respondents' initial perceptions about the most difficult aspects of learning programming, as a way to determine whether these perceptions were accurate. It also asks how much respondents used the interactive slides, and to rate their quality on various aspects. Furthermore, it investigates the importance of certain features for a slides system, and how helpful respondents found those features in ICPs. It asks which development environment respondents preferred, and how many technical problems were encountered with each. Finally, open-ended questions invite general comments, while also asking whether any bugs were found in the system.

6 Results

Programming classes in Higher Education currently rely on complex and non-standardized tools, demanding significant computational resources. Latest web technologies can be used to simplify pedagogy, providing students with a hassle-free and accessible learning experience. Additionally, the combination of Javascript and WebAssembly (WASM) enables the development of frontend tools that eliminate the need for a backend server performing computations. The proposed didactic material introduces a simplified development environment that aims to reduce technical difficulties and promote active and interactive lectures.

The following Sections describe more in depth the intervention contexts and the obtained results, by first providing an objective description of the collected data and then discussing the results based on our interpretation.

6.1 Results Description

As mentioned in Chapter 5, our research methodology follows a mixed-methods approach, employing both surveys and field notes. While we employ descriptive and inferential statistics to a certain extent, the data analysis process primarily relies on qualitative analysis. Therefore, the applicability of scientific replicability mainly pertains to the research process rather than the obtained results.

To maintain a coherent narrative, this Section provides a subset of the collected data, focusing on the most significant aspects.

6.1.1 General Information

The contexts we followed and analyzed come from three disciplines: Sociology, Psychology and Computer Science, with 95 students in total following the three courses we observed (Table 6.1).

Discipline	Course	Students
Sociology	Python Seminar	20
Psychology	Informatica ed Elementi di Programmazione I	54
Computer Science	Computer Programming 2	21
	Total	**95**

Table 6.1: General information about the intervention contexts

As depicted in Table 6.2, there are various reasons why students want to learn how to code. Among the 95 responses collected, only 2 individuals indicated that they were following a course just because they were required to do so. The others expressed interest in acquiring coding skills to develop projects, for future opportunities, or simply because they thought this field was exciting.

Why do you want to learn programming?	
Codes	**Occurrences**
Develop projects	29
Importance for Future & Jobs	26
Hedonism	23
Understand the field better	4
Required by the University	2

Table 6.2: Why students want to learn how to code - Codes occurrences

In our survey we inquired about the primary obstacle students encounter when learning to code. The findings, presented in Table 6.3, consistently indicate that "Learning and remembering the syntax"

is commonly regarded as the initial hurdle. Subsequently, students identified the need to adapt to a "New logical way of thinking" as a significant challenge.

What is the first obstacle in learning how to code?

Codes	Occurrences
Learning and remembering the syntax	39
New logical way of thinking	28
The learning process	12
The development process	10
The required knowledge about computers & maths	4
Field complexity	4
Languages heterogeneity	4
Use the IDE and solve technical issues	3

Table 6.3: What students think the first obstacle in learning to code is - Codes occurrences

Most of the respondents in our surveys are psychology students, hence a very small subset of the population - 21 computer scientists - already have a solid learning background on programming. However, table 6.4 demonstrates that a significant portion of students, specifically 62 out of 95, have already had exposure to programming languages. Additionally, 26 students already have familiarity with the programming language taught during the course.

Already known Programming Languages

Programming Language	Students (non-CS)	Students (CS)
Any	41	21
Java	14	8
Python	15	7
C or C++	13	21
Stata	13	0
Others	20	11

Table 6.4: Already known programming languages by students

Knowing a programming language does not imply high experience in programming. As indicated in Table 6.5, the self-assessment survey section revealed that participants generally rated their programming competence quite low. Non computer scientists answered on a scale of 1 to 5 with an average of 1.9±0.2, with a mode of only 1. Computer scientists rated themselves with higher scores, but when we proposed them a more detailed question about their actual programming experience, the results were comparable, with an average of 2.4 and, again, a mode of 1. Even their familiarity with the course programming language received a rating of only 2.0±0.2, which increased to 2.9 for students who already had prior knowledge of the course's programming language. Computing competence was rated slightly higher, with an average score of 3.4±0.2.

The students' motivation is generally very high, achieving a score of 3.9±0.2, that increases to 4.4 with mode and median of 5 when taking into consideration only students already knowing the course programming language. This level of motivation is consistent across both computer scientists and non-computer scientists. However, The field notes present a different perspective: while students following Processing and Python lectures were really found to be highly motivated, computer scientists in Java lectures usually seemed unmotivated, especially during theory explanations. Following is an extract of the field notes:

> Unfortunately, it doesn't seem to me that students are paying much attention to the lesson today. It often happens that they remain on already explained slides, getting distracted by their smartphones or other things.
>
> `05/05 - Computer Programming 2 - 09:30`

Section 6.2 will try to address this discrepancy.

Self-Assessment			
Question	**Average**	**Median**	**Mode**
Motivation	3.9 ± 0.2	4	4
Computing Competence (non-CS)	3.3 ± 0.2	3	4
Computing Competence (CS)	3.8	4	3
Programming Competence (non-CS)	1.9 ± 0.2	2	1
Programming Competence (CS)	2.4	2	1
Familiarity with the course's programming language	2.0 ± 0.2	2	1
		Answers	**95**

Table 6.5: Students' self assessment

Using ICPs was completely optional, and students were free to choose between using the proposed tool or the original didactic material offered by the instructor. However, based on our field observations, we can state that in every intervention the amount of students employing the proposed tool was the vast majority, reaching approximately 90-95%. In fact, on a scale of 1 to 5 students rated their ICPs use as 4.4±0.2, with a median and mode of 5. Compared to traditional, passive, theory lectures, the adoption of ICPs-based lectures appeared to have a positive impact on student motivation. Including exercises between the explanations made the lectures more interactive, and students enjoyed being challenged. Even the professor teaching Computer Programming 2 observed this behaviour, and a few small talks we had were centered around the integration of a more interactive approach that could exploit ICPs better and keep students focused.

> I briefly discussed today's lesson with the professor. He told me that these lessons are purely theoretical, and his teaching method currently does not fully utilize the potential of ICPs because the material has simply been translated 1:1. He mentioned [...] that probably by adopting a more interactive approach, the students would maintain their concentration for longer. In fact, he confirmed that they are typically more interested in the hands-on laboratories.
>
> `28/04 - Computer Programming 2 - 10:40`

In some cases our slides were not available, and we managed to observe a typical lecture with didactic material in PDF. During these observations, it was noted that some students appeared to lose interest in independently executing the code and verifying the results. Instead, when ICPs were used, a significant majority of students actively engaged in executing the code and checking the results. It is important to acknowledge that for this particular piece of information, we lack data triangulation and additional supporting evidence. Therefore, we present this observation as the only evidence we have without further discussion or analysis.

> Almost everyone has the PDF slides open, but for now, I see few people copying the code onto their IDE to test it. I have to admit that this never happened with ICPs. When the proposed tool was used, as soon as the professor started explaining theoretical concepts, the students would usually go to the first slide containing an example to try running it.
>
> `21/03 - Informatica ed elementi di programmazione I - 11:00`

6.1.2 Data Analysis

One of the main challenges identified in literature (Chapter 3) regarding programming classes is the difficulty in setting up a development environment. As presented in Table 6.6, before our intervention 16 out of 95 students encountered difficulties installing the course's prerequisites. Responses to open-ended questions further revealed that versions incompatibilities were a major issue. Specifically, 10 out of 54 students encountered issues installing Processing (18.5%), 4 out of 20 installing the Jupyter

Notebook (20%) and 2 out of 21 the Java IDE (9.5%). Clearly, computer scientists following the Java course were already experienced in this kind of process.

Troubles Installing Prerequisites

Course	Students who had troubles	Percentage
Informatica ed Elementi di Programmazione I	10	18.5 %
Python Seminar	4	20.0 %
Computer Programming 2	2	9.5 %
Total	16	16.8 %

Table 6.6: Number of students who had troubles installing the course prerequisites

Some issues, however, were also encountered afterwards, during the laboratories and lectures. When students were asked to rate the number of problems they encountered while using the course IDE on a scale of 1 to 5, the average response was 2.1±0.2, with a median and mode of 2 (Table 6.7). Field notes provide some examples of these problems. We propose here an extract that shows the limitations deriving from using a non-standardized development environment:

> The girl who had sought assistance from the professor regarding an issue with the Processing program has reached out to me. It seems that the professor was unable to help her due to the disparity between the Mac version of the Processing program and the Windows version used by the professor.
>
> `07/03 - Informatica ed elementi di programmazione I - 09:10`

In comparison, the results for ICPs indicate a similar average of 1.9±0.2 for encountered issues, but with a mode of 1. However, upon closer examination and comparison of the students' responses row by row, it became clear that the issues encountered with ICPs were generally fewer compared to the ones encountered with the course IDE (Table 6.8).

Encountered Issues	Average	Median	Mode
Course IDE	2.1 ± 0.2	2	2
ICPs	1.9 ± 0.2	2	1
Differences Among Envs	2.4 ± 0.3	2	2
		Answers	**65**

Table 6.7: Issues found by students during the intervention

Issues Comparison	Occurrences	Percentage
IDE issues >ICPs issues	26	40%
IDE issues <ICPs issues	14	22%
IDE issues = ICPs issues	25	38%
	Answers	**65**

Table 6.8: Comparison of issues found in the course's IDE and ICPs

What were the problems found in ICPs? Before starting our intervention we were already aware of problems regarding the compatibility of some programming languages. Especially when using Processing, the outputs may sometimes be wrong and different with respect to a real IDE, as shown in the following quote from the field notes:

> A student brings to my attention that an exercise presented by the professor does not work properly on our slides system. The exercise involves implementing a while loop to introduce a 5-second delay before executing a certain piece of code. However, when attempted in the playground, the solution is mistakenly identified as an infinite loop, resulting in the execution being halted.
>
> `14/03 - Informatica ed elementi di programmazione I - 09:56`

Task	Students Understanding it	Percentage
Execute code	87	88%
Find the output	82	83%
Copy Button	79	80%
Edit the code	76	77%
Enable full-screen	73	74%
Open Settings	68	69%
Reset Button	65	66%
Change theme	60	61%
Code Scaffolding	21	21%
Tabs Handling Button	11	11%

Table 6.9: Number of students understanding how to perform some tasks given some images of ICPs

To assess the negative impact of differences between the results of ICPs and the IDE on the students' experience, we included a question in the final survey. The average response indicates a relatively high negative impact, with a score of 2.4±0.3. We can deduce one of the main issues students found derived from this specific problem. This observation is further supported by both the field notes and the answers to the open-ended question regarding encountered bugs: out of 11 answers, 9 of them were related to this incompatibility complication.

In the surveys we presented students with some images of ICPs, asking them about their understanding of performing tasks and the functionality of various buttons. The results, as presented in Table 6.9, indicate that students generally grasped how to use the main features of ICPs at first glance. Specifically, 96% of students were able to understand how to perform at least one task. However, a smaller percentage of students demonstrated comprehension of certain more advanced features. Only 21% understood the code scaffolding feature, and 11% comprehended the tabs handling button.

The comparison between students' perceptions of the difficulty of various aspects of learning to code before and after the intervention is shown in Figure 6.1. The results indicate a clear reduction in the perceived difficulty across most aspects, with the exception of "Fall Behind". When comparing computer scientists and non-computer scientists, there are minimal differences among the results. Therefore, further analysis regarding these differences may not be necessary, as the overall trend of reduced perceived difficulty was consistent among both groups.

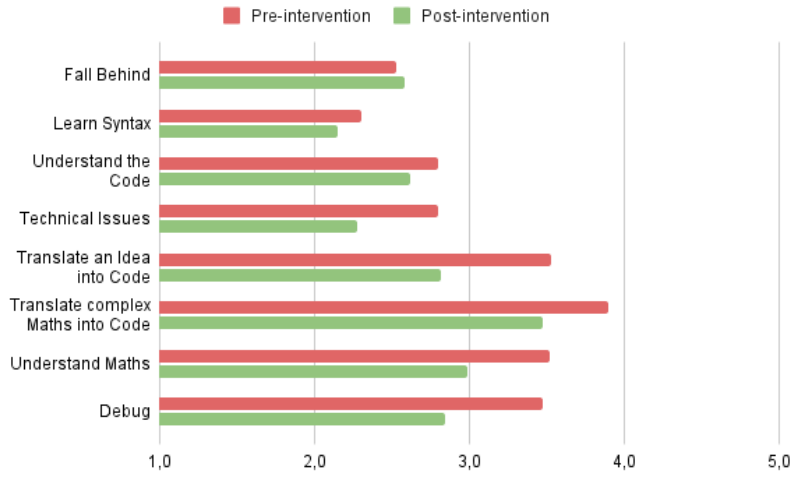


Figure 6.1: Comparison of the perceived difficulty in learning to code, before and after the intervention

In the proposed survey students expressed their preferences regarding the features they would like to have in a slides system, exposed in Table 6.11. The most important features identified by students were compatibility with any environment, the ability to copy and paste code, and the option to search for keywords. Additionally, the availability of offline access and a PDF version of the slides were also

Most useful ICPs feature		
Feature	**Occurrences**	**Why**
Stay in Browser & on-the-fly changes	30	You don't need any other software or IDE It's convenient and it saves time It helps maintaining focus Doesn't require space It's easier to follow the lecture ICPs automatically save my changes It allows you to execute code in other devices On-the-fly changes help understanding the code better
Move freely	17	Check previous or next exercises I can focus more on unclear concepts
Exercises drawing attention to the right concepts	9	They allowed me to challenge myself They allowed me to understand which were my knowledge gaps I find it a good way to learn how to program
Compare code	6	It's easier when using a standardized environment
Others	3	-

Table 6.10: Table showing what students reported to be the most useful feature of ICPs

highly rated. Features related to physical paper and handwritten notes, instead, were considered less significant.

In the final survey conducted at the end of experiment, students were asked instead to evaluate the helpfulness of the features offered by ICPs. The feature that received the highest appreciation was the ability to stay in the browser and perform on-the-fly changes to code snippets. Students found this feature to be convenient, time-saving, and helpful in maintaining their focus (Table 6.10).

However, as we can see from Table 6.12, nearly all ICPs' main features were rated helpful, except for the change theme button, which was however typically essential for the instructor to ensure all students could clearly see the code.

Feature	**Average**	**Median**	**Mode**
Slides Compatibility with any Device	4.1 ± 0.2	4	4
Offline Availability	3.7 ± 0.2	4	4
Copy / Paste code snippets	3.9 ± 0.2	4	4
Search for keywords	3.8 ± 0.2	4	4
Standardized environment	3.4 ± 0.2	3	4
Print on Paper	2.3 ± 0.2	2	2
Take handwritten Notes	2.6 ± 0.2	3	3
PDF version	3.7 ± 0.2	4	4
		Answers	**95**

Table 6.11: Importance of some features a slides system could have

Feature	**Average**	**Median**	**Mode**
Stay in Browser	4.0 ± 0.2	4	4
Exercises drawing attention on the right concepts	3.8 ± 0.2	4	4
Move freely through the slides	4.2 ± 0.2	4	4
Perform on-the-fly changes to code snippets	4.1 ± 0.2	4	4
Easily compare code with that of other students	3.8 ± 0.2	4	4
Button to download a code snippet automatically	3.8 ± 0.2	4	4
Button to copy a code snippet automatically	3.7 ± 0.3	4	3
Button to change the theme of the code editor	3.2 ± 0.3	3	2
		Answers	**65**

Table 6.12: Evaluation of the features provided by ICPs

The slides have also been evaluated in terms of more general aspects. Even in this case, Table 6.13 shows that students generally appreciated every aspect of ICPs, especially their convenience of

use (4.4±0.2) and performance (4.3±0.2).

Slides Quality evaluation			
Aspect	**Average**	**Median**	**Mode**
Graphics Quality	4.4 ± 0.2	4	4
Contained Information	4.3 ± 0.2	4	4
Convenience of Use	4.4 ± 0.2	4	5
Exercises Effectiveness	4.2 ± 0.2	4	4
Performance	4.3 ± 0.2	4	5
		Answers	**65**

Table 6.13: ICPs slides quality, evaluated by students on different aspects

As a result, on average ICPs was the most widely used study tool (3.9±0.3), together with other interactive environments such as SoftPython and the Jupyter Notebook. However, PDF slides were still used a lot by students, as the average reaches 3.6±0.3. Notes taken during the lectures, instead, were less commonly used, especially the ones taken on paper (Table 6.14).

Used Study Tool	**Average**	**Median**	**Mode**	**Answers**
ICPs	3.9 ± 0.3	4	5	65
SoftPython	3.8	4	5	12
Jupyter Notebook	3.8	4	4	12
PDF Slides	3.6 ± 0.3	4	4	53
Instructor's Exercise Files	3.2 ± 0.3	3	3	43
Notes taken digitally	2.9 ± 0.3	3	2	65
Notes taken on paper	2.0 ± 0.3	1	1	65

Table 6.14: Evaluation of how much students used each study tool available during the course

The preferences of students regarding the type of learning material were assessed, and the results indicate a strong preference for interactive material. On a scale of 1 to 5, where 1 represents static material and 5 represents interactive material, the average rating was 4.4±0.2.

Evaluating how ICPs perform in terms of accessibility is complex. Only a couple of students who tried the tool had visual impairments, and we did not include any question about it in the surveys. Hence, the only evidence we have regarding how accessible the tool is are the field notes.

One student had the screen contrast set to the maximum value, and was not using any accessibility tool. With this contrast slides were completely usable, and the code snippets' text was visible. However, it was noted that the arrows used for slides navigation disappeared, which could potentially pose a challenge for some users. Another student used a zooming tool to magnify the page. While the tool generally worked well in our environment, there was a minor inconvenience related to the auto-completion popup, which caused the zoom window to move automatically. Following is an extract of the field notes:

> A student with visual impairments increased the contrast to the maximum in order to see the slides better. The only problem resulting from this is that the arrows disappeared. Fortunately, it is possible to navigate the slides using the arrows, so I didn't have to assist the student in any way. The playground is also clear enough with this contrast, and all the buttons appear white on a black background, making them fully visible.
>
> `28/03 - Python Seminar - 14:54`

As a last note, we want to take into consideration the features students missed from ICPs. In the Java class, some students expressed the need to take notes directly on the provided material, which was possible with the PDF slides but not with ICPs. This feature allows students to add comments directly on the text, enhancing their note-taking experience. Both the answers to the open-ended question about missing technical features and the field notes prove the importance of this feature:

> Two other students are using the original PDF slides to take notes directly on the text, by using tools like Adobe Acrobat or Google PDF Reader.
>
> `28/04 - Informatica ed elementi di programmazione I - 09:30`

In Python and Processing classes, instead, students often had issues copying the solutions presented by the professor during the lecture. The ability to automatically copy the instructor's code from a slide was identified as a highly useful feature that would significantly improve the learning experience for students.

> As the professor presents the solution to an exercise, I notice a girl capturing a photo of the slide, while others try to quickly copy the code snippet. This situation highlights another instance where an appropriate tool [*to automatically copy code*] could revolutionize the students' experience.
>
> `28/03 - Python Seminar - 16:15`

All the other features students recommended to include into our system are summarized in Table 6.15.

Missing Features	
Codes	**Occurrences**
Write page number to navigate slides	3
Show the errors in the editor	2
The theme change button should also affect the slides	1
Put a button to download the PDF automatically	1
Add the possibility to take notes on the slides	1
Enable by default the tabs handling	1
Add automatic indentation	1

Table 6.15: List of additional features asked by students

6.2 Results Discussion

In this section, we will provide an overview of the outcomes and key discoveries from the research that was discussed. The results will be categorized into thematic areas, aligning with the structure of the questionnaires utilized in the study.

Students' Programming Knowledge

When considering non-computer scientists, 41 students out of 74 already knew at least one programming language. The most widely used were the ones typically employed for data analysis (Stata and Python), or as a typical starting point for learning programming (Java, C/C++). Interestingly, 18 of these students were already familiar with the programming language taught in the course. However, the general self-assessment for programming knowledge shows that despite the number of known languages, students still felt like beginners in this field. The same goes for computational competence, where on average students felt they had a medium level of knowledge.

If we consider computer scientists, all these students already knew at least one programming language, as C/C++ was taught during the first semester. Among these students, 8 of them already knew the programming language taught during the course. In general these students felt they had more competence in computers and programming with respect to non computer scientists. However, when delving deeper into their coding experience through an additional survey question, the results were comparable to those of non-computer science students. This suggests that despite their foundational programming knowledge, computer science students still perceived themselves as beginners in the broader field of programming.

Field notes strongly support these observations, especially when considering non computer scientists, where even seemingly simple exercises, such as swapping two variables, proved challenging and required considerable time and effort.

As a result, all the students on average appear to have limited knowledge in programming, so the proposed population meets the context we wanted to observe and study. At the same time, this population gives us the possibility to test our tool with both students interacting with development environments for the first time, and students having already some experience in these products.

Motivation

The field notes generally highlight the students' high level of motivation, especially when challenged with some exercises. This observation is further supported by the survey results, where the majority of students (82 out of 84) expressed genuine interest in learning programming and were not merely fulfilling the study course requirements. Students cited various motivations for learning programming, including the desire to develop projects, the anticipation of future opportunities, or simply hedonistic reasons. Additionally, by asking students to self-assess their motivation we received very positive results. The students' familiarity with multiple programming languages also underscores their personal interest in the field.

However, it is important to note that the field notes occasionally documented instances where students experienced a decline in motivation or difficulty maintaining focus, particularly during theory-based lectures that lacked interactivity. These observations, which seem to contradict the positive self-assessments provided by students, suggest that there may be a missing piece of the puzzle regarding the factors influencing students' motivation.

Active Lectures

All the settings in which a focus drop was noted during field observations had one thing in common: they were theory-based lectures were the professor was mainly explaining concepts and presenting code snippets, without engaging students in active learning tasks. On the other hand, the importance of interactivity was consistently emphasized by students in surveys. When asked to rate their preference for static (1) versus interactive (5) material on a scale of 1 to 5, most students expressed a preference for interactive material, with an average score of 4.4 ± 0.2, a median of 5 and mode of 5. Additionally, the standard deviation of 0.8 indicates that approximately 99% of students preferred interactive material.

The ability to stay in the browser and perform on-the-fly changes to code snippets are the main ingredients for an interactive lecture, and 30 out of 65 students defined these as the most useful features ICPs offer. They found it convenient, time-saving, and helpful for maintaining focus and understanding the code better. Furthermore, 9 students highlighted the presence of exercises drawing attention to the right concepts as the most useful feature of ICPs, as they directed their attention to the relevant concepts while helping them identifying their knowledge gaps.

We can deduce having an easy-to-access development environment not only saves time and reduces technical difficulties, but it also helps student understanding the explained concepts and exercises better. Additionally, by allowing more active lectures it helps maintaining attention and motivation throughout the learning process.

Programming Languages Compatibility

The integration of the Processing programming language into ICPs encountered several challenges, resulting in students finding incorrect results or discrepancies between the outputs of ICPs and the Processing IDE. The Python integration proved to be more robust. However, students still encountered some issues even when using this programming language.

Field notes show that these differences often led to confusion among students. For instance, in our Processing integration, initializing some variables in the global scope was sometimes not possible, whereas it is allowed in the actual Processing environment. Consequently, students would approach me for assistance, mistakenly believing that the issue was with their code rather than a limitation of our tool. This would leave them feeling somewhat disoriented. Similarly, In Python classes sometimes the interpreter mode would not work, leading students to question their understanding of the language.

The findings from the surveys align with these observations. In the open-ended questions nine students complained about the compatibility of the provided programming language, asking to improve

it. Additionally, students reported that these differences between the execution results generally negatively affected their experience.

Browsers limitations

One limitation of ICPs is their compatibility with browsers other than Google Chrome or Firefox. Students using Safari experienced minor issues, while a student using Brave could not even execute the code snippet provided on the slides. Fortunately, installing a new browser is a simple process that takes a few minutes. After having asked students to install Google Chrome, ICPs seemed to work perfectly. In fact, in the surveys no students complained about this.

However, relying on a specific browser for optimal functionality poses a limitation to the usability of the tool. While instructing students to install a different browser can mitigate this issue to some extent, it is still a technical requirement that adds an additional step for students. Therefore, ensuring a broader compatibility across different browsers would be desirable to minimize technical issues.

Technical Issues

Throughout the research, various technical issues were encountered by students. Some of them highlighted the importance of a standardized and user-friendly programming environment. Field notes document a particular instance where a student using the Mac version of the Processing IDE faced difficulties, and the professor was not able to help her due to the substantial differences between the Mac and Windows versions. This is a typical example of the challenges that students face when using different versions of an IDE.

In the surveys, 16 students out of 95 students reported issues installing the course requirements, and among the problems identified and described by students in the open-ended questions, we also find issues deriving from the absence of a standardized environment.

The course IDE in general led to issues too, as on a scale of 1 to 5, on average students asserted a number of issues equal to 2.1 ± 0.2. In terms of average, the amount of issues found in ICPs is comparable, even if the mode in case of ICPs is only 1. However, since most of the students (26 out of 65) found more issues in the course IDE compared to ICPs, we can conclude the proposed tool has reduced technical difficulties. In fact, a comparison between the first and final surveys reveals a decrease in the reported technical issues. Additionally, our analysis found a statistically significant positive correlation between the issues encountered with ICPs and how much differences in execution results caused by ICPs negatively affected the students' experience, with a correlation coefficient of 0.15 ($p = 0.0004$). This indicates that students who were negatively affected by ICPs' wrong results were more likely to report issues in ICPs. Although the correlation coefficient is small, it suggests a positive linear relationship between the two variables. Moreover, the field notes clearly show that a number of times students encountered difficulties deriving from the wrong outputs provided by ICPs. Therefore, addressing this particular kind of issues - which would simply require more development time - could lead to an improvement of students' overall experience with our development environment, and a reduction in technical problems.

To summarize, although some technical issues were encountered with ICPs, students reported fewer problems compared to the course IDE. The data from field notes and surveys suggest that these issues could mainly be related to the differences in execution results, which could be resolved through further development efforts. Hence, with appropriate measures we think ICPs have the potential to further mitigate technical issues.

ICPs UI evaluation

The majority of students, approximately 80%, understood through an image of ICPs how to perform the most important tasks, namely editing code, executing it and finding the output result. Approximately 65% of students understood how to perform more advanced tasks, such as opening settings, enabling full-screen and changing the theme. As a result, only 4% of students had no idea how to use ICPs.

The copy and reset buttons were generally well understood by students, with percentages of approximately 80% and 60%, respectively. However, two elements of the UI proved less clear to students: the tabs handling button and scaffolded code snippets. Field notes taken by observing students interacting with an actual instance of ICPs actually substantiate the survey's data: most students quickly learned how to use the basic functionalities of the tool and buttons to change theme, copy or reset code. However, we have seen only a couple of them employing the tabs handling feature, and one of them advised us to enable it by default. Unfortunately, the evaluation of the scaffolding feature was limited since the course slides did not contain such playgrounds.

After the intervention we asked students which they thought was the most familiar environments between ICPs and the IDE proposed by the course. Out of 65 students, 47 (apprixmately 72%) considered ICPs more familiar. This finding suggests that students generally had no difficulies in understanding the user interface of ICPs. However, it is crucial to address certain areas for improvement, such as providing a clearer way to handle tabs while ensuring accessibility and designing a more intuitive interface for scaffolded exercises.

ICPs errors handling

Learning and remembering the syntax has been designated by most students (39 out of 95) as the first obstacle in learning to code, and 7 students expected following the syntax as the task taking more time. Especially in Processing lectures, field notes support this observation: students frequently encountered errors due to an inappropriate usage of keywords or brackets.

In the context of error handling, ICPs have some limitations. Currently, errors are not shown directly in the editor, requiring stsudents to execute the code to identify any syntax errors. IDEs, instead, offer more support for detecting and highlighting syntax mistakes. As a result, two students talked about this problem in the survey, inserting it in the "Missing technical features" section.

Since learning and following the syntax is expected and seems to be one of the first obstacles in learning to code, ICPs should provide errors feedback directly in the editor, to help students identify and rectify their mistakes efficiently.

Maths requirements

Among the surveyed students, 24 of them thought solving maths-related issues would have been the task taking more time, primarily due to their limited familiarity with the subject. In some cases, field notes further substantiated this hypothesis, particularly in Processing lectures where students encountered difficulties in solving exercises that involved implementing the movement of a ball in a 2D space. Even after observing the solution, students struggled to grasp the calculations related to the ball's velocity.

Apart from these specific scenarios, maths knowledge was not really needed. In fact, at the end of the intervention the percentage of students thinking solving maths-related issues would have been the task taking more time was a bit lower (14% against 21%). However, it is prudent to anticipate potential difficulties that may arise when engaging more complex programs that rely on basic mathematical principles. Maybe, providing additional support or resources to students having no maths background could facilitate their progress in programming.

Move Freely

The possibility to freely move and navigate at students' own pace has been noted to be highly valuable during field observations. Numerous times students were able to solve exercises more quickly than the professor, or needed to spend more time focusing on previous exercises. Particularly during Processing lectures, students tended to navigate to the next code snippet as soon as the professor started explaining a new concept. This behaviour suggests they wanted to observe the actual outputs of the code snippets and make adjustments while receiving explanations from the professor.

The surveys confirm the importance of this feature: 17 out of 65 students identified the ability to move freely as the most useful feature provided by ICPs. As described by a student in the survey,

move freely does not only mean navigating the slides at will. It also means moving from one exercise to the other very easily, since they are already differentiated and clustered by subject:

> The fact of having all the exercises already "open" in a single file is very convenient and saves time.

This recurrent theme marks the advantages of consolidating all necessary resources within a single environment.

Students' experience in learning to code

After the intervention, students reported encountering fewer difficulties than expected across various aspects of learning to code, especially in translating an idea into code and debugging. The average ratings for these tasks decreased from 3.5±0.2 to 2.8±0.2.

Interestingly, while "following without falling behind/ahead" still perceived as challenging, no student out of 65 actually felt it was the task taking more time. However, field notes did document instances where students struggled and spent a lot of time on some exercises, suggesting that maintaining pace throughout a lecture remains a concern.

However, both survey responses and field notes highlighted the challenge of following the lecture without falling behind, particularly during interactive lectures. Determining the appropriate pacing for students can be challenging, given the heterogeneous knowledge and reasoning capabilities observed in class. Further research is needed to analyze this issue more deeply and explore strategies to address it effectively.

Most important features of a slides system

According to students' feedback, the most important features students think a slides system should have are its compatibility with any device and the ability to copy/paste code and search for keywords (approximately 4.0±0.2). PDF slides generally fulfill these requirements, except for the possibility to easily copy code, which sometimes may be cumbersome or may require transcription. ICPs are compatible with any device, offer a better experience when copying code, but only provide the searching functionality when entering the vertical version, typically used to print on PDF. Hence, each kind of didactic material has its own pros and cons.

With a relatively smaller average (approximately 3.7±0.2), students think it is important to have slides available offline and the corresponding PDF version. Both features are available in ICPs, but further investigation is recommended to gain a deeper understanding of students' continued preference for PDF slides.

The availability of a standardized environment was comparatively considered a bit less important (3.4±0.2). This particular feature is probably more valued by instructors for troubleshooting purposes than by students themselves.

ICPs Use

The usage of ICPs during the lectures was voluntary, and students had the option to choose between the original didactic material and our alternative. However, the majority of students opted to use ICPs throughout the lectures. Additionally, the survey results indicate that ICPs were also the most widely used study tool among students, achieving, on a scale of 1 to 5, a mode of 5 in terms of utilization.

These results suggest a high level of engagement with the tool, and the fact that it generally met students' needs effectively. Additionally, as shown by the surveys, these responses came from students who typically used ICPs whenever they were available. This makes their feedback particularly valuable and reliable.

ICPs features

The feedback from students indicates that most features offered by ICPs were considered important and valuable. However, one feature that stood out as less important to students was the ability to

change the theme. It was noted in the field notes that this feature might have been more useful for instructors to enhance the visibility of code snippets in different situations, rather than for students.

Among the features provided by ICPs, the ones that were selected as the most helpful by students were "on-the-fly changes", "move freely", and "stay in the browser". These features contribute to the interactive nature of the tool, allowing students to freely navigate through exercises and edit the code while staying in the same browser environment.

The fact that these features resonated with students suggests that one of our main objectives was successfully achieved: students appreciated the interactivity of the proposed learning environment.

Accessibility

While we acknowledge the importance of accessibility in educational tools, we must note that our evaluation of ICPs in terms of accessibility is limited. We did not have the possibility to gather extensive data for this aspect, and further research is needed, especially to evaluate ICPs' use through a screen reader.

However, based on the limited field observations we conducted, there are indications that ICPs provide at least some degree of accessibility for students with visual impairments, since zooming functionalities and contrast-based solutions work within the proposed environment with few issues.

Missing Features

Students often take notes directly on PDF slides during the lectures, by exploiting tools like Adobe Acrobat or Google's PDF Reader. Our slides system currently does not support this feature. While only a couple of students out of 95 explicitly mentioned this need, we believe that implementing the ability to take notes directly on the slides would be a valuable addition to our system, enhancing the user experience and providing a more comprehensive learning environment.

In Python and Processing classes, students encountered difficulties when trying to copy the solutions provided by the instructor. Although no students specifically mentioned this issue in the surveys, it is worth considering as it directly affects the use of the proposed tool. To address this, an option to automatically copy the instructor's code on a given slide could be introduced in ICPs. This feature would not only assist students in following the lecture without falling behind, but also provide them with readily available solutions.

Slides quality evaluation

The evaluation of slides quality conducted by students yielded consistently high scores across all aspects. On a scale of 1 to 5, the average was 4.3 ± 0.2. The convenience of use and performance were particularly appreciated, with a mode rating of 5 out of 5.

During the pilot study a typical issue encountered with ICPs ICPs, as identified mainly through qualitative analysis, was the performance or execution speed of the system. Results indicate that the recent improvements made to address this concern had a positive impact.

Study tools

Among the most widely used study tools we find ICPs, SoftPython and the Jupyter Notebook. PDF slides were also used, but less frequently. Note-taking was generally less prevalent among students.

Even as study tools interactive systems seem to be the typical choice for students. In fact, as we have already seen, approximately 99% of students expressed a preference for interactive learning materials. Compared to the Jupyter Notebook system, ICPs were generally employed more, but we don't have enough data to further argument this fact.

7 Limitations & Future Work

In this Chapter we discuss the limitations of ICPs we encountered in our study, by describing potential areas for future research and improvements. While the proposed tool has proved to to be a solid first step towards enhancing programming classes with browser-based solutions, it is essential to identify the constraints and challenges that emerged during our study. By identifying these limitations, we can get a better understanding of the areas needing further development, and define more clearly the future of ICPs.

The students' feedback we received and the field notes we collected throughout our study shed a light on the strenghts and weaknesses of the proposed tool, allowing us to evaluate it. The contexts we analyzed are clearly not enough for a thorough and accurate study, but by identifying the encountered limitations, we can at least outline the major and most evident issues, and provide a roadmap for future research.

7.1 Strengths and Shortcomings

ICPs introduced a high level of interactivity, allowing students to actively engage with the presented code snippets and performing on-the-fly changes. The interactive nature of ICPs lead to an alternative more hands-on learning experience based on fast tinkering, which seemed to facilitate a deeper understanding of programming concepts. Students enjoyed being challenged with exercises throughout the lectures, and they consistently expressed a preference for interactive material. As a result, from a pedagogical standpoint the proposed tool eased the transition to a more active teaching methodology, which was generally much appreciated by students.

ICPs offered the convenience of an all-in-one environment that eliminates the need to use multiple resources and softwares during programming lectures and laboratories. The ability to access slides, a code editor with execution capabilities and exercises in a single environment contributed to streamline the learning process. Firstly, it simplified the instructor's workflow by allowing them to present and share content without the need to switch between different platforms; secondly, it allowed students to quickly test code snippets independently. Additionally, since the proposed tool is entirely browser-based, students had no installation processes to follow, and they encountered fewer issues compared to using the IDE required for the course.

One of our main objectives was to significantly reduce the number of issues encountered by students when interacting with a development environment. While students did experience fewer problems with ICPs, the improvement was not substantial. In fact, they often found inconsistent and wrong outputs generated by the browser engine we developed for compiling and executing programming languages. More development effort should be invested to address this issue, especially because even state-of-the-art solutions are still limited.

Although students recognized ICPs' features fostering interactivity, there were certain aspects of our slides system that they felt were missing. One area for improvement is error handling, where automatically displaying syntax errors within the editor would greatly assist students in troubleshooting code snippets. The inclusion of a button to remotely copy the solutions provided by the professor would drastically enhance the students' experience during laboratories, as this was one of the main reasons why some students used to fall behind. Finally, some students expressed a desire for additional features commonly found in PDF slides, such as the ability to take notes or quickly locate a specific slide by entering its number.

7.2 ICPs & Research Limitations

Our study primarily focused on a limited number of Higher Education contexts. ICPs have shown potential to be a valuable tool for education, but additional research is essential to evaluate this tool

on other aspects that are not completely explored in this research. In this Section, we discuss these particular issues and limitations of ICPs and this dissertation work, while also highlighting possible areas requiring further investigation.

Testing in Critical Contexts

It is important to acknowledge that our evaluation of ICPs did not include critical contexts. The proposed tool should be tested in scenarios with restricted or limited internet access to the internet, outdated hardware or other challenging environments that could potentially impact the usability of ICPs. While our study focused on more controlled and accessible environments, it is essential to conduct additional field testing to assess the effectiveness of the proposed tool in a broader range of settings.

Offline Slides Size

The size of the offline slides is another important aspect to consider, especially for students with limited internet access. Although the Javascript bundle required to set up the slides and development environment has an average size of only 3MB, the offline version, which was not tested during this study, currently has a significantly larger size of approximately 40MB.

The substantial difference in size between the online and offline version is primarily due to the offline version including environments for all supported programming languages. This can clearly pose a challenge for students with poor internet connectivity. Hence, it is crucial to invest further development effort in modularizing the offline versions. This would involve optimizing the bundle by only including the essential dependencies for the utilized programming language.

Longevity & Dependencies

The sustainability and longevity of ICPs are closely tied to the number of dependencies and the durability of the underlying technologies. We have successfully tested the proposed tool on older PCs, as demonstrated in Table 7.1. However, the compatibility of ICPs with future systems cannot be guaranteed, as technological advancements and evolving web standards may render certain technologies obsolete. Backward compatibility of newly introduced protocols and technologies will be essential to ensure the longevity of ICPs.

Programming Language	**Initialization (s)**	**Execution of a code snippet (s)**
Java	59	15.5
Python	59	immediate
C/C++	209	0.5 to 8, depending on *includes*
Processing	0	immediate
P5	0	immediate
Standard ML	0	4.6
Javascript	0	immediate
Typescript	0	0.6
SQL	0	0.5

Table 7.1: Performance on a PC with 4GB DDR3 RAM and a dual-core AMD E-300 1.3GHz CPU

Accessibility Evaluation

The evaluation of accessibility features, particularly the compatibility with screen readers, was a significant limitation of our study. Accessibility is a critical aspect of any educational tool, and the ability of ICPs to accommodate users with visual impairments should be thoroughly evaluated. Future research should prioritize comprehensive accessibility testing, gathering feedback from users with diverse accessibility requirements, and implementing necessary enhancements to ensure a more inclusive learning experience. Furthermore, it is essential to stay informed about the Web Content Accessibility Guidelines (WCAG), ensuring ICPs align with the best practices.

Programming languages compilation/interpretation

The compilation or interpretation of certain programming languages through WASM is still not production-ready due to limitations of state-of-the-art technologies. This can restrict the ability to fully support and execute code in certain programming languages within ICPs, and this research clearly shows how compatibility complications and wrong outputs can potentially confuse beginner students.

To address this challenge, we need to regularly update ICPs exploiting the ongoing advancements of WASM technologies. By collaborating with the open-source community, ICPs could leverage emerging new solutions to improve the reliability of in-browser code compilation and execution.

Slides Editor limitations

Despite having developed a slides editor, all slides in our study were manually created due to limitations of the editor itself. The slides editor allows users to create simple and uncomplicated user interfaces, which sometimes may not be enough for an actual lecture. For instance, the editor lacks animation capabilities, such as text appearing gradually, which can be valuable for enhancing explanations.

To address this limitation, further improvements should be made to the editor's functionalities. This could involve expanding the range of available slide layouts, providing more options to customize text formatting or elements position, or incorporating animation features. Simplifying the creation process would allow instructors to easily design didactic material without requiring HTML knowledge.

7.3 Thoughts on Next Steps

The implementation and evaluation of ICPs have provided valuable insights into its potential and limitations. We can build upon these findings, creating a roadmap for future development and research that takes into consideration the enhancement of limited components and improvement of its effectiveness in challenging settings.

First of all, development effort should be invested to fix the compatibility problems of some programming languages. Most of the issues students found when using ICPs were related to wrong or incoherent outputs given by our development environment, hence solving this particular issue may drastically reduce the difficulties encountered by students.

The offline version of our slides system has not been evaluated during our research, but we realize it has some pitfalls. While the idea of using a single-file distributable web server seems the best choice to execute web-based applications locally, it is essential to reduce the size of the resulting file. In detail, it should only contain the dependencies needed to compile and execute the programming language used in the slides, instead of the entire ICPs engine.

The slides editor component has some limitations hindering its full potential. It should be refined, integrating more slide templates, animations and drag-and-drop functionalities. Expanding the editor's ability to create different user interfaces would greatly impact its effectiveness, allowing instructors to create didactic material without needing to modify directly the underlying HTML code.

Finally, conducting in-depth pedagogical research is essential to evaluate ICPs in different settings. Exploring aspects such as learning outcomes, student engagement and the impact of more interactive lectures can contribute to University programming education. Additionally, verifying whether ICPs can be used in challenging contexts may become a first step towards the creation of more inclusive development environments.

By focusing on these steps, we can advance the development and research of ICPs, making this tool for programming education more robust and effective. In particular, the continuous refinement and evaluation of ICPs will contribute to the creation of interactive and inclusive learning environments, ultimately enhancing the experiences of both students and instructors.

8 Conclusion

In this dissertation, we have presented Interactive Code Playgrounds (ICPs), a web-based alternative slides system designed to be inclusive and to simplify programming education by providing an all-in-one didactic material that allows to follow the lectures, test code snippets and solve exercises.

Chapter 3 provided a comprehensive review of the related work in the field of programming education, together with a presentation of some important challenges in digital transition. We discussed existing tools and platforms developed to enhance and simplify programming classes, with their strengths and shortcomings. ICPs try to address technical, socio-technical and pedagogical difficulties arising from the current state-of-the-art solutions. In detail, the project is proposed as a way to simplify the installation process of the development environment for introductory programming courses, foster active pedagogy, and reduce technical difficulties, while taking into consideration social aspects related to digital divide and accessibility, which are too often unconsidered for computer science-related courses.

From the evaluation of ICPs conducted in three introductory courses on programming, we have gained valuable insights. In this Chapter, we summarize the key findings and contributions of our research, discuss their implications, and outline potential directions for future work.

The findings revealed both strengths and limitations of the proposed slides system. Among the main issues encountered by students, we find compatibility problems of the integrated programming languages and limitations deriving from browsers other than Google Chrome or Firefox. Additionally, there were some features of our slides system students felt were missing.

On the other hand, students appreciated the convenience of an all-in-one environment, the ability to test code independently, and the interactive features provided by ICPs. Specifically, students reported the resulting pedagogy more engaging and productive, improving their comprehension of programming concepts and exercises. The slides quality and the code playgrounds' user interface were generally positively evaluated by students, and most of them considered the proposed tool more familiar than the course IDE. Furthermore, although some technical issues were encountered with ICPs, students reported fewer problems compared to the course IDE.

The research does not represent a complete and comprehensive evaluation of ICPs, and further investigation is needed. We did not analyze challenging contexts lacking a strong internet connection or employing outdated and obsolete hardware. We also did not have the possibility to comprehensively evaluate the tool's accessibility, and its effectiveness in the offline version, which we already know may have some technical issues. As a result, for future research and development we think it is essential to fix compatibility issues with some programming languages, improve the offline version, and conduct a more sophisticated pedagogical research, evaluating ICPs in different settings.

Overall, however, the evaluation of ICPs has shown the tool's potential for programming education. The integrated environment provided by ICPs offers convenience, interactivity, and the ability to facilitate hands-on coding experiences. While there are limitations and areas for improvement, the positive feedback from students and the field notes indicate that ICPs can have a significant impact on programming education.

In conclusion, our research has contributed to the field of interactive learning environments and programming education by developing and evaluating ICPs. The findings from our study provide valuable insights into the strengths, limitations, and potential improvements of ICPs. We think that with further development and research, ICPs can become a valuable tool for programming education. By leveraging technology to create interactive and inclusive didactic material, we can enhance the effectiveness and accessibility of programming education, ultimately helping students develop logical reasoning and coding skills.

Bibliography

[1] Applied webassembly - compiling and running c in your web browser. https://cppcon2019.sched.com/event/SiVN/applied-webassembly-compiling-and-running-c-in-your-web-browser. Last Accessed 07/04/2023.

[2] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. Interviews and observation of blind software developers at work to understand code navigation challenges. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 91–100, 2017.

[3] Christopher M Boroni, Frances W Goosey, Michael T Grinder, Jessica L Lambert, and Rockford J Ross. Tying it all together: Creating self-contained, animated, interactive, web-based resources for computer science education. *ACM SIGCSE Bulletin*, 31(1):7–11, 1999.

[4] Christopher M Boroni, Frances W Goosey, Michael T Grinder, and Rockford J Ross. A paradigm shift! the internet, the web, browsers, java and the future of computer science education. *ACM SIGCSE Bulletin*, 30(1):145–152, 1998.

[5] James Dean Brown et al. *Using surveys in language programs*. Cambridge university press, 2001.

[6] Jeffrey Buckley, Tomás Hyland, and Niall Seery. Examining the replicability of contemporary technology education research. *Techne Serien*, 2021.

[7] Anne Burns. *Collaborative action research for English language teachers*. Cambridge University Press, 1999.

[8] Sue Clegg, Alison Hudson, and John Steel. The emperor's new clothes: Globalisation and e-learning in higher education. *British journal of sociology of education*, 24(1):39–53, 2003.

[9] Pamela Fox. The benefits of html slides for programming lectures. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, pages 1055–1055, 2022.

[10] Jack R Fraenkel, Norman E Wallen, Helen H Hyun, et al. *How to design and evaluate research in education*, volume 7. McGraw-hill New York, 2012.

[11] Bill Gillham. *Developing a questionnaire*. A&C Black, 2008.

[12] Dion Hoe-Lian Goh and Peng Kin Ng. Link decay in leading information science journals. *Journal of the American Society for Information Science and Technology*, 58(1):15–24, 2007.

[13] Thomas F Griffin III and Zack Jourdan. Educational use cases for virtual machines. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 365–366, 2012.

[14] Philip Guo. Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 1235–1251, 2021.

[15] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584, 2013.

[16] Phil Hackett, Michel Wermelinger, Karen Kear, and Chris Douce. Using a virtual computing lab to teach programming at a distance. In *Computing Education Practice*, pages 5–8. 2023.

[17] Elizabeth J Halcomb and Louise Hickman. Mixed methods research. 2015.

[18] David P Harvie, Jason R Cody, Christopher Morrell, and Tanya T Estes. Using virtual machines to enhance the educational experience in an introductory computing course. In *Proceedings of the 20th Annual SIG Conference on Information Technology Education*, pages 28–32, 2019.

[19] Michael Henderson, Neil Selwyn, and Rachel Aston. What works and why? student perceptions of 'useful'digital technology in university teaching and learning. *Studies in higher education*, 42(8):1567–1579, 2017.

[20] Jason Hennessey and Steven Xijin Ge. A cross disciplinary study of link decay and the effectiveness of mitigation techniques. In *BMC bioinformatics*, volume 14, pages 1–11. BioMed Central, 2013.

[21] Burke Johnson and Lisa A Turner. Data collection strategies in mixed methods research. *Handbook of mixed methods in social and behavioral research*, 10(2):297–319, 2003.

[22] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2):83–137, 2005.

[23] Oren Laadan, Jason Nieh, and Nicolas Viennot. Teaching operating systems using virtual appliances and distributed version control. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 480–484, 2010.

[24] Margaret D LeCompte and Judith Preissle Goetz. Problems of reliability and validity in ethnographic research. *Review of educational research*, 52(1):31–60, 1982.

[25] David J Malan. From cluster to cloud to appliance. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 88–92, 2013.

[26] David J Malan. Standardizing students' programming environments with docker containers: Using visual studio code in the cloud with github codespaces. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2*, pages 599–600, 2022.

[27] David R McIntyre and Francis G Wolff. An experiment with www interactive learning in university education. *Computers & Education*, 31(3):255–264, 1998.

[28] Sean Mealin and Emerson Murphy-Hill. An exploratory study of blind software developers. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 71–74. IEEE, 2012.

[29] Sharan B Merriam. *Qualitative Research and Case Study Applications in Education. Revised and Expanded from" Case Study Research in Education.".* ERIC, 1998.

[30] Fisseha Mikre. The roles of information communication technologies in education: Review article with emphasis to the computer and internet. *Ethiopian Journal of Education and Sciences*, 6(2):109–126, 2011.

[31] Missy Morton and David Mills. Ethnography in education. *Ethnography in Education*, pages 1–200, 2013.

[32] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. Addressing accessibility barriers in programming for people with visual impairments: A literature review. *ACM Transactions on Accessible Computing (TACCESS)*, 15(1):1–26, 2022.

[33] Marc Prensky. The role of technology. *Educational Technology*, 48(6):1–3, 2008.

[34] Jack C Richards and Richard W Schmidt. *Longman dictionary of language teaching and applied linguistics*. Routledge, 2013.

[35] Steven Ross. Program-defining evaluation in a decade of eclecticism. *Evaluation in second language education*, pages 167–195, 1992.

[36] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. Visual programming languages integrated across the curriculum in elementary school: A two year case study using "scratch" in five schools. *Computers & Education*, 97:129–141, 2016.

[37] Debra L Scammon, Andrada Tomoaia-Cotisel, Rachel L Day, Julie Day, Jaewhan Kim, Norman J Waitzman, Timothy W Farrell, and Michael K Magill. Connecting the dots and merging meaning: using mixed methods to study primary care delivery transformation. *Health services research*, 48(6 Pt 2):2181, 2013.

[38] Herbert W Seliger, Elana Goldberg Shohamy, Elana Shohamy, et al. *Second language research methods*. Oxford University Press, 1989.

[39] Neil Selwyn. The use of computer technology in university teaching and learning: a critical perspective. *Journal of computer assisted learning*, 23(2):83–94, 2007.

[40] Brian Sierkowski. Achieving web accessibility. In *Proceedings of the 30th annual ACM SIGUCCS conference on User services*, pages 288–291, 2002.

[41] Volker Sorge, Akashdeep Bansal, Neha M Jadhav, Himanshu Garg, Ayushi Verma, and Meenakshi Balakrishnan. Towards generating web-accessible stem documents from pdf. In *Proceedings of the 17th International Web for All Conference*, pages 1–5, 2020.

[42] Andreas Stefik, Richard E Ladner, William Allee, and Sean Mealin. Computer science principles for teachers of blind and visually impaired students. In *Proceedings of the 50th ACM technical symposium on computer science education*, pages 766–772, 2019.

[43] Marko Teräs, Juha Suoranta, Hanna Teräs, and Mark Curcher. Post-covid-19 education and education technology 'solutionism': A seller's market. *Postdigital Science and Education*, 2(3):863–878, 2020.

[44] Marko Teräs, Hanna Teräs, Patricia Arinto, James Brunton, Daryono Daryono, and Thirumeni Subramaniam. Covid-19 and the push to online learning: Reflections from 5 countries. 2020.

[45] Sander Valstar, William G Griswold, and Leo Porter. Using devcontainers to standardize student development environments: An experience report. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 377–383, 2020.

[46] Wim Vanderbauwhede. Frugal computing–on the need for low-carbon and sustainable computing and the path towards zero-carbon computing. *arXiv preprint arXiv:2303.06642*, 2023.

[47] Lucy Lu Wang, Isabel Cachola, Jonathan Bragg, Evie Yu-Yen Cheng, Chelsea Haupt, Matt Latzke, Bailey Kuehl, Madeleine N van Zuylen, Linda Wagner, and Daniel Weld. Scia11y: Converting scientific papers to accessible html. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*, pages 1–4, 2021.

[48] IS Zinovieva, VO Artemchuk, Anna V Iatsyshyn, OO Popov, VO Kovach, Andrii V Iatsyshyn, YO Romanenko, and OV Radchenko. The use of online coding platforms as additional distance tools in programming education. In *Journal of physics: Conference series*, volume 1840, page 012029. IOP Publishing, 2021.

[49] Mohammad Zohrabi. Mixed method research: Instruments, validity, reliability and reporting findings. *Theory and practice in language studies*, 3(2):254, 2013.

Appendix A Pre-experience Survey

A.1 Informed Consent

I'm Luca De Menego, a master's student in Computer Science at the University of Trento. As part of my thesis, I'm creating an alternative slides system that we would like to propose in some lessons of the `[COURSE NAME]` course.

To reflect on the tool's effectiveness, I would like to ask for your consent to collect some data about you, exclusively for research purposes. And in particular:

- your answers to a couple of questionnaires (including this one);
- field notes (notes on what happens in class) that I will take during the lessons in which we will propose the tool.

The data I will collect, when presented in public (in my thesis, in articles, or in presentations), will be presented in an aggregated or anonymous form to preserve your privacy. For example, we could tell how many students were in the class without giving their names, or quote a particularly interesting response that you gave on a questionnaire, without specifying who said it.

The only people who will be able to see the data in non-anonymous form are:

- `[MAIN COURSE PROFESSOR]`
- Luca De Menego (DISI, developer of the tool)
- Lorenzo Angeli (DISI, principal researcher of the tool, my thesis supervisor)

All the collected information will be treated in compliance with the European regulation on the protection of personal data (GDPR, EU Regulation 2016/679).

If you do not want your data to be collected, you can deny your consent below. If so, I won't take any field notes about you, or include you in any form of data analysis.

Even if you deny your consent, you will always be free to follow the class in the way that you prefer, including using the tool we propose.

You can change your mind (to give or withdraw your consent) at any time by telling one of us, or by sending an email to *luca.demenego@studenti.unitn.it*, CC *lorenzo.angeli@unitn.it*.

```
Q. Do you authorize us to collect data for research purposes, as explained above?
  A. Yes | No
```

A.2 General Data

```
Q. Your Name and Surname

Q. Your Study Course / Degree

Q. Which operating systems do you use on your PC?
  A. GNU/Linux | MacOS | Windows 11 | Windows 10 | Windows 8.x | Windows 7 |
     Windows Vista | Windows, but older than Vista | Others...
```

```
Q. Do you already know some programming languages?
  A. C/C++ | C#/.NET | Java | Python | Javascript/Typescript | NodeJS | Go |
     Rust | PHP | Kotlin | Objective C/Swift | R | Others...

Q. Did you encounter any difficulty installing the software required for
   the course?

Q. How motivated do you feel to learn to code?
  A. [1 - Little, ..., 5 - Very]

Q. What are the main reasons that motivate you in learning programming?

Q. How competent do you feel about using computers in general?
  A. [1 - Little, ..., 5 - Very]

Q. How competent do you feel at programming?
  A. [1 - Little, ..., 5 - Very]

Q. More in depth, how would you describe your coding competence?
  A. [1 - Little (I started when I enrolled in university), ...,
      5 - Very (I already work autonomously on complex project)]

Q. How familiar do you feel with [COURSE PROGRAMMING LANGUAGE]?
  A. [1 - Little, ..., 5 - Very]

Q. What do you think is the first obstacle for a person who wants to
   learn to code (or, what was your first obstacle in learning how to code)?

Q. How difficult do you think these elements of learning to code can be?
     - Installing the development environment
     - Fixing technical problems (unexpected errors, compatibility issues, ...)
     - Following the lesson without falling behind / ahead
     - Learning code syntax (semicolons, parentheses, keywords..)
     - Understanding what the code does
     - Translating an idea into working code
     - Translating complex maths into code
     - Understanding math's underlying programming
     - Debugging (figuring out what I did wrong in my code)

  A. Not at All | A little | Enough | Very | Extremely

Q. What is the one thing you expect you will have to spend the most time on
   during this course?

  A. - Installing the development environment
     - Fixing technical problems (unexpected errors, compatibility issues, ...)
     - Learning code syntax (semicolons, parentheses, keywords..)
     - Understanding what the code does
     - Translating an idea into working code
     - Translating complex maths into code
     - Understanding math's underlying programming
     - Debugging (figuring out what I did wrong in my code)

Q. Why so?
```

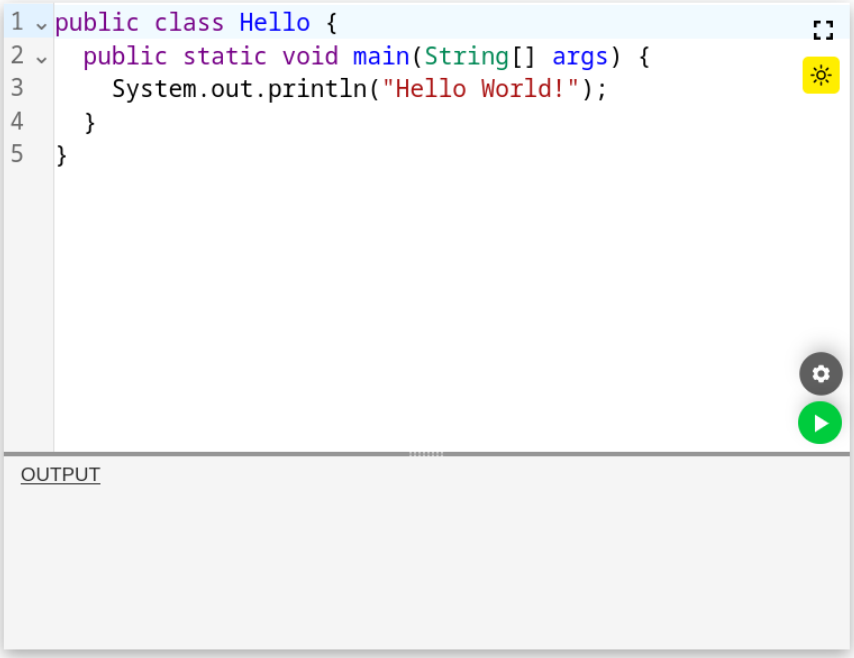
A.3 The proposed tool

In this last section of the questionnaire we will see some screenshots of the proposed tool.

What you will see is a tool that allows you to execute exercises code directly inside the slides. The slides we will propose, however, will not be PDFs: they will be web pages accessible via a browser.

If you have problems accessing the pictures below, please let us know and we will provide a more accessible version.

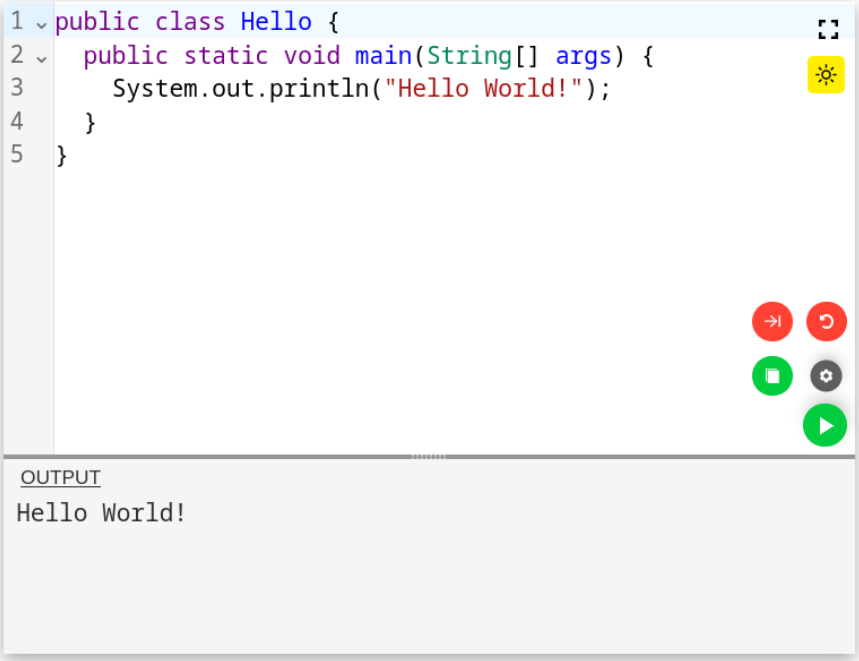
`This is a screenshot of (a part of) the tool we will propose.`



`Q. Looking at the picture, do you think you would be able to...`

- `Execute the code`
- `Find where the code execution result will appear`
- `Modify the code`
- `Change the editor's theme`
- `Enlarge the editor to full-screen mode`
- `Access additional settings`
- `None of the above`
- `Others...`

The settings button on the right provides three more buttons

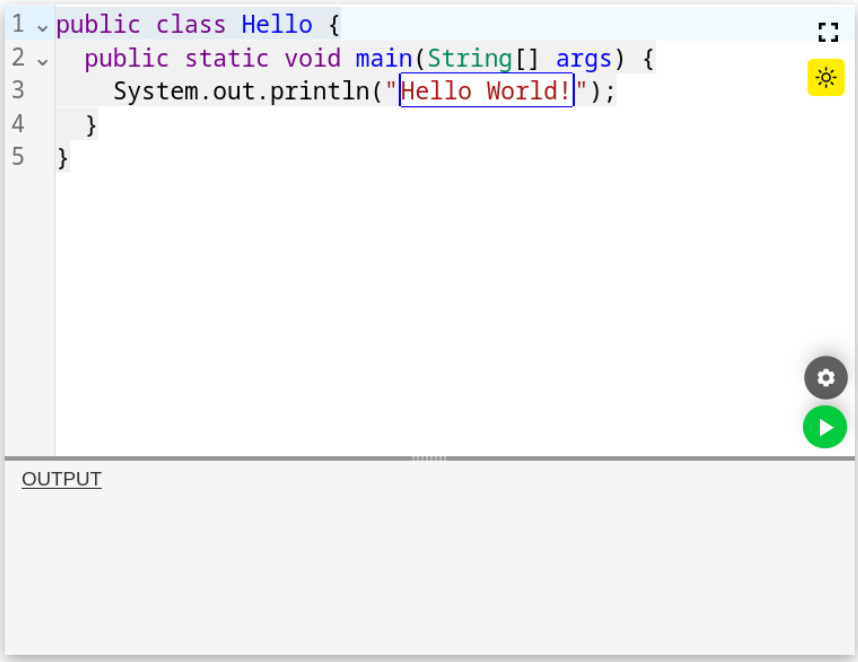


Q. What do you think this button does?

Q. What do you think this button does?

Q. What do you think this button does?

Q. In this case, part of the code inside the editor has a blue outline, while the rest of the code is gray. What do you think it might mean?



Q. How important do you consider the following things to be?
- Availability of slides compatible with any device
- Being able to access slides while offline
- Copying and pasting code from the slides
- Searching for keywords (CTRL+F)
- Having an identical development environment to that of other students
- Being able to print the slides on paper
- Taking handwritten notes on the slides
- Availability of the slides in PDF format

A. Not at All | A little | Enough | Very | Extremely

Q. Do you have any additional comments?

A.4 Survey Completed

Thanks for following the survey. Remember to finish by submitting the form.

In case you would like more information about the proposed educational tool, the source code is publicly available on GitHub:

- Source Code: `https://github.com/lucademenego99/icp-bundle`
- Demo: `https://lucademenego99.github.io/icp-slides`

Appendix B Post-Experience Survey

Q. Your Name and Surname

Q. How difficult was it for you to do these things so far?
- Fixing technical problems (unexpected errors, compatibility issues, ...)
- Following the lesson without falling behind / ahead
- Learning code syntax (semicolons, parentheses, keywords..)
- Understanding what the code does
- Finding a strategy to solve a problem
- Translating an idea into working code
- Translating complex maths into code
- Understanding math's underlying programming
- Debugging (figuring out what I did wrong in my code)

A. Not at All | A little | Enough | Very | Extremely

Q. Which of these things do you think you spent the most time on?

A. - Fixing technical problems (unexpected errors, compatibility issues, ...)
- Learning code syntax (semicolons, parentheses, keywords..)
- Understanding what the code does
- Finding a strategy to solve a problem
- Translating an idea into working code
- Translating complex maths into code
- Understanding math's underlying programming
- Debugging (figuring out what I did wrong in my code)

Q. How much did you use the interactive slides when they were proposed?
A. [1 - Never, ..., 5 - Whenever they have been proposed]

Q. What do you think was the quality level of the slides for...
- The clarity of the graphics
- The clarity of the information contained
- The convenience of use
- The effectiveness of the exercises
- The performance (execution speed)

A. Bad | Poor | Acceptable | Good | Excellent

Q. How helpful did you find that slides could help you do these things?
- Being able to stay in the browser without having to use other programs
- Having exercises that draw attention on the right concepts
- Being able to go forward/backward freely, choosing my own pace
- Being able to make changes "on the fly" to the proposed exercises
- Being able to compare my code with that of my classmates, having identical development environments

```
     - Having a button to copy and paste code quickly
     - Being able to change the theme of the editor

  A. Not at All | A little | Enough | Very | Extremely

Q. Which of these slides features was most useful to you?

  A. - Being able to stay in the browser without having to use other programs
     - Having exercises that draw attention on the right concepts
     - Being able to go forward/backward freely, choosing my own pace
     - Being able to make changes "on the fly" to the proposed exercises
     - Being able to compare my code with that of my classmates, having identical development env
     - Having a button to copy and paste code quickly
     - Being able to change the theme of the editor
     - Others...

Q. Why?

Q. Do you prefer static or interactive didactic material?
  A. [1 - Static material, ..., 5 - Interactive material]

Q. How often did you use these study tools (when they were available)?
     - PDF Slides
     - Interactive Slides
     - Notes taken digitally
     - Notes taken on paper
     - Others...

  A. Never | Rarely | Sometimes | Often | Always

Q. Of these development environments, which did you find most familiar?
  A. - [COURSE DEVELOPMENT ENVIRONMENT]
     - Interactive Slides

Q. How many technical problems have you encountered using [COURSE DEVELOPMENT ENVIRONMENT]?
  A. [1 - None, everything went smoothly, ...,
      5 - Too many, there was always something wrong]

Q. How many technical problems have you encountered using the interactive slides?
  A. [1 - None, everything went smoothly, ...,
      5 - Too many, there was always something wrong]

Q. How much did differences between the results of the interactive slides and
   [COURSE DEVELOPMENT ENVIRONMENT] affect your experience?
  A. [1 - Not at all / I did not find any differences, ..., 5 - Extremely]

Q. Is there anything in general that you would like us to improve on the slide system
   we have proposed?

Q. Does it seem to you that the proposed slide system is missing some technical features?

Q. Did you find any bugs (technical errors) in the slides?

Q. Any other comments?
```