



**Treball de Fi de Grau**

**GRAU D'ENGINYERIA INFORMÀTICA**

**Facultat de Matemàtiques i Informàtica**

**Universitat de Barcelona**

---

**JUPYTER NOTEBOOKS AS A DEVELOPMENT  
AND DOCUMENTATION TOOL FOR  
SUPPORTING COMPUTER PROGRAMMING  
LEARNING AMONG ADOLESCENTS: A CASE  
STUDY IN A K-18 SCHOOL**

---

**Ferran Mañà Marín**

Director: Sergio Sayago

Realitzat a: Departament de

Matemàtiques i Informàtica

Barcelona, 1 de febrer de 2018

**Keywords:**

Jupyter notebooks, Computational narratives, Literate programming, Code learning

## Abstract

This Treball Fi de Grau (TFG) reports on an exploratory case study aimed at facilitating computer programming learning in a K-18 school through Jupyter Notebooks, to test their usability for this user group, and find possible improvements to the interface. Over a period of 4 months, we were in charge of running an extracurricular activity intended to train a team of students to participate in a competition of code challenges, HP Code Wars, thereby adopting a learning-service approach (aprenentatge servei). Within this context, we looked into different aspects of the use of computational notebooks, which were not used in the educational institution, and came up with some possible features to add to the user interface of Jupyter Notebooks to serve better our students' needs and encouraging good programming practices. From these results we designed a variable inspector for Jupyter, and co-designed an extension that allows the user to add a second dimension to the narrative, where students seemed to agree on a cell folding/grouping approach for a multi-layered structure. The evaluation activities yielded positive results, with students preferring the use of notebooks over interpreters, and documenting their work in an explanatory narrative.

## Resum

*Aquest Treball de Fi de Grau (TFG) informa d'un cas d'estudi exploratori per a facilitar l'aprenentatge de programació en una escola 3-18 a través de Jupyter notebooks, examinant la seva usabilitat en aquest perfil d'usuaris, i trobar possibles millores a la interfície. Al llarg d'un període de 4 mesos, vam estar a càrrec d'organitzar una activitat extraescolar en la qual hem preparat un equip d'estudiants per a participar en una competició de reptes de programació, HP Code wars, adoptant així un enfocament*

*d'aprenentatge servei. En aquest context, hem analitzat diferents aspectes de l'ús d'aquests notebooks computacionals, que no s'utilitzaven en aquesta institució, i vam pensar en possibles funcionalitats per a afegir a la interfície de Jupyter Notebooks, tot ajustant-nos a les necessitats dels aprenents i incentivant bones pràctiques de programació. A partir d'aquests resultats hem dissenyat un inspector de variables per a Jupyter, i co-dissenyat una extensió que permet a l'usuari d'afegir una segona dimensió a la narrativa, en la qual els estudiants semblaven estar d'acord en un model basat en agrupació/plegament de cel·les per aconseguir aquesta estructura de diverses capes. Les activitats d'avaluació han donat resultats positius, amb els estudiants preferint l'ús de notebooks per damunt d'interprets, i per a documentar la seva feina en una narrativa explicatòria.*

## Resumen

*Este Trabajo de Fin de Grado (TFG) informa de un caso de estudio exploratorio para facilitar el aprendizaje de programación en una escuela 3-18 a través de Jupyter notebooks, examinando su usabilidad en este perfil de usuarios, y buscando posibles mejoras en la interfaz. A lo largo de un período de 4 meses, estuvimos a cargo de organizar una actividad extraescolar en la que hemos preparado un equipo de estudiantes para participar en una competición de retos de programación, HP Code wars, adoptando así un enfoque de aprendizaje servicio. En este contexto, hemos analizado diferentes aspectos del uso de estos notebooks computacionales, que no se utilizaban en esta institución, y pensamos en posibles funcionalidades para añadir a la interfaz de Jupyter Notebooks, ajustándose a las necesidades los aprendices e incentivando buenas prácticas de programación. A partir de estos resultados hemos diseñado un inspector de variables para Jupyter, y co-diseñado una extensión que permite al usuario añadir una segunda dimensión a la narrativa, en la que los estudiantes parecían estar de acuerdo en un modelo basado en agrupación / plegamiento de celdas para conseguir esta estructura de*

*varias capas. Las actividades de evaluación han dado resultados positivos, con los estudiantes prefiriendo el uso de notebooks por encima de intérpretes, y para documentar su trabajo en una narrativa explicatòria.*

I would like to thank Sergio Sayago for his implication in this project  
Salva, Dani, and the rest of the TAC department for their help,

and some very special thanks to:

Guillermo

Jara

Marc

Eva

Jordi

Xavi

For their huge commitment and support during the activities

# Table of contents

<b>1. Introduction</b>	<b>7</b>
<b>2. Objectives</b>	<b>10</b>
<b>3. Context and related work</b>	<b>11</b>
3.1. The role of Computer Science in education	11
3.1.1 Our case: Daina Isard K18 school	14
3.2. Programming environments for a first contact	17
3.2.1. Our case: HP CodeWars	22
3.3. Research activity: first contact with Python	24
3.3.1. Choosing an interface to evaluate	31
<b>4. Notebooks and computational narratives</b>	<b>32</b>
4.1. Literate programming and computational narratives	32
4.2. Project Jupyter	33
4.3. Related work	35
4.4. Research activity: using Jupyter as an IDE	38
4.5. Research activity: using Jupyter to make programming notes	44
<b>5. Software design</b>	<b>51</b>
5.1. Requirements	51
5.2. Co-designing activity for bi-dimensional notebooks	53
5.3. Prototyping and relevant art for variable inspection	57
<b>6. Conclusions and future work</b>	<b>59</b>
<b>7. References</b>	<b>61</b>

# 1. Introduction

In a time where digital technologies are taking over many aspects of daily life, the concept of **code literacy** is on the table of engineers and educators alike. Being literate in code means that you can read and understand computer code, but also write it, opening up software development and use, something which, like in other cases in history, has serious implications in society [25]. This interest in code literacy is related to other concepts, such as **computational thinking**, and how it can help to understand technology as well as other disciplines; or **software democratization**, which enables unexperienced programmers to customize commercial products through coding, or the construction of complex digital platforms, for example, using frameworks, which are gaining traction. In this regard, studies have shown that most Computer Science undergraduate students have taken some kind of programming course in high school, often it being extracurricular [16]. This renders teenage years as crucial for engaging students of all conditions in these career choices, in order to meet society's technological needs and to promote income and gender equality in increasingly more areas [3], and puts technologists and software engineers under the spotlight, for them to design according to this new population which is educated on computers and programming, as well as deciding how to open and present the content that they will leave for non-professionals to tweak with, or that educators will use to teach coding skills to new generations. Furthermore, the transition towards this model has to take into account the current motivation of students for learning programming, the interest of their tutors, and the goals and methods of teachers.

In this project we will look into a currently widespread method for distributing computational and statistical knowledge, in the form of **computational notebooks**, which take at their heart **literate programming** [4]. In particular, we will focus on Jupyter Notebooks, which are becoming widely spread and recently awarded (2017 ACM Software System Award)[8]. There are other computational notebooks, such as RStudio, ObservableHQ,

Mozilla Iodide, or Codestrates, but they are not as widely used as Jupyter and most of them do not support the programming language that we chose, which is Python.

Jupyter notebooks combine cells of code and formatted text; allowing to build **computational narratives** that break down and atomize code, while keeping it verbalized and understandable. Apart from the cells, the other main “new” feature that notebooks introduce is markdown notation for the code’s documentation, also known as explanation. More specifically, we will check to which extent these features students have a deeper understanding of their code through a narrative and explanatory scheme, if they do use them.

The research part of this project has been done in the form of a weekly activity about code learning with Python, and three occasional activities involving Jupyter notebooks with a group of students from a high school institution in Olesa de Montserrat, Catalunya. The students in this group, aged 15 to 16, have taken an optional extra-curricular activity, in which they learn high-level programming for problem solving and learning about computers and algorithms, with the end goal of participating in the *HP Code wars* competition. This was done with consent from the school and the students. The activities were designed to clarify some concerns of researchers about the use that is given to the Jupyter interface. These include: to which point users document their code with text cells, how much of the exploratory process involving data science and other script-based disciplines is supported by the interface, how is the code structured, and how it is supposed to be navigated through. Additionally, there is interest in what are the different ways people use notebooks and the purpose that they give them. This evaluation of the interface was done through two use cases of notebooks given the circumstances: using them to develop solutions to the challenges of the competition, and using them to document what they learn for lookup during the competition. The first activity, which focuses on the exploratory part, has been done with all the students at once, while the other one, focusing on explanation, was done separately with two students,



watching their behavior while using the interface in a pair programming setup.

The resulting notebooks show how they generate interest in students, who use cells to type and test their code by parts, and that they preferred this approach to a single text file to execute. However, students who followed an imperative programming paradigm tended to concentrate algorithms in only one cell, using other cells to test different inputs or output methods for that same algorithm. In terms of usability, the interface proved to be simple and intuitive enough. For the explanatory part, students were able to craft documents with a reasonable structure and purpose, but made little use of the text cells. Support for multi-dimensionality, and visualization of the current state of variables in notebooks could help them become more enriching and better represent what the students want to annotate.

These results have served as a foundation for a software design process with the objective of implementing some of the improvements that came out of the study, one more related to code development, and another to navigation through the narrative. We also included a co-design activity with some of the involved students.

## 2. Objectives

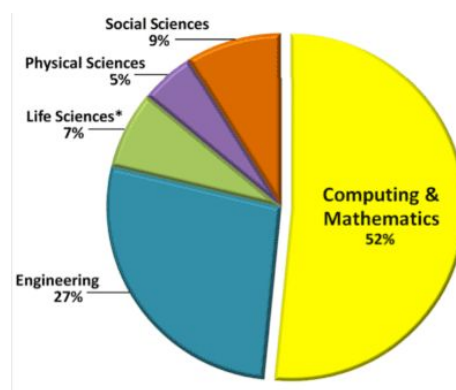
1. Find relevant aspects of notebook interfaces to evaluate them in a school environment.
2. Contextualize the setting that we are going to work with in order to find what to look for, and extract relevant conclusions from the study.
3. Effectively implement a software solution for the students participating in the competition.
4. Investigate on whether these interfaces, specifically Jupyter notebooks, can make a positive difference for novice programmers.
5. Design or co-design improvements for the interface.

### 3. Context and related work

*In this section we cover the background necessary to support the activities carried out with computational notebooks, covering the context, which is secondary education in Catalonia, and one specific high school with a robotics and programming itinerary; studying interfaces with a similar purpose, and the activity that encompasses the fieldwork.*

#### 3.1. The role of Computer Science in education

Nowadays, there is a growing interest in teaching Computer Science (CS) in K-12 and K-18 schools. [20]. A number of reasons account for this interest, ranging from job opportunities in a labor market where digital technologies play a key role, to the benefits or potential of CS. Nevertheless, STEM (Science, Technology, Engineering and Mathematics) career paths are expected to grow in demand (see Figure 1), so encouraging teenagers to pick a career in these areas, and more specifically in Computer Science, is bound to be a priority for most administrations [2]. Computer programming is an important part of Computer Science, and seems to be the link to all other disciplines, since most university curriculums of STEM degrees include programming.



(Figure 1) Graph showing the projected Annual Growth of STEM Job Openings from 2010 to 2020 [10]

However, there is something about Computer Science that makes it useful and appealing for educators beyond the labor factor. In a way, a computer and the act of programming it is not something to be learned *per se*, it is something to create with, which can be used to solve problems, encode and visualize information, and share those creations [15]. Programming is a great way of simulating real-world problems and situations where you can see the output of what you have developed, along with the process of making it behave in the way you wanted. This means that Computer Science has the potential to become a core subject. This calls for programming environments which are not focused on software engineering, but on exploiting these properties of Computer Science in order to develop useful skills for a variety of fields and situations. Some of these skills are summarized under the umbrella of **computational thinking**, conceived of as a thought process that formulates a problem and expresses its solution so that a computer can carry it out. In the same way that studying our language or mathematics helps us understand aspects of the world, computational thinking allows us to understand what goes on behind the software that we use or that is used around us, and provides us with resources to break down problems. In this sense, computers and how they are used are secondary: humans also compute [26].

We can establish a connection with the definition of computer science [1] and computational thinking; for instance, in the idea of abstraction, which is key in both concepts. Apart from that, computational thinking has been defined in terms of how other disciplines can incorporate algorithmic and programming elements. One good example is biology, which has seen the rise of a new field, bioinformatics. Besides, considering how CS is intrinsically mathematic in nature, we can also state a strong relationship between mathematical thinking and computational thinking. This goes against the idea that teaching programming is only useful for those that will end up taking Computer Science as their major, or software development as their career. This generalistic idea of computation shows how intertwining computer science with other subjects makes sense on its own, without having to picture a scenario of necessity in a computer-run and

algorithm-governed world: that would only be one more incentive. Another incentive could be how effective learning *through* programming proves to be.

In Catalonia, where this project takes place, the education department of the regional government classifies programming (in the secondary school, 12-16) along with robotics in the “digital” curriculum, and establishes a relation of “study and use” with the mathematics and technology curriculum, and a “learning through use” with science [12]. The “robotics and programming” skill is also linked with the following cross-curricular skills:

- Selecting, configuring and programming devices according to the tasks to complete.
- Developing new personal knowledge through information treatment strategies with the support of digital applications.
- Carrying out group activities while using tools and virtual environments of collaborative work.

The network of the catalan school community, created to share learning resources, provides mainly robotics initiatives, many having to do with Arduino. The ones centered in computers are either about Scratch or App Inventor [13]. A part from these guidelines and resources, it is up to each center how and to which extent they teach programming during high school years [7].

### 3.1.1 Our case: Daina Isard K18 school

The work done in this project is focused on the context of a school in Olesa de Montserrat named Daina Isard (Figure 2), located in the province of Barcelona, which covers education from 3 to 18 years. The study plan integrates programming at different levels from early on, with the end objective that all students finish school knowing how to code.



(Figure 2) The school's logo and view.

We have had access to the study plan that this institution has on robotics and programming, and it contains activities from very early on and up until 15-16 years of age (Figure 3), at which point all activities become optional and extracurricular. We have classified these according to the approach taken (Table 1), whether it is using code to achieve some other task or project besides programming a computer, or, by contrast, learning to program through an environment that is more or less enriched, in order to provide a relevant and satisfactory experience for beginners. Lighter and in italics are those activities which are extracurricular or optional.



(Figure 3) Slide of the presentation for the 3-18 project.

Ages	Supporting projects with code	Programming for the sake of programming (Computer Science)
3-6		<b>BEE-BOT:</b> a programmable bee that executes previously programmed movements in either four directions
6-8	<p><b>Lego WeDo</b>, making a toy (Robotics)</p> <p><b>Scratch</b>, making a videogame (Videogames)</p> <p><i>ROBO-TIC workshop</i></p>	
8-12	<p><b>Lego Mindstorms</b>, building an “eco-city” (Robotics, multi-disciplinary)</p> <p><i>ROBO-TIC workshop</i></p>	<b>Scratch</b> , various challenges
12-13	<p><b>Drone challenge</b> (Technology, Swift)</p> <p><b>Snap4Arduino</b>, domotics in a house model (Multi-disciplinary)</p> <p><i>ROBO-TIC workshop</i></p>	<b>Swift Playgrounds:</b> controlling graphical agents, first contact with <i>textual</i> code
13-14	<p><b>Arduino</b>, working with the board (Electronics, Technology)</p> <p><i>ROBO-TIC workshop</i></p>	
14-15	<p><b>First Lego League</b> (Robotics)</p> <p><i>ROBO-TIC workshop</i></p>	
15-16	<p><b>Arduino, Picaxe</b> (Electronics)</p> <p><b>3D printing</b></p>	<b>App Inventor</b> , Android app development

	<b>Kodu</b> (Videogames) <i>ROBO-TIC workshop</i> <i>Video Games workshop</i>	<b>HP CodeWars</b> , <i>problem solving through scripts with python</i>
16-18	<i>Video Games workshop</i>	

(Table 1)

The activity highlighted in blue is the one that was introduced for this project, and its end goal is to be able to demonstrate the mastery of a general-purpose language such as Python or C++ when solving several problems programmatically, designing and applying algorithms.

From the plan's detail and the activities, as well as conversations with the TAC (Technologies applied to knowledge) team, we can extract that the motivation behind this itinerary is that students develop a programmatic problem-solving ability, through computational thinking. In this sense, they view these skills as part of a process which starts at analyzing the problem, and synthesizing it, that is, extracting the relevant parts and making sense of them.

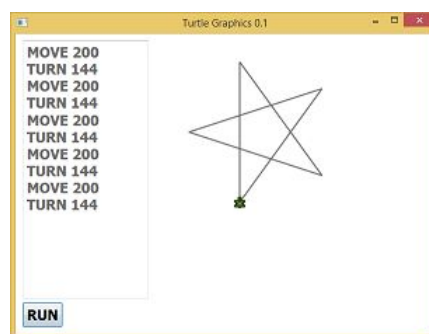
The students that we will be working with in this project have not gone through this exact itinerary. The main differences, though, are in the K8 span, so, in principle, them having interacted with Lego's solutions, as well as Scratch and Swift, gives us a good approximation of the students under this plan. The activity that they will take part in is optional, which means that the students taking part in the study are interested in learning to program and participating in the contest.

The next step is to study the tools that these students have worked with to find patterns and peculiarities in their design, and the principles they follow. One of them is Scratch, a web application or desktop program by the MIT Media Lab, aimed at children. The other is Swift Playgrounds, an app for the iPad for learning different uses of Apple's Swift programming language.



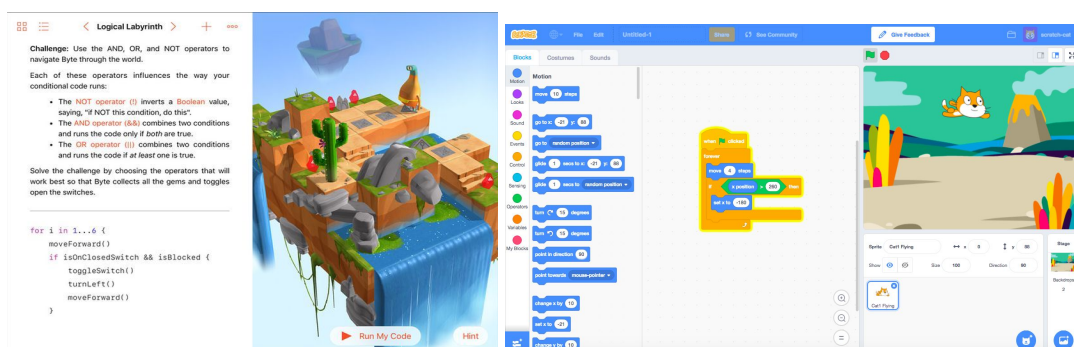
### 3.2. Programming environments for a first contact: Scratch and Swift Playgrounds

We can draw a line of progressive abstraction from the beginning of electronic computation, in the 1960's until now, both in programming languages and programming languages with educational purposes. Languages like **BASIC** or **Logo** allowed people that did not know about a computer's architecture to run simple programs in it. What Logo introduced was the ability to show the result of the program with **turtle graphics**: the program controls a cursor, the turtle, which draws on the screen in a sequential manner (Figure 4) [17].



(Figure 4) A screenshot with some version of the logo programming language.

Both Scratch and Swift Playgrounds implement turtle graphics in a sort of way, as they preserve the idea of watching the output of your program as it progressively unfolds in a canvas (Figure 5).

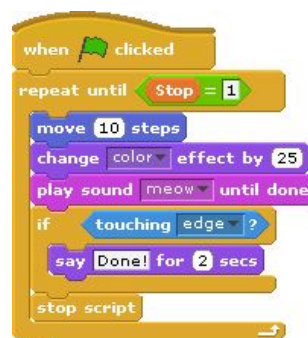


(Figure 5) At the left, a screenshot from a Swift Playgrounds lesson, (Figure 6), at the right, shows the web interface of Scratch 3.0

Software for learning to program often enables users to build programs in a safe and limited context, so that afterwards students can move on to general-purpose languages [18]. Seymour Papert, the creator of the Logo programming language, set three conditions for a good first-contact programming experience: it must have a **low floor**, as in, it must be easy to get started with it; a **high ceiling**, meaning it can reach a high level of complexity and sophistication; and finally, another MIT professor, M. Resnick, added the concept of **wide walls**: “kids must be able to explore multiple pathways from floor to ceiling”, so that they can have personal experiences, which are individually meaningful to them [19]. We will use this criterion to assess the validity of the two pieces of software that we will be evaluating prior to the investigation with notebooks.

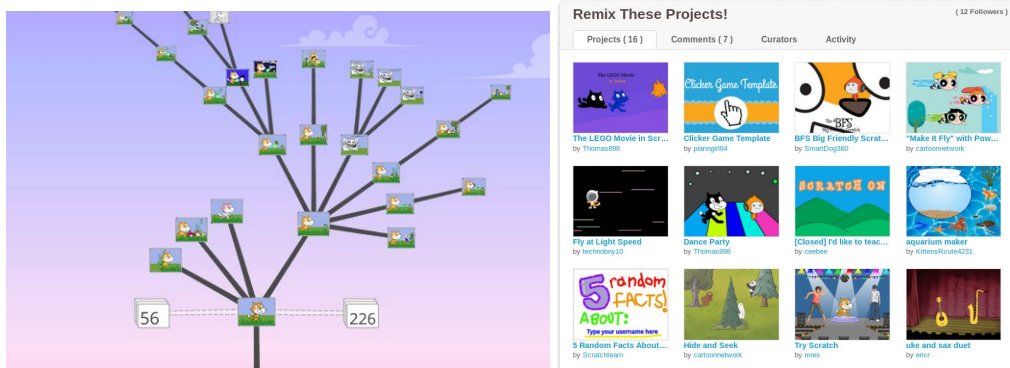
## Low floor

Scratch’s approach for writing a program is visual programming. More specifically, Scratch introduced the concept of blocks that connect with one another vertically, only being able to combine blocks that are compatible, thus removing all syntax errors and leaving only the semantic ones. By having all possible commands (blocks) in sight, and restricting their compatibility, students do not need to memorize them, or learn what are they compatible with: it is visually evident by seeing their *in* and *out* shapes. Also, the blocks’ color differentiates them according to their purpose: statements, routines, input, output... (Figure 7)



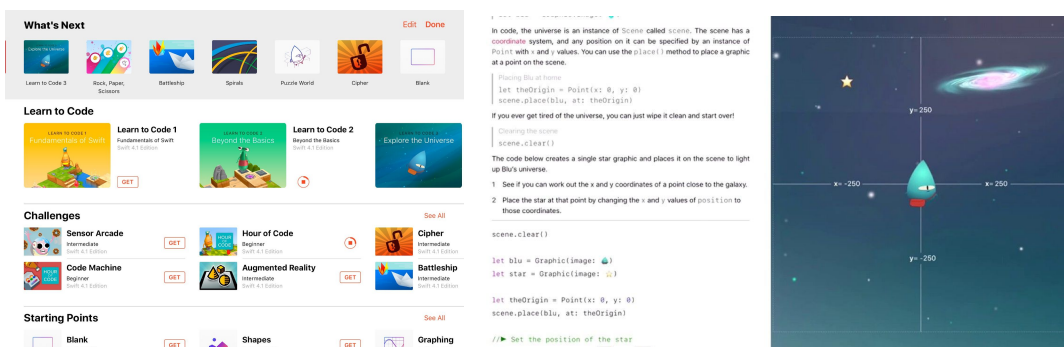
(Figure 7) An example of a Scratch script, made with connecting blocks.

Scratch is renowned for being widely used and having a social component, as it allows its users to share their creations, and ‘remix’ each other’s programs. This way, they can get ideas, communicate, and generate deep and meaningful experiences through coding. This is easily done through scratch’s website. The recent addition of a web editor makes the software more accessible to students around the world.



(Figures 8, 9) Some screenshots from Scratch’s remix system

In Swift Playgrounds, on the contrary, learners undertake a guided process (Figure 11), where there is a desired output, and they often have to fill in the gaps of a piece of code in order to get there. The options they have to fill these gaps can be drawn from a selection menu that contains all the statements and functions which are relevant for the lesson. Also, there are different “playgrounds” - which are thematic boxes in the form of courses, stories or games; and each of these can require a different knowledge of programming, which can be none (Figure 10).

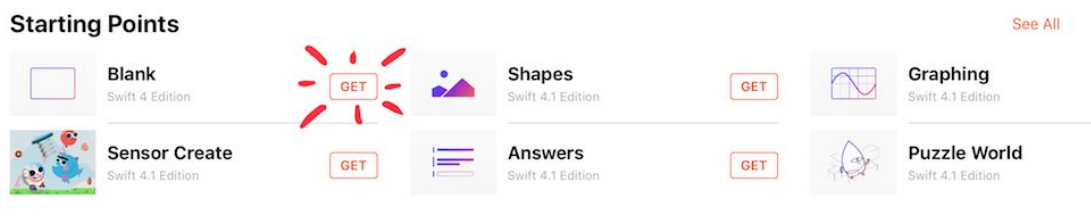


(Figures 10, 11) Swift Playgrounds’ “store”, and an example of a lesson involving code and explanation.

## High ceiling

In Scratch, because it is thought of as a sort of sandbox, or IDE, in which there are usually no restrictions on how the program should behave, there are no right or wrong answers, and the result can always be improved or tweaked. The program is thought for continuously trying, failing, going back and fixing the code, as well as computing what the code should do mentally, which are all useful programming practices. Animations, videogames and applets can easily grow in complexity without the need for mathematical or algorithmic knowledge. There is some debate on visual languages and the amount of code they can fit while maintaining healthy practices like scalability and modularity (see Deutsch limit). In this sense, one could say that space is a limitation for Scratch.

In Playgrounds there are limitations to what you can do inside a specific lesson, so in this case it would depend of how complex lessons - and the coding implied - can get. However, one can also choose “Starting points”, including one called “Blank”, where users can freely write code in Swift and see the output. Some starting points include *Shapes*, for visualizing geometry, *Answers*, for mapping inputs to outputs to create chatbots or quizzes, or *Puzzle World*, to experiment with moving agents around obstacle courses or mazes - a sandbox version of some of the *Learning to Code* courses (Figure 12).



(Figure 12) Some examples of Swift playgrounds' sandboxes: “Starting points”

Since these environments offer an endless experience with a general-purpose language with graphics, physics and graphing libraries, among others, and allow the user to tweak aspects of their code to a considerable extent, a high ceiling for Swift Playgrounds is also justified.

## **Wide walls**

Among all the things one can do with Scratch we can list animations, videogames and applets, and all these visual and interactive experiences count as different ways of getting to complex creations [9]. At the same time, the different playgrounds available let the user choose many different paths towards complex and meaningful programming creations and knowledge.

### 3.2.1. Our case: HP CodeWars

After meeting with two members of the TAC team in the school and informing them of my interest in taking part in some of their code learning activities, they offered us the possibility to teach some programming language to a group of 6 students in order to participate in the HP CodeWars competition (Figure 13) [11]. Three of them had participated in the previous edition, without any previous training, and the remaining three had not. We gladly accepted the task. In total, we ran 13 “classroom” hours - one hour a week. The teachers of the TAC department were also available and willing to exchange information about their work, in the form of a conversation about their motivations and goals with programming, and providing us with the detail of their programming and robotics plan for the school, summarized in table 1.



(Figure 13) The competition's graphic art.

HP CodeWars is an event held in Barcelona since 2015 by the Spanish division of Hewlett-Packard, with the aim of raising interest in STEM careers in students. It consists of a competition where teams of three members are given a list of 30 problems that they have to solve using a programming language like Java, Python, or C++. These problems require increasingly more algorithmic skills, as well as a deep knowledge of the language's resources. Every problem is specified with an explanatory text that gives it a context, a specification of the input that the judges will type, a specification of the output that is expected, and then an example of one possible input and its corresponding output (Figure 15).



(Figure 14) A photo during a competition of a previous edition of HP Code wars.

## 1 Savings for the Fallas

1 point

### Introduction

Ana wants to know how much money she will spend to travel from Barcelona to Valencia to enjoy the “Fallas”. She only has a few days to continue saving money to go. Ana will send you a list with the prices of the train tickets. The first line will be the one-way ticket and the second will be the return ticket. Please, can you tell her how many euros she should save?

Come on, help Ana!

### Input

The input consists of two integers in two lines:

<One-way ticket cost>

<Return ticket cost>

### Output

Print out the total of euros that should save following this output format:

Ana should save a total of <total euros> euros.

### Example

#### Input

50

35

#### Output

Ana should save a total of 85 euros.

(Figure 15) The first problem of the 2018 edition of the competition



### 3.3. Research activity: first contact with Python



#### Introduction

This fieldwork reports on an ongoing extracurricular activity taking place at the Daina-Isard school in Catalonia, with teenagers with ages from 15 to 16, who are interested in participating in the HP Code wars competition.

During the first two months of the activity, we were in charge of teaching the Python programming language and problem-solving skills to the students, without any specific interface in mind. This work is the predecessor of the following studies involving notebooks, and sought to gather information about the usability, needs and motivations of students when using Python for learning how to code, and the code challenges provided by the previous HP Code wars competitions as a guideline.



## Related work

Since its early stages, Python has always been praised as a good language for novice programmers, although it is not always the preferred option [24]. Its simple syntax, which allows a high level of abstraction, and avoids curly braces and semicolons, has proved to be easy to learn, which lets learners focus on creating more complex algorithms. It being interpreted and easy to work through scripts also helps at learning the basics of coding.

## Objectives

During these sessions, we will look for problems, motivations, and other particularities at:

- Understanding Python's syntax.
- Working with the interpreter.
- Debugging their code.
- Following good code practices.

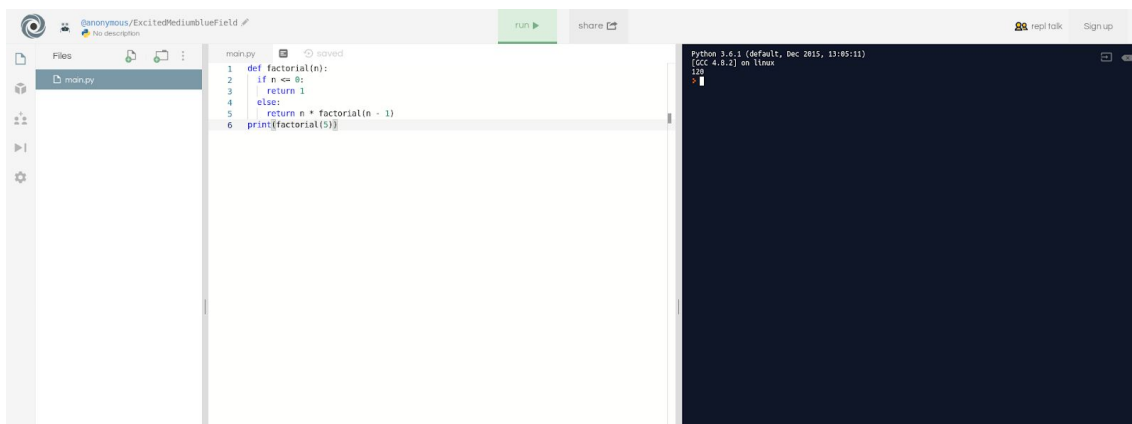
## Methodology

The activity started on October the 9th 2018, and took place every tuesday from 2 pm to 3 pm. The class usually followed this structure:

- Some discussion or lesson on algorithms and/or python using either:
  - The blackboard, to graphically model problems, data, computer science and algorithms, enabling participation from the students.
  - Jupyter notebooks we brought prepared with interesting python functions, which the students had to interpret (execute mentally), and collaboratively discuss, suggest possible changes, or come up with other ideas.
- Solving problems of the competition in groups of 3, pairs or individually.
  - They were free to choose the grouping.

- We followed the order of the competition, so the problems increased in complexity each class.

From the first session, students started coding with repl.it, an online interpreter for multiple languages, as it provided a quick access to the language, optimized for quick scripts, and with no need to install anything in the computer, and we set it to python 2.7 or 3.x. The interface is simple (Figure 16), it has one side for code and another one for execution (input and results) [6]. We observed whether any aspect of the interface could be relevant for the task, or if it could be improved.



(Figure 16) Repl.it's interface

## Results

**On using the competition's problems as a guideline:** we could say that the HP Codewars as a learning context does have a "low floor", because the students did not have many problems understanding and setting out to coding the solutions to the problems. Python's simple and readable syntax also plays a role, but, for example, the competition does not specify which software the students should be using to develop their solutions, and does not provide resources and materials for the competition, so it was up to us to find a suitable software and extract the necessary knowledge from the problems to be able to teach how to solve them. For those students who want to go further than what is taught in the class, the problems of the competition go up to a very high level, involving algorithms with

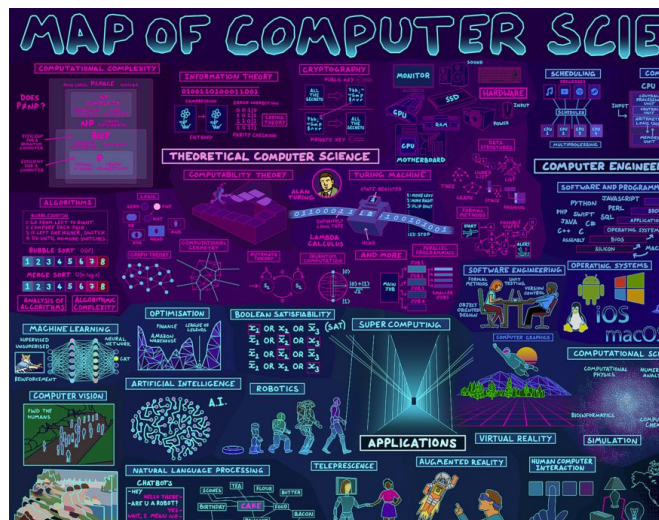
2-dimensional mazes, graph algorithms (such as Dijkstra’s), or string metrics; which sets a “high ceiling” for the challenge.

**On showing them interesting algorithms:** we got to install the Anaconda Python distribution in one computer that was not protected, and we used it to show the students pre-crafted notebooks with functions that they could find interesting (Figure 17), which they did. Students then tried to replicate some of the code practices they had seen in the notebooks.

```
def is_prime(N):  
    if N <= 1:  
        return False  
  
    for factor in range(N // 2):  
        if N % factor == 0:  
            return False  
  
    return True
```

(Figure 17) One of the algorithms they were shown.

**On discussing computer science topics:** on one occasion, we showed them a map of computer science to talk about its different areas and how they relate to each other (Figure 18). Students were interested in cyber-security, and we talked about complexity and problems where algorithms like A\* can be applied, such as TSP or SAT solving.



(Figure 18) The resource we worked with when discussing the different fields of Computer Science.

**On using repl.it:** Students quickly got used to the interface, since they were already familiar with this way of coding where one button executes all the visible code from the first line to the last one, (they had seen it in Scratch, Swift Playgrounds and Arduino). Soon they learned the syntax for variables, conditionals, loops and functions.

Repl.it has debugging capabilities, but students did not want to learn to use them after seeing how they worked, for several reasons:

- Considering the size of the code they were working with, mental debugging was realistic and more enriching than using the debugger.
- Students were solving specific problems, but they were also experimenting, meaning students kept commenting out lines and trying different commands to see what happened with each one. Which brings us to the next point:
- Debugging is slow, compared to repeatedly executing the program.

Other concepts we tried to instill are imperative and functional programming, to see which could be easier to conceive, and work with. Thinking in terms of “what should happen to the data”, modifying it through expressions and functions until it has the desired form, against “what should the computer do”, specifying through statements which steps to take and how. Every program incorporates both paradigms, but it was a matter of seeing the dominant one. This lesson was also done in hopes that it would help them visualize when to create a function and when not to, by seeing what they are done for: for example, we saw the implementation of some of python’s built-in functions, like `sum()`, `all()`, `any()`... (figure 19).

### **all**(*iterable*)

Return `True` if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

### **any**(*iterable*)

Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

(Figure 19) An extract of the documentation for the Python programming language that we analyzed.

## Discussion

During the process of learning python's syntax, students did not find it difficult to understand it, but the many ways to do things python offers probably confused them, and they receded back to doing things in the style of C/C++. When using this imperative approach, python's abstract features were of no use. For example, when having to use the range function for going through list indices whenever looping a list over their elements (`for element in list...`) was not enough. However, without comparing the same methodology with another language, it is difficult to see if Python really is too complex when it comes to functionalities. What is more clear, though, is that its minimalistic syntax makes it more readable and easier to write.

The simplicity of the interpreter allowed the students to focus on the code. They did not use any of the functionalities, like the debugger, or the code completion utility. This supports the idea that all that a software for learning to program needs is a quick and straightforward access to the programming environment, to repeatedly execute and tweak programs.

The general progress they made through solving the problems of the competition validates the use of these problems to work on computational thinking simultaneously with learning the programming language.

### 3.3.1. Choosing an interface to evaluate

As we worked on problems from previous competitions and read about the rules and the way we would divide the work, we started finding some requirements for the software that the students should be using. First, according to the rules of the contest, students cannot go online to look up anything. Instead, they can bring any code and documentation they want, as long as it is brought offline. This meant finding a way to put relevant snippets of code that they could use to solve the challenges, classified in a way that they can look for them easily and, once they find them, understand them so that they can modify the code accordingly.

Furthermore, when analyzing the strategy that they would use to work on the different problems, ranging in different difficulty, the students needed a flexible interface that allowed them a quick access to the language to use it as a test bench, as well as the ability to be working in several problems at once, as they would have only one computer for three people, and they may parallelize the solving of problems.

We thought that Jupyter Notebooks could be a way of accomplishing those things, and the reception of the interface had been positive, so we asked the staff of the school to install the software in all the computers of the class.

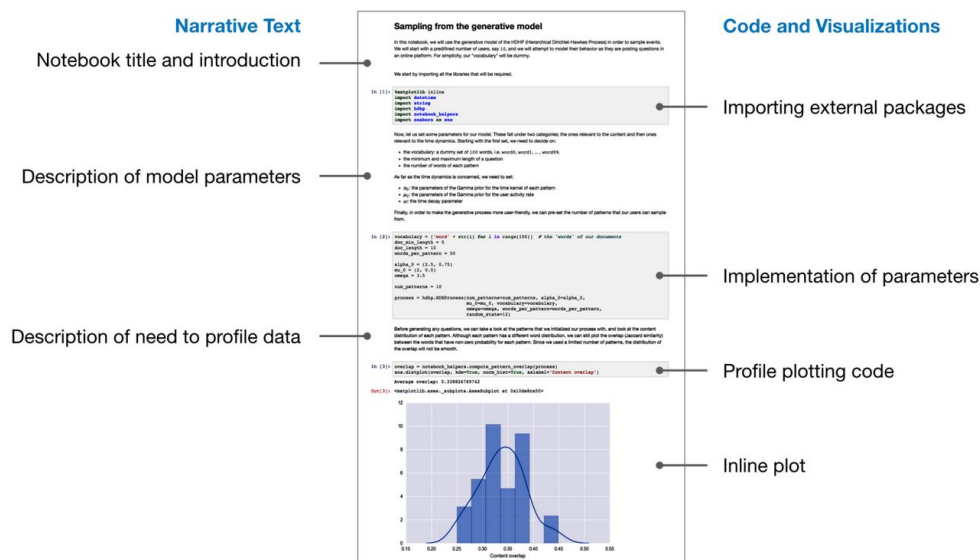
In a meeting with this project's tutor, we validated the idea of introducing Jupyter notebooks in the activity.

# 4. Notebooks and computational narratives

In this section we will grasp the current state of the art regarding computational notebooks such as Jupyter Notebook, which is the software that we have worked with, in two specific activities with their respective reports and posterior discussion.

## 4.1. Literate programming and computational narratives

Computational notebooks are documents that combine fragments of executable code, intertwined with text that explains the thoughts behind the process towards the end result, explaining the code in a sort of narrative. What summarizes this concept is **literate programming** [5], a concept introduced by Donald Knuth, which is the idea that code should follow the logic and flow of the programmers' thoughts, combining source code with natural language. While documentation is extracted from source code, when following the paradigm of literate programming, source code is extracted from documentation [4]. Some well-known pieces of software comply to this idea, such as Wolfram's *Mathematica*, or the *Jupyter* project - the software that we are going to focus on in these research activities. (Figure 20)

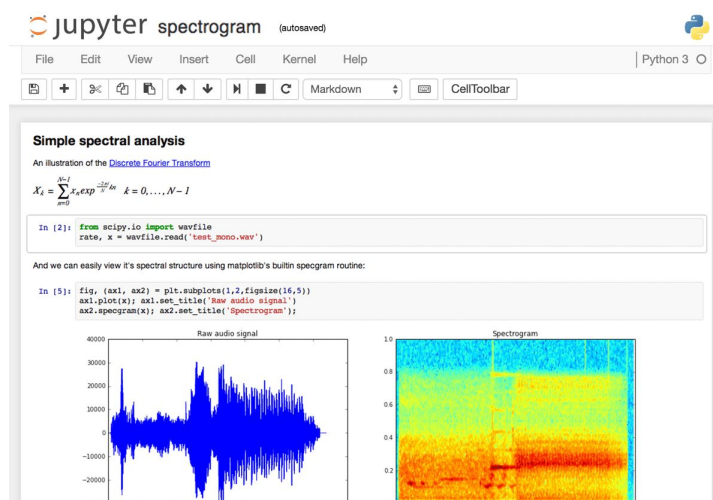




(Figure 20) Elements of a computational notebook in Jupyter's interface

## 4.2. Project Jupyter

Project Jupyter is an open-source project evolved from the IPython initiative in 2014. Its main product is Jupyter Notebook, a web application that allows the user to create documents with interactive code cells (Figure 21), which can be set to over 40 languages, as well as cells narrative text, equations and visual content. The project also features and is supported by tools for running individual notebooks online with no need for installing anything, such as nbviewer or binder, as well as the resources to set up a server which can run a multi-user hub, so that everyone who has access can run code on the same cloud-based kernel, JupyterHub.



(Figure 21) Screenshot of the Jupyter notebook interface, with cells of code, text, and equations.

In Jupyter's interface, the user can append cells of either code (in R, Python or Julia) or Markdown notation. Cells with code can be executed independently, since all three languages are interpreted, and the output is displayed under the cell. The resulting notebook can be saved in the .ipynb format, or it can be exported to .pdf, .html for reading, and .py, in the case of python, for executing [14]. Jupyter notebooks combine code, visualization, and text, making them easy to maintain and share. This methodology is

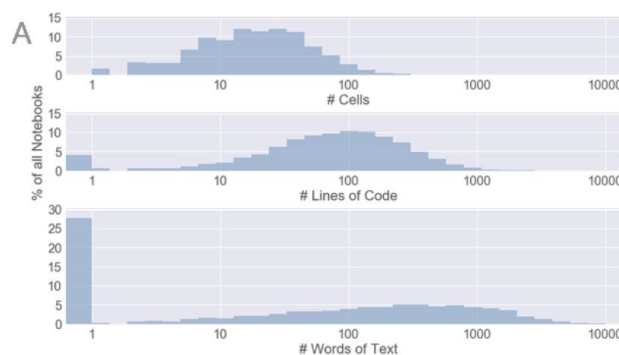
increasing in popularity, and progressively being adopted by researchers and data scientists to elaborate and present their work [23].

Some characteristics that are attributed to Jupyter notebooks are [23][21]:

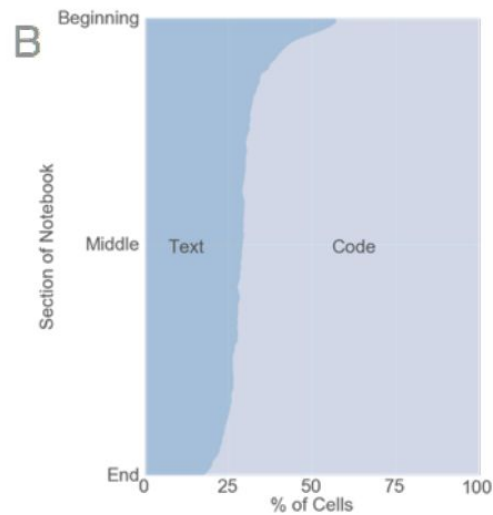
- Easy to share, or be structured as a deliverables, as one file contains all the work: code, documentation and results.
- Promotes modular, atomized code; as it makes it easier to execute it by parts, making tests and benchmarking. This is specially useful in fields like scientific research, data analysis, or machine learning.
- Being a web-based technology, as it runs on a browser and uses HTML5, it features a familiar and simple interface, that allows the user to focus on the content.

### 4.3. Related work

A recent study has analyzed millions of notebooks that were posted on GitHub and found that most of them incorporated little to no text, thus missing the narrative part, something referred to as **explanation**. Some studies confront explanation to **exploration**, which is the process of experimentation involved when working with data. Evidence of a exploratory process leaves a “messy” notebook, which has no purpose of being stored, and so analysts do not bother annotating it [22].



(Figure 22) Histograms for the number of cells, lines of code and words of text of Github notebooks.



(Figure 23) Location of text, and code to text proportion throughout notebooks.

Notebooks made specifically for a divulgatory or educational, like those used as a class presentation or prepared as a deliverable, do not have this problem. In our case, though, students are to work on external resources - the set of problems, and, from there, both work out code from scratch to find the solution to each problem (the equivalent to exploration), and prepare notebooks for further consultation (which would be the explanation).

Besides the explanation to exploration proportion, studies also point at linearity: whether the notebook reflects the thought process of the project in its original order (linear), or if instead it is cleaned and sorted afterwards, recording only important decisions and information out of the actual workflow (non-linear). The world linear is also used to determine the order in which the notebook should be read and executed. Dependencies from preceding cells force a linear execution order, but not all of them follow this approach [21]. Most of these studies agree that notebooks could be redesigned to further encourage and enable users to write more compelling narratives and be more willing to share their notebooks, and supporting non-linear structures that could enable multi-dimensionality.

In Rule's study, many users, which were mainly scientists, reported that notebooks were a development tool for them, and most of the resulting documents were disposable. This concern could be classified as the purpose of notebooks and what is done with them after their use.

Other concerns of research involving notebooks are, for instance, IDE features like Version Control, State Inspection or Debugging. However, we did not focus on these so much, as most of them are not relevant for users who are just learning to code, plus the specific use that they were to give them was not advanced enough. Along with the tutor, we compiled which aspects to validate during the evaluation of the user experience of the students, and carried out two separate activities, one focused on the exploratory use of notebooks, and another one on the explanatory side.

## 4.4. Research activity: using Jupyter as an IDE



### Introduction

During this session (2 hours long), we addressed the use case of solving simple mathematics/programming problems using Jupyter Notebooks with the Python programming language. This research activity has taken place in the Daina Isard school, with a group of 6 adolescents ranging in age from 15 to 16, interested in preparing for and participating in the *HP Codewars* competition, which consists on solving a list of problems programmatically. Their knowledge of programming at the start of the activity was limited, and worked one hour every week between October and January.

The main goal of this activity was to register the students' exploration of programming solutions when using an interface with semi-independent cells of code. Through observation and posterior analysis of the notebooks they left behind, we found evidence of a decent utilization of the interface's features, as well as a positive reception of the introduction of this software.

## Methodology

1. Since it is going to be their first time using the software, it would be worth **watching their behavior while they are setting up the workspace**, and if any doubts or errors come up. In classes, students will be using MacOS, as it is the operating system of the computers in the room that we were assigned, while during the competition they will use Windows.
2. **Observing their use of the interface.** To see whether programming with cells represents an improvement, if they take advantage of them to better atomize the code and making it more scalable or modular, &c.
3. Given that the challenge involves teamwork and sharing the work done, **observing what they do with their work after using the software** could be relevant as well.
4. **Gathering comments on their experience** with an interview, specially comparing them to their previous interface, *repl.it*.
5. Analyzing the resulting notebooks.
  - Counting the number of cells used per problem.
  - Whether they use them to separate different parts of problem solving.
  - If they organized their code with functions.
  - Any other particularity we find.

The problems that they had to solve were the following:

- *Inverting the case of every character of a string*
- *Comparing two lists*
- *Filling a square matrix with booleans, Trues in the diagonal and Falses everywhere else.*
- *Uniquifying a list*

The details of the implementation, input and output were not specified, so that they would focus on finding a correct algorithm which solved the problem. Some problems were missing details on purpose (comparing two

lists according to what?), so that they would experiment and choose the criterion they could implement best. Two of the students worked on their own, while four worked in pairs, sharing the same computer.

After the session, we asked them the following questions:

- Do you think Jupyter is better than repl.it to develop your solution?
- What about presenting your solution? Which format do you prefer?
- Did you find any difficulties when using the cells?
- Why did you (not) save the result? Do you think of it as disposable?

## Results

### 1. Watching their behavior while they are setting up the workspace

All of the students were able to follow the steps previously showed on the screen of the teacher's computer, without looking at any document or having to repeat the walkthrough. This means that the platform is quite easy to access, once installed. In Windows it is even easier, because Jupyter Notebook appears as an executable program and there is no need to open the command prompt.

### 2. Observing their use of the interface

Students used as little as 2 cells and up to 4 cells for every problem. One student and one pair managed to solve the first one and then made improvements and variations to the code, and the other student and pair were able to solve two problems in total.

One of the pairs used markdown cells to put a title to each group of cells according to the problem they solved, despite not being told to do anything in particular with text cells, only their existence.

### 3. Observing what they do with their work after using the software

None of the students saved their notebooks afterwards, but four of them saved their solutions on Google Keep.

### 4. Gathering comments on their experience

*Do you think Jupyter is a better interface to develop your solution compared to repl.it?*

All the students agreed that it is better, as it lets you organize the code and work on several things at the same time, for example, making quick tests. They found the interface more comfortable and quicker than repl.it. One student also mentioned that there was no need for internet connection as a good thing, and another one praised the aesthetics of Jupyter.

Most students, after the first part of the session, decided to give it a try to install it in their personal computers. They said that the process was slow and you had to download many things.

*What about presenting your solution? Which format do you prefer?*

Four students said they would prefer Jupyter for showing a solution to someone else or saving it for themselves, while two others reported that they would rather present it and save it as a plain text file, as it is format-independent.

*Did you find any difficulties when using the cells?*

None of them had found any.

*Why did you (not) save the result? Do you think of it as disposable?*



Students who worked in pairs saved their result in Google Keep, to be able to access it for both of them. Meanwhile, students who worked alone did not think of saving their code, and if they had to they would probably have saved it in some format that can be opened from anywhere.

5. In the following table, with the notebooks that they left, we gathered the following data, where S1 to S6 are the students, some in pairs (Table 2):

	S1	S2	S3	S4	S5	S6
Number of cells	3		2 + 1		3	3+4
Used individual cells to test individual operations (functions, operators)	Yes		No		Yes	Yes
Used separate cells for input	No		1 + 0		Yes	Yes
Used separate cells for output	Yes		1 + 0		Yes	Yes
Auxiliary functions	0		1		1	0
Number of problems solved	1		2		1	2

(Table 2)

## Conclusions

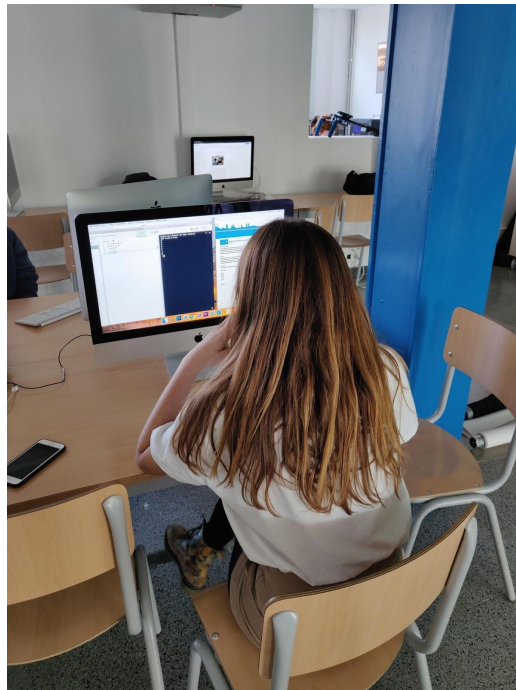
The first thing to highlight from this activity is the fact that, unexpectedly, students that had a powerful and personal enough computer at home tried to install the software for their own use, after doing the first part of the session. In some universities, they set up a cloud-based kernel so that students all work with the same version of the language and libraries, and they access it, for example, using *binder*. This could be a good solution for students who don't have powerful enough computers to install the whole python environment.

The overall feeling about notebooks for supporting code exploration is positive, both in terms of user experience and effectiveness. For being a first contact, and being used to a simpler interface, they all have put the cell system to us, with positive results.

At the same time, though, students did not make an effort in leaving a presentable notebook and saving it in the .ipynb format. It is true that they had no instructions, and the activity was posed as an exercise for practice.

When analyzing the resulting notebooks, we find that most students did not separate one same algorithm in different cells, except in two cases, where they created an auxiliary function in a different cell, and another one who separated different statements and expressions. The student who did the last one was following the declarative/functional paradigm, where data was modified through expressions and functions, which may have encouraged the more intensive use of cells. We could do another session to deepen on this correlation of notebooks with functional programming.

## 4.5. Research activity: using Jupyter to make programming notes



### Introduction

During this session we have addressed the use case of writing down useful functions about specific topics in mathematics and computer science using Jupyter Notebooks with the Python programming language. This research activity has taken place in the Daina Isard school, working individually with students from a group of 6 adolescents with ages ranging 15 to 16, interested in preparing for and participating in the *HP Codewars* competition. They had the task to craft a notebook covering some subject for future use during the competition. Our supervision during their activity was mainly to remind them of the purpose of the notebook: that they will not be the only ones consulting it, and that it is better not to take knowledge for granted. Through posterior evaluation of the notebooks, we did not find straightforward evidence of students being interested in documenting their code using cells, but we did find some other requirements

## Methodology

This activity was done individually, in a pair programming setup, to a total of 2 students, and for each student we followed the following steps:

1. **Choosing a suitable subject** for the student's knowledge and interest.
2. **Preparing a corpus with information and requirements** for the student to fill, in the order they think is optimal. This could contain definitions, pictures, and snippets of code, as well as links to documentation pages and encyclopedic articles.
3. **Observing their use of the interface** while crafting the notebooks. How many cells of text for every cell of code? Do they use comments as well? How deep is the text they write? How is the document structured in terms of dimensions and linearity?
4. **Gathering comments on their experience** with a conversation on the format they would rather use for these documents, and their overall assessment of the interface.

## Results

### Student 1

(1) We decided to make a notebook with practical list operations. It may not be the most relevant subject, since most of this information is present in the documentation, but we thought of it as a summary. Making a more complex subject would have supposed too much intervention from the supervisor.

(2) For this session, we decided to cover the creation of lists (empty list or list with fixed size), access, assigning, removing (by element and by index) and finding (checking for existence and finding the index).

(3) The student used text cells as a title for the cell below. They did not explain the nature of lists in comparison with other data structures or what they can be used for. Instead, they created variables with neutral names. They also followed a narrative structure, making cells below dependent to

having run cells above. The problems they encountered is when trying to follow a non-linear path: most titles reflected variants of the same problem, and they also wanted to have different ways to solve the same problem represented, but not all having the same 'importance'. This is when they explained that when taking notes, they copy everything and highlight what they think is important.

Their notebook ended with the following characteristics (Table 4):

Linear	No
Linear execution	Yes
<b>Total code cells</b>	8 (+ 3 discarded but not deleted)
Code cells avg length (bytes)	40
<b>Total text cells</b>	5 (+ 1 to indicate discarded cells)
Text cells avg length	8.2 words
Text cells purpose	Informing of the operation that the following code cell executes or the problem that it solves

(Table 4)

Their process writing the document encouraged exploration, since they had some basic atomized problems and they had to think of the best way to showcase their solution. They often changed the order of cells, and quickly got used with the interface's up and down arrows.

(4) When the notebook was finished, we discussed on which format they would use during the competition: ipynb, pdf, html. They did not know, so we tried both options and html worked better for them, because it preserves text format so that they can copy, and leaves code for the notebook they will be using to develop their solution.

The student, who has a beginner's guide on python printed out and brings it to class, reported that this would be better than consulting the documentation, because it is more straightforward and practical.

## Student 2

(1) Going through some problems, we detected the need for a notebook about input methods: obtaining individual values, groups of them, definite and indefinite successions and matrices, from the user's input.

(2) We made a progression of cases in increasing complexity, starting from obtaining a string, to casting it to several types, including integer, float, boolean, and finally a finite iterable: tuple. Then they had to explain how to obtain a succession of elements (of any nature), filling a list or some other data structure, covering the case of asking for input for a previously given number of times, or until some input was entered.

(3) This student also encountered the problem of not having a specific situation to model, in order to decide the variables' names, the protocol expected for the input, or how they would process it. Their solution was to make every cell independent from previous and posterior cells, as well as leaving comments where extra code should be inserted in order to customize the algorithm.

These are the values extracted from the resulting notebook (Table 5):

Linear	No
Linear execution	No
<b>Total code cells</b>	7
Code cells avg length (bytes)	160

<b>Total text cells</b>	7
Text cells avg length	11.1 words
Text cells purpose	Informing of the purpose of the algorithm or expression shown below

(Table 5)

The student also used comments in code cells to go into the specifics of the language syntax. They managed to cover all cases of input for the competition, in an increasing order of complexity and multiplicity, thus giving the document a logical order, which was different from their exploration - they changed order of cells once.

(4) In the posterior discussion, they were shown the different alternatives for saving the notebook, and they reported HTML is the best option for the competition.

Talking about the use of the interface that we suggested, they gave a positive feedback, describing it as “useful”, and “versatile”, and an effective way of recording useful snippets for the competition.

## Discussion

Despite putting the students in context and telling them the uses of what they were going to build, they seemed to prefer letting code speak for itself, using in-line comments, and only using text cells to entitle what is in the cell below. In the text, they could have talked about which situations could demand that specific code to be used, or the computational thought process behind it. Instead, they just used it as sort of a cheat sheet. Several reasons could account for this: the lack of deep knowledge, confidence, and practice documenting code may reflect in short, safe text. Besides, the more simple subjects covered may make it unnecessary to explain thoroughly what is going on in the code. The activity they had to carry out is not conceptually different from one they already do on the daily: taking notes from the topics

covered in their classes, or making a diagram summarizing them. It is the first time that they have to do this with code though, and Jupyter may have helped in it, in contrast with the only note-taking they had done before: in-line comments, and saving relevant pieces of code in Google Keep notes. This application is more general-purpose, but it is available in all the platforms these students use, is easy to access, use, and it allows them to share it amongst themselves. Jupyter lacks these features, and it would be relevant testing to which extent the richer explanatory potential makes up for the less accessible and shareable format. Once the activities are put in a favorable context, like ours, in which they will be working offline with a computer that will have the python environment installed, Jupyter represents a clearly superior solution. In other cases, like the context of a class, they could have a server-based solution that allowed them quick access to cloud-based notebooks, and the shareability and accessibility concern would disappear.

Explanation was poor in both cases, but we can look at other aspects of the notebook such as dimensionality. Both students were covering different problems, and one found that some things could be done in different ways, something that is bound to happen when creating educational content, as it has to dive into the specifics of how a problem is solved, something which would not happen in a data science dissertation (one would not expect a data scientist to code two different cells showing two ways of accomplishing the same thing). Meanwhile, the other student struggled in making a general enough algorithm that does not adhere to one specific problem, but can be easily changed. Maybe, in a bi-dimensional approach, they would have made several examples that showed how each algorithm can be applied to different cases.

Concerning linearity, defined as leaving the exploration process unchanged in order, this activity was bound to create non-linear results, as it was designed to make a document that was optimized for lookup and learning. When it comes to navigating the notebook, one of the students made it necessary to follow its order to a certain point, whereas the other made the

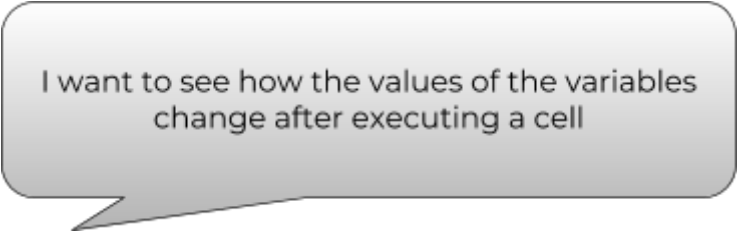


cells linear on purpose, sacrificing the connection they could possibly have between them, such as showing how one same input/result is transformed differently by different statements or expressions.

## 5. Software design

### 5.1. Requirements

The first step of the design process is to decide what do we want the software to accomplish: its requirements. In our case, instead of redesigning the Jupyter interface from the ground up, we opted for thinking of ways for extending it. These new features or extensions are based in the results of the investigation we have carried out, and their purpose has to fit the profile of the users we have worked with, as well as their context, in hopes that these same improvements are also useful in similar situations. From the reports of the activities, we can extract the following relevant thoughts, and put them in the form of a requirement:

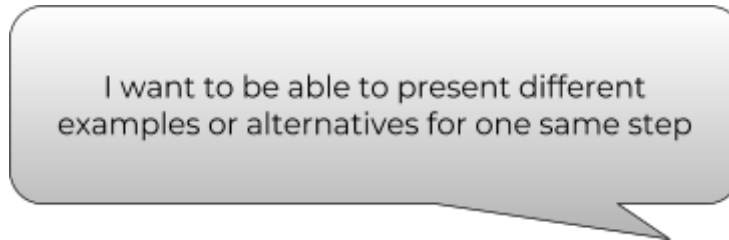


I want to see how the values of the variables  
change after executing a cell

(Requirement 1)

This functionality (Requirement 1), when making a notebook that showcases different mutations you can perform to a specific piece of data, will decrease the probability to pick a wrong execution order, and will make the user conscious of how their execution path is affecting the final output. Knowing this, students can design notebooks with “wider walls”. A real-life example of this could be in the case of the first activity. The student showed how to create an empty list, and a list with hardcoded elements, and then used the latter to see how it can be modified - with deletions, value changes, or insertions; and which expressions could be called on it: accessing, searching, slicing... If they could keep track of the state of the list, the students who navigated through that notebook could predict how each operation is going

to affect it, and how it ends up doing it, without adding print statements before and after each line



(Requirement 2)

As we commented in the discussion section for the second research activity, this would mean adding a second dimension to the now one-dimensional narrative structure of the notebooks (Requirement 2). This would not mean enabling a tree-like structure, but a multi-layered one. For one same cell, there would be more than one alternative, accomplishing the same task.

## 5.2. Co-designing activity for bi-dimensional notebooks

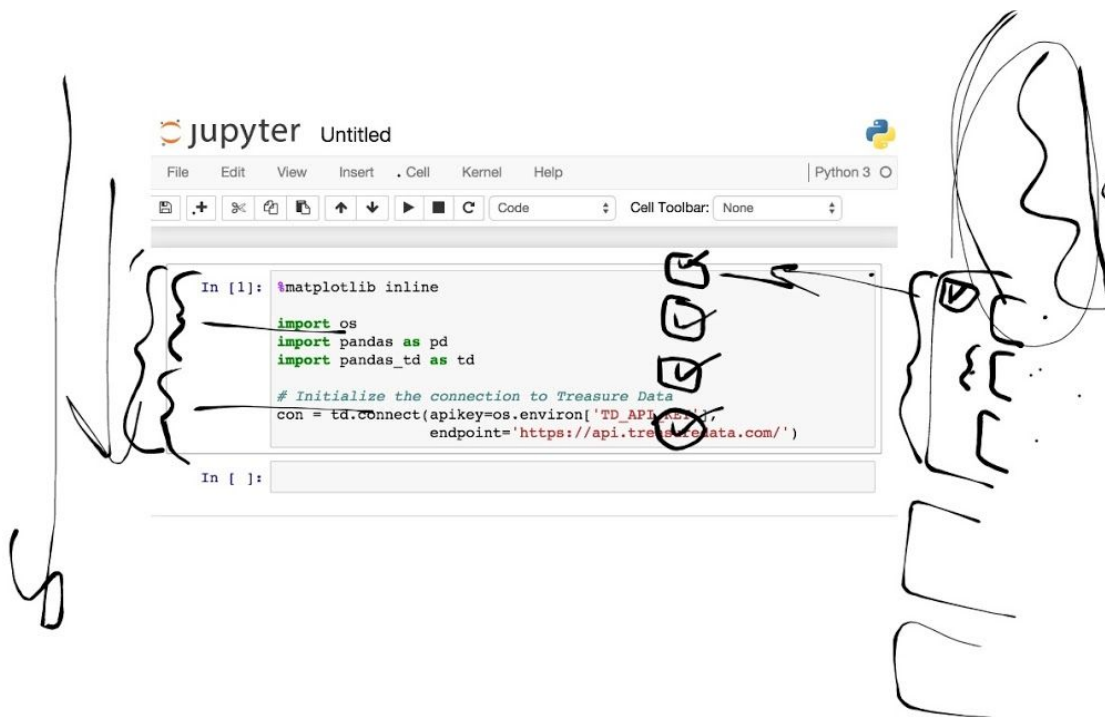
In order to meet the needs of the students when designing the extension for the Jupyter interface that would fulfill requirement 2, we took the following approach:

### Methodology

Using an interactive screen, we put an image with a fragment of Jupyter's interface, on top of which the students could draw and explain their ideas for designing the feature that would allow them to make more than one cell for one same level, concept, to present different alternatives or examples, and so on. We did this with a total of three students, and they did it separately, not to get ideas from the previous participants.

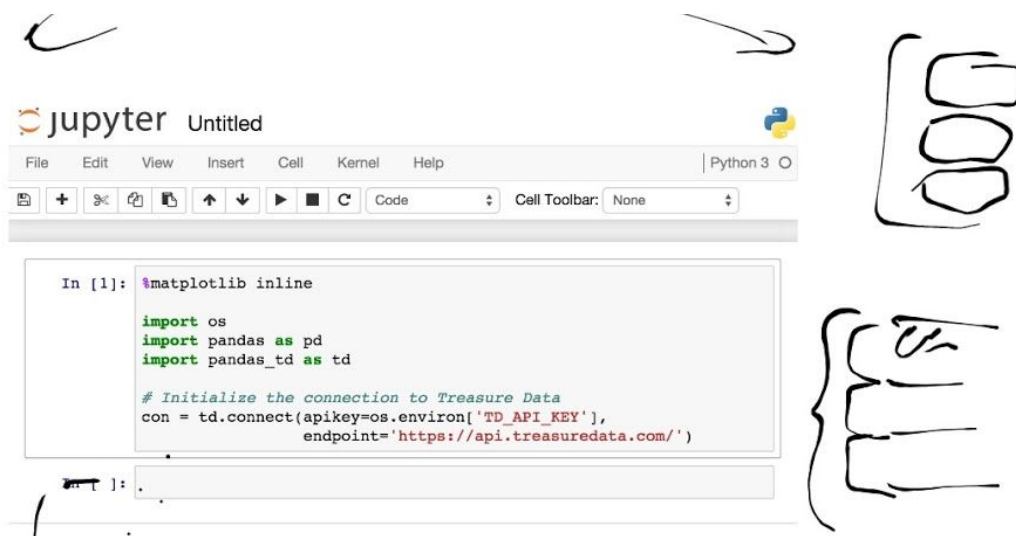
### Results

The results were the following:



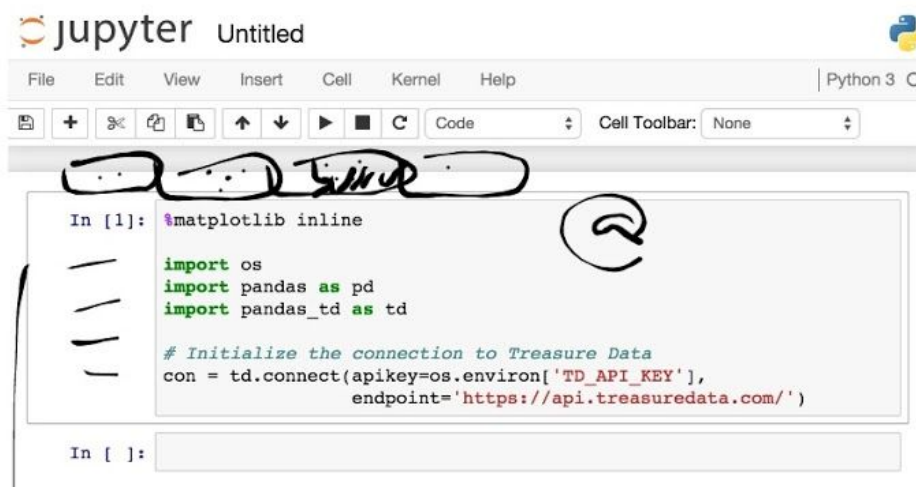
(Figure 24) The first student's drawing.

In this design, the user would be able to group cells in “supercells”, the behavior of which could be specified. The user, when executing, would be able to choose executing one or more sub-cells, or executing the super-cell.



(Figure 25) The second student's drawing.

In this design, the student suggested folding cells, with a title for when they were minimized. This way, you could group them, and unfold them all at once, thus making it visible that they belong to the same concept, making every group or non-grouped cell a unit of the underlying narrative.



(Figure 25) The third student's drawing.

In this approach, the single narrative would be broken into multiple narratives, selected from the general view with tags. This way, tags could be combined to make some cells appear in several of these sub-narratives. When visualizing alternative cells, users would see them all at once, but they would be seeing less cells in total because the rest would be filtered. This means that navigation also has a second component. The student drew a horizontal menu for tags, to click on them and filter the cells below.

## **Conclusions**

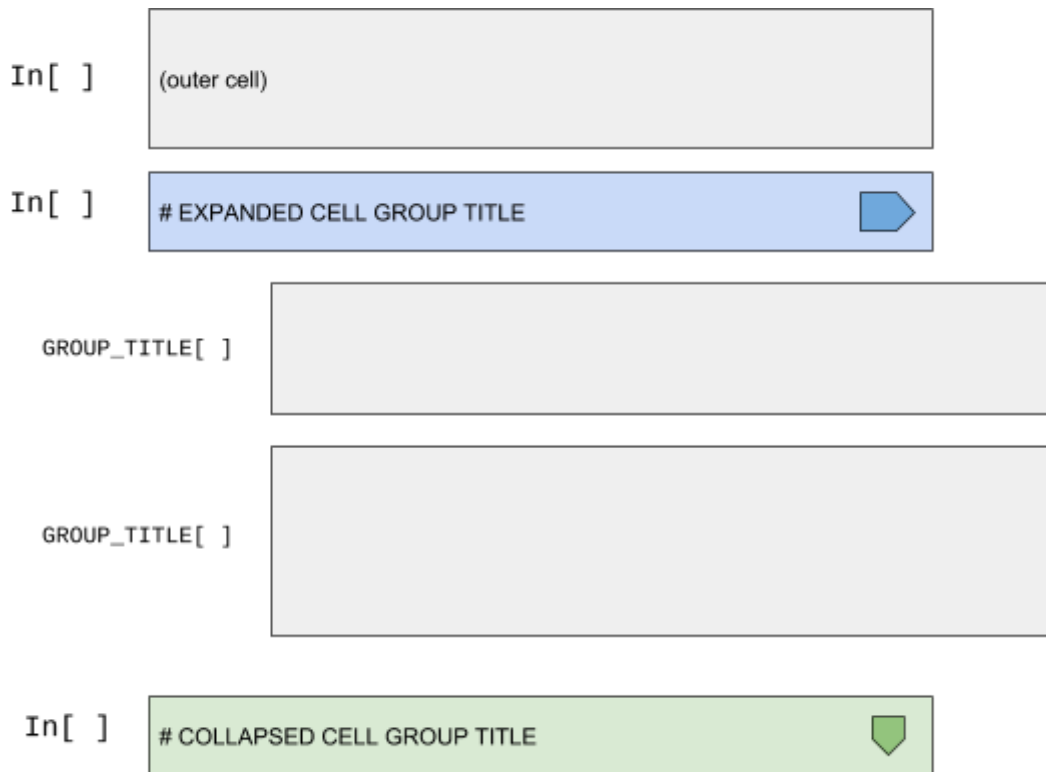
The three students' solutions had these traits in common:

- Alternative cells were always seen together, not hidden and excluding each other.
- All of them thought of some kind of grouping of similar cells, which the user would have to specify. That is, to link similar cells to a group.

Here is our assessment of each idea:

1. Making executable groups of cells could be helpful to guide exploration and in some specific cases, but to the point that it seems oddly specific and unrealistic.
2. Cell folding is useful to economize vertical space, and grouping cells can be visually helping. The only thing this would lack is some way of visualizing that execution of cells is
3. This more general feature, which would change the current vertical-only structure of notebooks, can be used to make alternative narratives with an intelligent use of tags. It takes a bit more work to design these paths, tagging all cells, but it is an interesting path to explore.

After this activity, rather than adding a horizontal slider for alternative cells, we see that students would rather group them according to categories, and make them collapsible. This way, the top to bottom narrative is preserved and cells can be in an outer or inner level (Figure 24).



(Figure 24) Our proposed solution.

### 5.3. Prototyping and relevant art for variable inspection

Designing a GUI for keeping track of variable values is a challenge that most IDE developers face. It is typically integrated in a debugging system, where the value of variables is updated after each step of the debugging process. The most widely spread concept displays the variable's name, its type and its value in a list. The programmer has to explicitly mark the variables they want to follow (Figure 25).



Variable	Type	Value
▼ Parameters		
argc	int	1
argv	char **	0x7ffeb8...
▼ Locals		

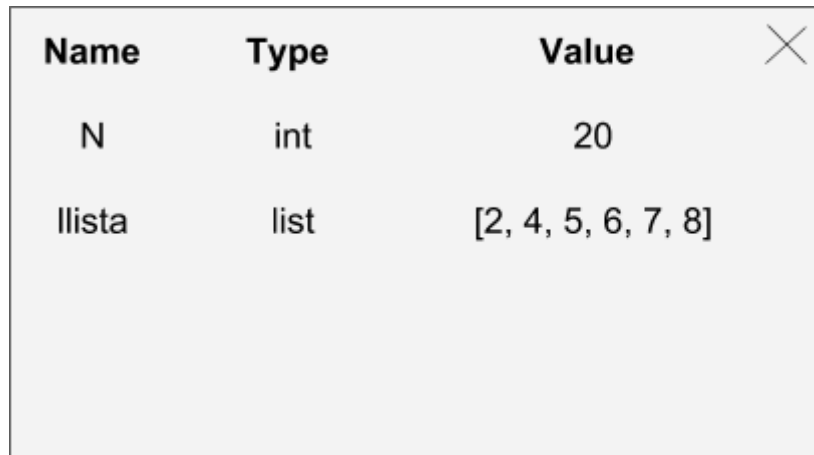
(Figure 25) The variable inspector in Gnome's Builder

One approach could be to automatically print all the variables involved in a cell's code, if they have changed. However, according to what we have seen, this has some disadvantages:

- It keeps variable evaluation in the last executed cell, concentrating all the attention there, and discouraging students to navigate the notebook freely if they want to see the value they are about to change and how it changes.
- As more variables are involved, running code would take up more space.

A better option would be to automatically add the variables as they are added to the environment, in a separated container which floats above the main page (Figure 26).





Name	Type	Value
N	int	20
llista	list	[2, 4, 5, 6, 7, 8]

(Figure 26) An example of the floating container embedded in Jupyter's interface

## Implementation

These features could be implemented as extensions of Jupyter. In fact, they already exist unofficially, under the names “variable inspector” and “collapsible headings”. According to their documentation<sup>1</sup>, the Jupyter team warns the users of front-end extensions, as “the notebook front-end and Javascript API are not stable (...) any extension written for the current notebook is almost guaranteed to break in the next release”.

---

<sup>1</sup> <https://jupyter-notebook.readthedocs.io/en/stable/extending/>

## 6. Conclusions and future work

At the end of the project, taking into account the current guidelines from the Departament d'Ensenyament and the school's objectives, we can say that implementation of computer science learning in high school is viable, and the general reception of the activities introduced is positive, so developing software for supporting learning of computer science for the sake of itself is an area to explore from the perspective of UI design, and achieving code literacy.

From the experience of the activities we can extract that Jupyter notebooks, and thus similar computational notebooks, are a good way of developing, editing, and presenting code for learning purposes, from adolescents' perspective, at least. The ease with which the students used the interface shows that it could be applied to other subjects, like mathematics or physics, allowing for a mix between code and explanation that is more straightforward and clear than just code. In this sense, we would recommend the school the use of this interface to implement all sorts of projects of different nature, one of which could be computer science, or other fields which do not require external peripherals like robotics do.

When it comes to more specific aspects of the use of the notebook interface, it seems that explanation of code is not a priority for students, who prefer code to speak for itself. In fact, they were more interested in enriching their cell execution experience or narrative, by showing alternative code cells for a same concept, or seeing how variables change along the execution. This renders notebooks good for filling up, for example, as deliverables; or for teachers to use as notes or slides for the students. Jupyter now allows the creation of slides from a notebook, to use as presentation, keeping the code and explanation structure. They have also been proved to be a nice tool for developing scripts to solve problems, but if the students' approach is more imperative than functional, students will end up using it as an interpreter of

source code, so this duality is relevant when further studying these interfaces.

Documentation is directly linked to reusability and shareability, which are also concerns of investigators regarding notebooks. In this aspect, we have seen that making notebooks in hopes of showing others your work, reusing their code, and reading the documentation of it, can indeed be positive, but it is a good practice which does not come up naturally when novice programmers use the interface. This project has ended before the competition, so we can not know yet if the notebooks they (and we) prepare will be really effective for the competition, but simulated tests in class have proven them to be very useful.

In the design and co-design activities, the investigations' requirements and subsequent features both already exist as extensions of the Jupyter interface, so, in a future iteration of the experience, these could be installed along with Jupyter to see whether the students actually use them and if it helps them code better or have a better user experience.

## 7. References

- [1] Aho, Al., 1994. *Foundations of Computer Science*.
- [2] Beth Gardiner (2014) 'Adding Coding to the Curriculum', *The New York Times*, March 2014. Visit at [this link](#).
- [3] Casado, C. (2017) 'Aprendre a programar a l'escola trenca barreres socials i de gènere', *Universitat Oberta de Catalunya news portal*. Visit at [this link](#).
- [4] Dominus, M., 2000. 'POD is not Literate Programming'. *Perl.com blog*. March 2000. Visit at [this link](#).
- [5] Donald Knuth. 1984. *Literate programming*. *The Computer Journal*, 27, 2 (Feb. 1984)
- [6] Farmer, J., 2014. 'Teaching Novice Programmers How to Debug Their Code'. *Code Union blog*. September 2014. Visit at [this link](#).
- [7] <http://ensenyament.gencat.cat/.../funcions-atribucions/>. Departament d'Ensenyament de la Generalitat de Catalunya.
- [8] <https://awards.acm.org/software-system/award-winners>
- [9] <https://scratch.mit.edu/discuss/topic/245/>. Scratch forum, discussion.
- [10] <http://www.exploringcs.org/archives/resources/cs-statistics>
- [11] <http://www.hpcodewarsbcn.com/>. HP Code wars Barcelona website.
- [12] <http://xtec.gencat.cat/ca/curriculum/eso/curriculum/> - Àmbit Digital - Àmbits Curriculars. Xarxa Telemàtica Educativa de Catalunya.

[13] <http://xtec.gencat.cat/ca/recursos/tecnologies/programacio/>.

Xarxa Telemàtica Educativa de Catalunya.

[14] Jupyter team. Jupyter Quick Start Guide. Visit at [this link](#).

[15] Kafai, Yasmin B., 2014. *Connected Code - Why Children Need to Learn Programming*. Cambridge, Massachusetts: The MIT Press.

[16] Lewis, S. (2014) 'Should Everybody Learn to Code?', *Communications of the ACM*, February 2014

[17] Lewis, S. (2014) 'Should Everybody Learn to Code?', *Communications of the ACM*, February 2014, chapter 2.

[18] Lewis, S. (2014) 'Should Everybody Learn to Code?', *Communications of the ACM*, February 2014, chapter 8.

[19] Manderson (2016). 'Low floor, high ceiling, wide walls in ELA classrooms'. *Schools & Ecosystems*, December 2016. Visit at [this link](#).

[20] 'Més de 1.500 alumnes adquireixen bases de programació, electrònica i mecànica a l'aula', *Ara.cat*, October 2014. Visit at [this link](#).

[21] Rule, A. (2018). *Design and Use of Computational Notebooks*. UC San Diego.

[22] Rule, A. (2018). *Exploration and Explanation in Computational Notebooks*. *ACM CHI Conference on Human Factors in Computing Systems*, April 2018.

[23] Shen, H., 2014. *Interactive Notebooks: Sharing the Code*. *Nature*. November 2014.

[24] *Stajano, F., 1999. Python in Education: Raising a Generation of Native Speakers. AT&T Laboratories Cambridge.*

[25] *Vee, A., 2017. Coding Literacy. Cambridge, Massachusetts: The MIT Press.*

[26] *Wing, Jeanette M., 2014. 'Computational Thinking Benefits Society', Social Issues in Computing, January 2014. Visit at [this link](#).*