SENG402 FINAL REPORT

BitFit exercises (Python)

Jessica Robertson

12430878

25/09/18

Abstract

Programming has recently been introduced into the New Zealand school curriculum. However, many teachers are not confident in their programming ability, which can have a negative impact on how well they can teach their students. The goal of this project was to develop a website where secondary school teachers can practice Python. The website assumes a basic knowledge of Python, because the main purpose of the website is to help the user keep up their programming 'fitness', not to teach them Python from the beginning. Motivational aspects have been incorporated in order to encourage the user to practice regularly, with the ability for the user to set their own goals. Three different question types are available: regular programming exercises, Parsons problems (ordering blocks of code), and debugging questions (finding an input for which the given program behaves in an unintended way). The website was developed using the Django web framework and has been successful so far, with potential to expand in the future.

Table of Contents

1 Introduction

With computer science having recently been introduced into the New Zealand curriculum, more teachers are required to be proficient at programming [1]. In a survey of over 300 teachers, a study found that the teachers' most commonly-mentioned challenge was lack of confidence in their subject knowledge [2]. A reason for this could be that maintaining skill in any language requires practice, yet there are often long breaks between times that a teacher needs to use their skills.

For the EDEM665 course at the University of Canterbury, the current solution to this problem are 'BitFit' exercises. These are a set of Python practice questions which students are encouraged to attempt regularly. The BitFit system is based on CodeRunner quizzes (from COSC121) and is built on top of Moodle – the same system used for Learn.

There are two main problems with the current system. The first is its lack of flexibility, which makes adding motivational features, monitoring, and feedback difficult. The second is its visual clunkiness and similarity to what teachers are familiar with giving their students; it feels like a site designed for students rather than for teachers.

The goal of this project was to develop a system which is more user-friendly and more motivational than the BitFit system currently being used. This will hopefully encourage teachers to practice programming more regularly, which will help them maintain their skills so that they are more capable of helping and inspiring their students. Ideally this system will not only be used in the EDEM665 course but will also spread to be used by working teachers across New Zealand.

2 Research

2.1 Requirements

Initially, this project was intended to be a joint project with Matthew Jensen, another SENG402 student. While we have both aimed towards roughly the same goal of improving BitFit, his focus was on primary school teachers while my focus was on secondary school teachers. The websites may eventually be combined, but during development our projects have been almost entirely independent.

As mentioned above, New Zealand secondary school teachers who teach programming are the target audience. There are a number of different languages taught at secondary school level, but it was decided that this project would only focus on Python based on the language's popularity and the potential complexity involved with supporting multiple languages.

The core requirements of the project are to enable teachers to practice questions in Python, to motivate them to practice regularly, and to provide useful feedback on their skills and progress. There are many ways in which this could be accomplished. In order to help clarify and narrow down the requirements, a list of user stories was written

(given in Appendix 1). This list was reviewed by Jack Morgan, an important stakeholder in the project because of his involvement in computer science education research. The list was then broken down into a set of technical requirements which were organised into categories and put in a rough priority order (Appendix 2). Both lists were further reviewed by Tim Bell, the project supervisor and key stakeholder.

On April 18, I presented some of these ideas for the project to a group of teachers from the EDEM665 course. They provided some useful feedback with regards to social aspects of the site which I was considering. For example, some of them said they would be interested in sharing their code and/or progress with others but would like the option to be anonymous. They also seemed interested in the proposed use of an intelligent tutoring system to give them hints and suggest suitable problems to attempt.

2.2 Alternative Solutions

A number of alternative solutions were researched prior to development. One of the simplest possibilities would have been to recommend that teachers practice their programming on an existing website. A list of sites which could fit the criteria was made (Appendix 3). However, most of these websites had a focus on learning programming rather than practicing it, and none were aimed specifically at teachers. Recommending an external website would also remove control over questions from the EDEM665 course administrators.

A similar option was to fork an open source repository – either a programming website or a website which aimed to motivate people to maintain their physical fitness. A fitness website could have been modified so that it encouraged practicing programming instead. However it would have taken a significant amount of time to learn the architecture and make modifications which worked with the existing structure. Doing this was deemed not worth the advantages which reusing some aspects of an existing site could bring.

Another alternative was to build off the existing BitFit system using Moodle's framework. This solution was at first promising, due to there being experts around at the university who were familiar with Moodle systems. One of these is Richard Lobb, who built the CodeRunner system which the computer science quiz servers use [3]. If Moodle was used, then the system to run questions would already be implemented, along with Moodle's framework (which provides user models and a badge system).

However, one of the concerns with Moodle was that it looks similar to many sites used in schools – which teachers may not like as it could make them feel like students rather than teachers. Several themes were found which would allow the site to look unique enough to avoid this problem, such as Adaptable [4]. United for Wildlife [5] is an example of a website which uses Moodle but does not look as though it is using Moodle. However, United for Wildlife has a large number of developers (at least compared to this project). Therefore it is unlikely that a high level of modification of the default aesthetics would be achievable for this project. In addition, several people I talked to

mentioned that Moodle had a steep learning curve and was not very flexible without deep knowledge of the system.

By contrast, the final option to build a website from scratch would allow much more flexibility. It was estimated to take more time to implement the basic requirements but offered greater potential for implementing features such as the intelligent tutoring system. It could also be more easily understood and maintained as there would be no unnecessary parts. Node.js, Express.js, and Vue.js was the first stack considered because I have experience with them from the course SENG365. The second option was Django – a Python web framework which Jack Morgan uses in his work. This meant that he would be able to answer questions about Django and provide advice on which packages might be useful. It also meant that it would be easier for the project to be continued after the end of the year. Another key advantage of Django is that it comes with a pre-built admin interface, which allows more time to be spent on development of features for regular users.

It was hoped that CodeRunner could still be used, but discussion with Richard Lobb determined that CodeRunner is tightly coupled to Moodle and could not easily be adapted. However, he did suggest that the JOBE server (which CodeRunner uses in order to run programming jobs) would still be useful in contexts outside of CodeRunner and Moodle. JOBE supports multiple languages including Python, C, C++, PHP, and Java, which could prove useful if the project needs to support more programming languages in future [6].

At the current stage, the project is using SphereEngine to run and test the user's programs. Like JOBE, SphereEngine also supports multiple languages. It has two main APIs: the Compilers API, and the Problems API [7]. The Compilers API is simpler than the Problems API, its purpose being to compile and run programs sent to it. The Problems API, on the other hand, provides a system for managing questions, including judging and storing of test cases. The Compilers API was chosen because using the Problems API would make it more difficult to move away from SphereEngine in the future. This was considered likely due to the pricing and the potential of the project to have a significant number of users if successful. In fact, the product owners (Tim Bell and Jack Morgan) have recently decided not to renew the subscription to SphereEngine, which ends in mid-October. Some of the most likely alternatives include the JOBE server (maintained by Richard Lobb) or Skulpt (a browser-based IDE which is being used for Matthew Jensen's project). In order to make the switch, several client and server side functions will need to be edited, to a lesser or greater extent depending on how similar the chosen solution's flow is to SphereEngine's. However the template for judging test cases was designed not to be dependent on SphereEngine.

3 Development

3.1 Technologies

The back-end of the application was developed using Django, which also provides templates for the front-end. The front-end is written in HTML, CSS (with Bootstrap), and JavaScript (with JQuery). I had planned to explore using Vue.js for the front-end in order to make the website more reactive and easier to maintain, but did not end up taking the time to do so as it was unclear whether using Vue instead of JQuery would actually provide value to the project in terms of ease of development.

Git was used for version control. The repository I worked on was a fork of an empty repository created on GitHub by University of Canterbury Computer Science Education Research, so that they (or another student) can continue work on the project after the end of the year.

An issue with using GitHub is that the repository is public but there exists sensitive information such as API tokens which need to remain private but still be used within the application. Originally the solution to this was to create a table for tokens in the database. However this did not work for running tests reliant upon having a valid API token, as the test database is separate from the development database. Therefore the tokens were stored in a file outside of the repository.

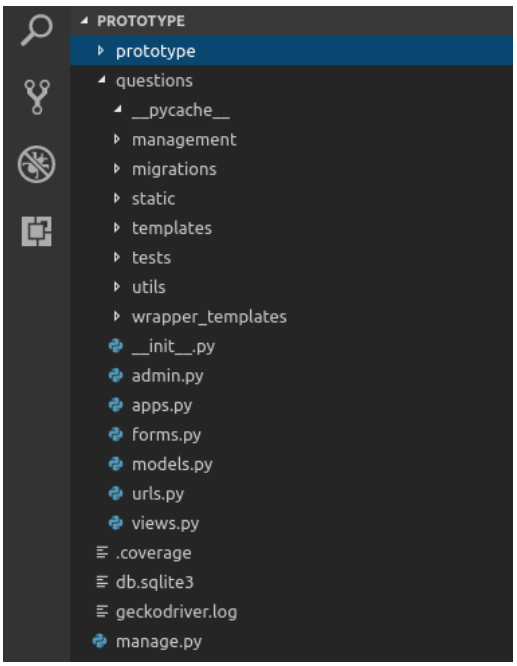


Figure 1: High level structure of the questions app.

I have endeavoured to structure my codebase according to the guidelines in Django's documentation. The management folder stores commands such as populate.py, which can be used to populate the database (although this file has not been properly used yet). The migrations folder stores database migrations, which are generated automatically but which can be edited. The static folder stores the favicon and all CSS and JavaScript files. A separate subfolder called vendor is used to separate third-party libraries. The

templates folder stores HTML templates, the tests folder stores tests, and utils is currently empty. The last folder is wrapper_templates, which stores Python templates (for judging code). The admin.py file is where the default admin interface can be modified. forms.py stores all forms, such as the signup form and forms for editing test cases in the admin interface. models.py holds the definitions for all database tables, urls.py specifies valid routes and which view each links to, while views.py stores all views (server-side classes or functions which handle web requests).

3.2 Database

The database used for development was SQLite. This was chosen based on its simplicity and compactness, and because it works well for websites with low to medium traffic [8]. If it is decided at some point in the future that the application should instead use a client/server database (such as PostgreSQL), then the switch should be a simple process as Django's database layer handles all SQL details so the only change should be in the settings.py file [9].

My original design for the database is shown below. Green are tables handled by Django, while purple and blue were implemented by me (purple had been finished before the progress report and blue was done after the progress report). In the below diagram, question type refers to whether the question is a program or a function. It was assumed when I created this design that all questions would be programming questions – Parsons problems and debugging questions did not yet exist.

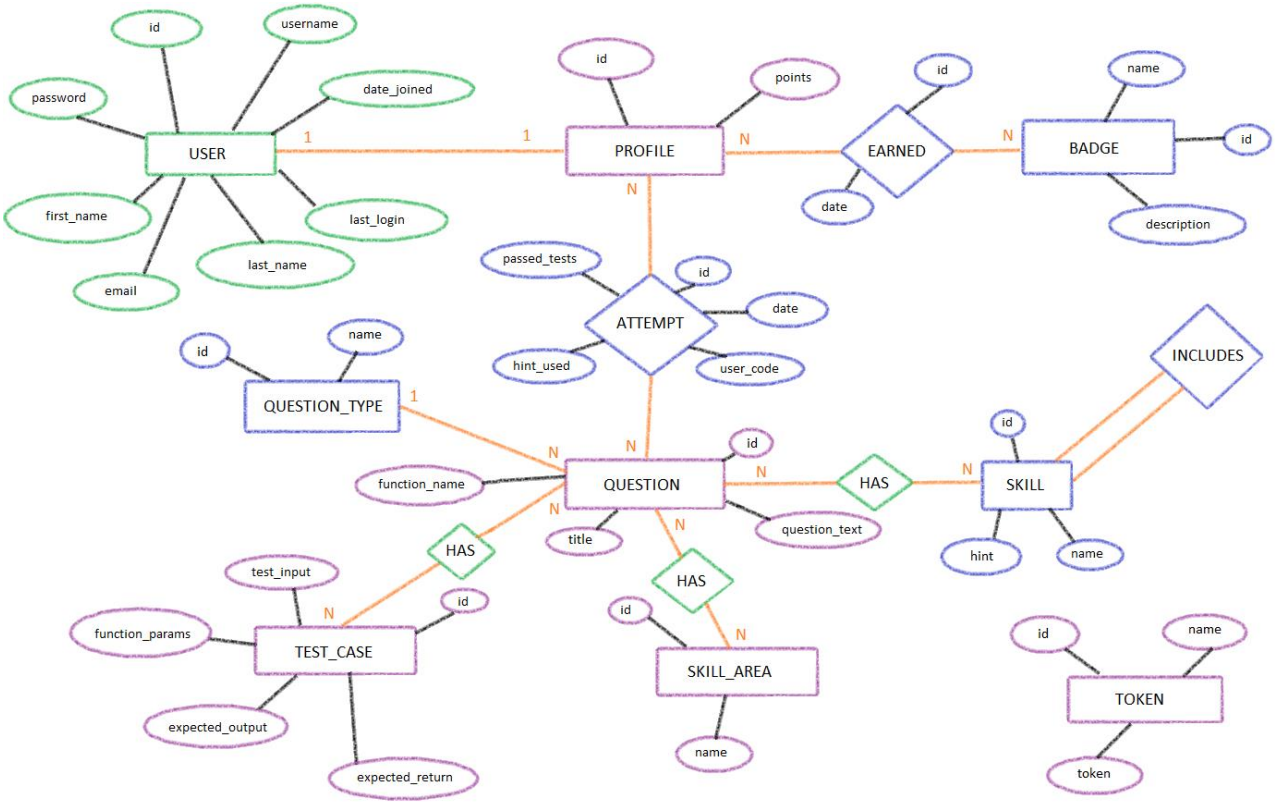


Figure 2: EER diagram of initial database design.

As more question types were added it became clear that the question table ought to be split into different tables with relevant fields for each question type. Some fields were shared between all question types, so Django's model inheritance was used (which works in a similar way to class inheritance in Python). The two options for inheritance were abstract base classes or multi-table inheritance (not to be confused with multiple inheritance). The key difference between these types of inheritance is that with abstract base classes the parent class does not have its own table in the database, whereas with multi-table inheritance it does.

Using abstract base classes is in general simpler and more intuitive. Unlike with multi-table inheritance, fields can be overridden in child models, which could be useful in many cases. However, because the abstract table does not actually exist in the database, it cannot be queried and foreign keys of other models cannot be related to it. In the codebase there are several places where all kinds of questions need to be retrieved, and there are several tables where the foreign key relationship does not depend on the type of question. Therefore I elected to use multi-table inheritance.

When designing the inheritance hierarchy, I considered two alternative structures. The first involved fewer tables and was based off the observation that a Parsons problem had no more fields than a generic question, and program-type questions were similarly equivalent to generic programming (or debugging) questions. The second alternative contained several empty tables but was conceptually simpler for a human to understand. The second option would therefore likely have translated better in the admin interface (discussed later). However it would have involved slightly more logic in the codebase and had the potential issue that questions could be created which were not of a leaf node type. For example, mistakenly creating a Parsons problem as a Question (both models have the same fields) could cause that question to be missed in a query searching for Parsons problems. Because of this, I ended up choosing the first option. I was not aware of the admin interface issue at that stage, but as this issue was not severe and was solvable to a certain degree, it is unlikely that that knowledge would have changed my decision.

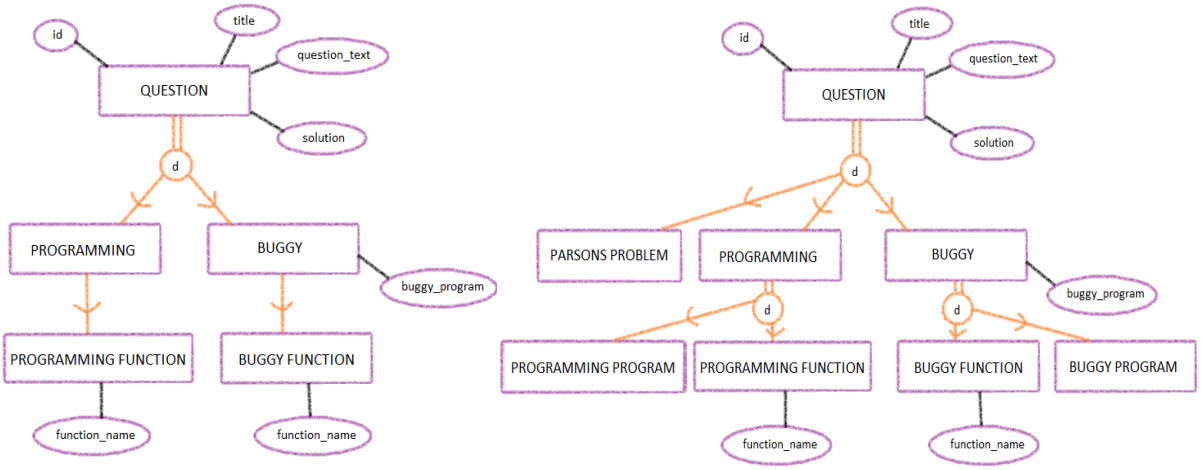


Figure 3: EER diagrams of the first hierarchy design (left) versus the second (right).

The only other database model which was impacted by the redesign of the question model was the test case model. I noticed that test cases could similarly be split into function-type test cases and program-type test cases. Function-type test cases were linked to function-type programming questions , while program-type test cases were linked to generic programming questions. While doing this I also changed the relationship between test cases and questions from many-to-many to many-to-one, because I decided that reusing test cases between questions would be more confusing than useful. Because multi-table inheritance does not allow parent fields to be overridden in child models and 'question' was a foreign key on test case, the test case model had to be split in a similar way to the second hierarchy option described earlier. The final database design is shown below.

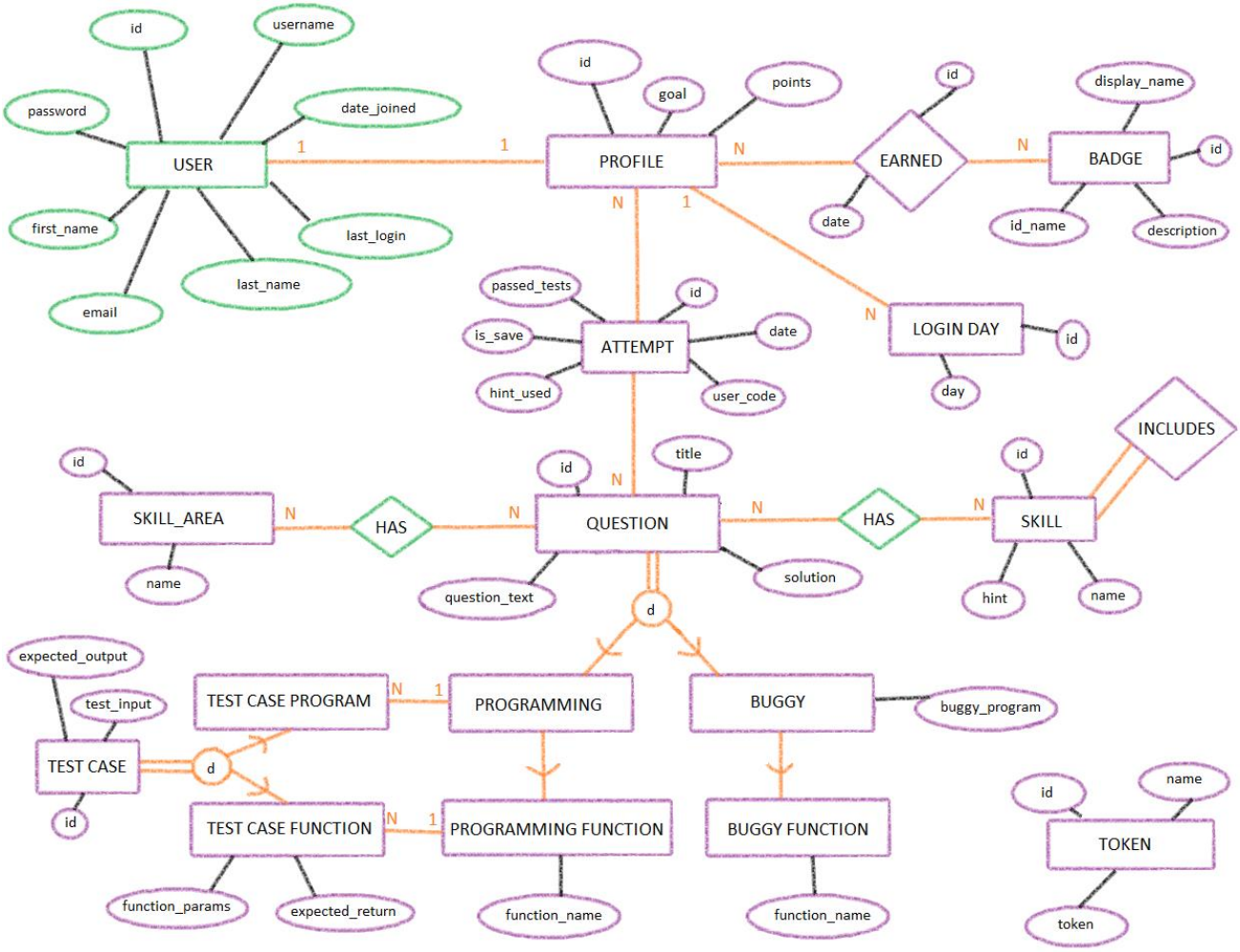


Figure 4: EER diagram of the final database.

3.3 Server

The use of multi-table inheritance in the database had some downsides which involved redesigning parts of the codebase. For example, whenever the parent class is queried, all instances are returned as the parent type. Each object has no knowledge of which subtype it is, which meant it had to be surrounded by try and except blocks when casting. I did not consider this practical, so I decided to install django-model-utils and

use their InheritanceManager [10], which has a method that returns the given object as an instance of the subtype.

I decided to use Jinja2 [11] to template all code which is sent to SphereEngine. The syntax of Jinja2 is very similar to the Django templating language used in the client. The main difference is that I used Jinja2 to template Python code instead of HTML. Variables such as the user-submitted code and the test case fields were then passed into the template. This required processing data from both the database and JSON sent as part of each Ajax request. Variables passed into the template had to be string representations of valid Python code – for most cases this is achieved by using literal_eval() followed by repr(). The literal_eval function does not share the same security issues as eval() because it is only able to evaluate literals.

The processing is successful in most ordinary use cases, but it is likely there are cases which I haven't considered or tested. For example, during user testing, one user managed to get programming questions correct by first printing the result of Python's globals() method (which returns global variables of the current program), then printing the variable which stored the test case's expected output. This, along with other identified issues, was recorded so that it may be fixed in future (see Appendix 4). It was not given priority to fix because it requires more advanced skill than the majority of users would have and there is no real advantage to gaming the system anyway.

One aspect of processing which was recently fixed was improving how newlines are handled. This relates to a previously identified usability issue in the question creation process on the admin site: administrators had to remember to write \n at the end of expected output for all test cases. Otherwise the user's code would fail the test case because the print function adds a newline to the end of whatever they had printed, whereas there would be no newline at the end of the admin-created test case. This was fixed by using rstrip() on both the user code and the test output. Additionally, the admin interface was changed so that expected input and output were textareas where newlines could be entered as real newlines rather than as \n.

All question types currently use the same Jinja2 template. The template logic is easy to follow, but could be split into sub-templates if more question types are added in future. If I was doing this again, I would make the debugging question type use a different template to the normal programming questions. Currently the same template is being used twice for debugging question types: once to get the output and/or return value of the ideal solution, then the second time to compare the ideal solution's results to the buggy solution's results. This could be combined into a single template so that only one request has to be sent to SphereEngine rather than two.

Another section of the codebase which holds important logic is the profile view. This view is responsible for calculating the question history, recent activity, points, and badges earned. Three badge types have been implemented: create an account, login for x consecutive days, and solve x questions. For each new badge type that is created, the code which checks the condition must be added on the server. Currently, all badge conditions are being checked when the user goes to their profile page as that is the only place where users can see their badges. This was done so that all badge checks would be

in the same place, making them easier to maintain. A disadvantage of this design is that users cannot receive immediate notifications when they gain a badge. For example, if a user completes their fifth question, they won't actually earn the corresponding badge until they go to their profile page.

The number of points the user has is also updated on the profile page. The maximum number of points a user can earn from each question is 10, awarded when the user passes all the tests for a question. 1 point is awarded for each attempt up to the third attempt. This means that the user can gain up to 3 points if they cannot figure out how to solve a question but attempt it multiple times. This is intended to reward users who attempt questions that challenge them and who persevere after a first failed attempt. If the user does solve the question after numerous attempts they still get 10 points, the same as somebody who completed the question first time. This is to prevent users from purposefully getting a question wrong a few times in order to gain more points. More checks could be added to discourage other ways of gaming the system, such as checking that each attempt is different.

3.4 Client

I made use of Django templates to reuse code across different HTML files on the client. All pages extend base.html, which includes the core elements and libraries necessary to every page. The navbar is included by default, although it can be overridden if desired. All files which extend base.html override the content block, and some also add their own JavaScript or override the page title. Most of the pages follow this simple format, except for the question page. Each type of question (programming, debugging, or Parsons) uses a different template. The question page decides which HTML template, CSS, and JavaScript to include based on the question subclass (given as a context variable).

One of the inconvenient things about Django is that when I moved my JavaScript into separate files, Django's context variables could no longer be accessed from the JavaScript. This was not a major issue, however, as the variables could be passed in by defining new variables in the base.html or question.html files.

The index page recommends up to five questions for the user to attempt. At the moment the only constraint these recommendations are based on is that the questions have not yet been attempted by the user. If the intelligent tutoring aspect of the website is developed further, these recommendations could be better attuned to the user's skillset.

The skill area page displays a list of questions belonging to that skill area. Each question is coloured green, yellow or purple depending on whether the user has completed, attempted, or not attempted each question. It is expected that the way this is indicated to the user will change, as the colours are not especially clear (or aesthetically pleasing).

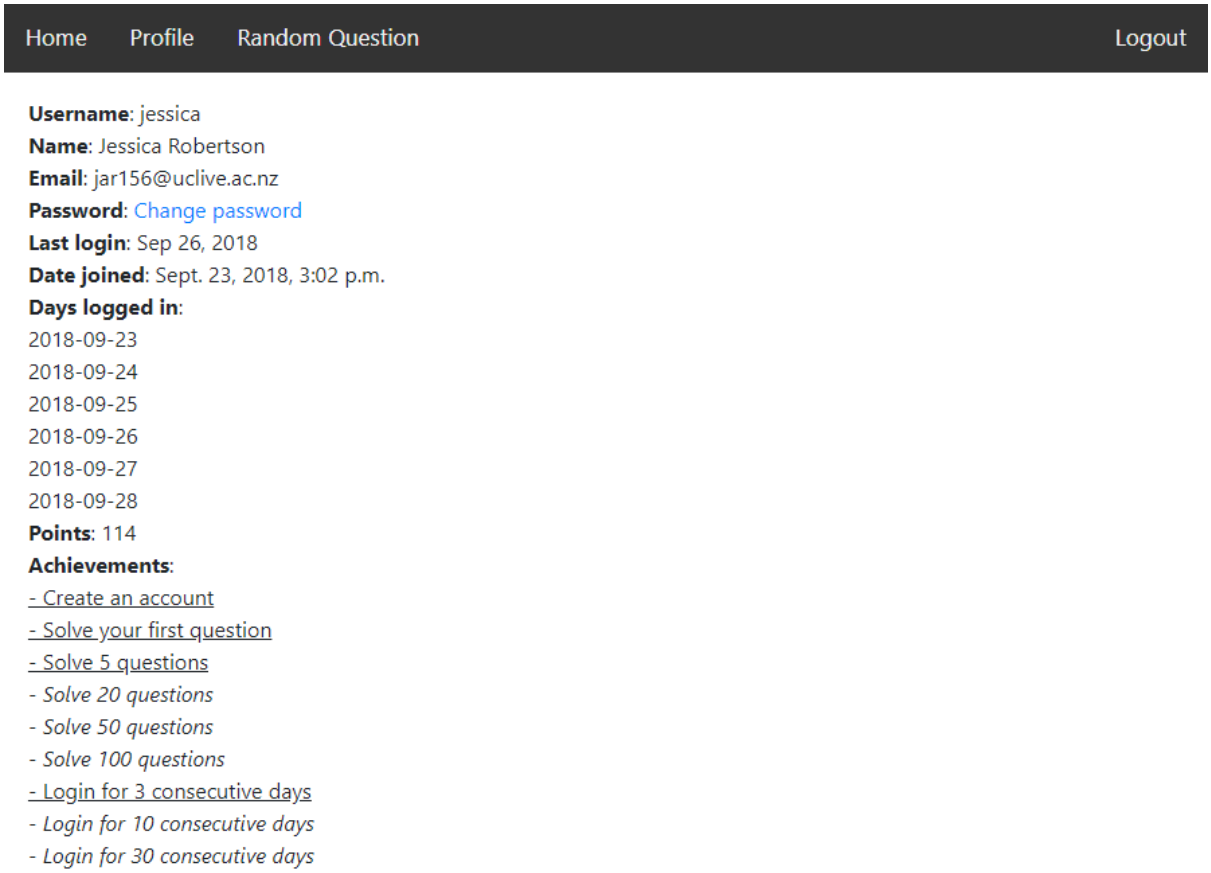


Figure 5: Profile page while logged in as "jessica"

The top part of the profile page can be seen in Figure 5. A design choice was made to only allow the user to view their own profile, as there is no reason users need to be able to view other users' personal details. If an "add friends" option is later added, this could be changed but it is more likely that a separate public profile view would be created which only contains a subset of each user's information.

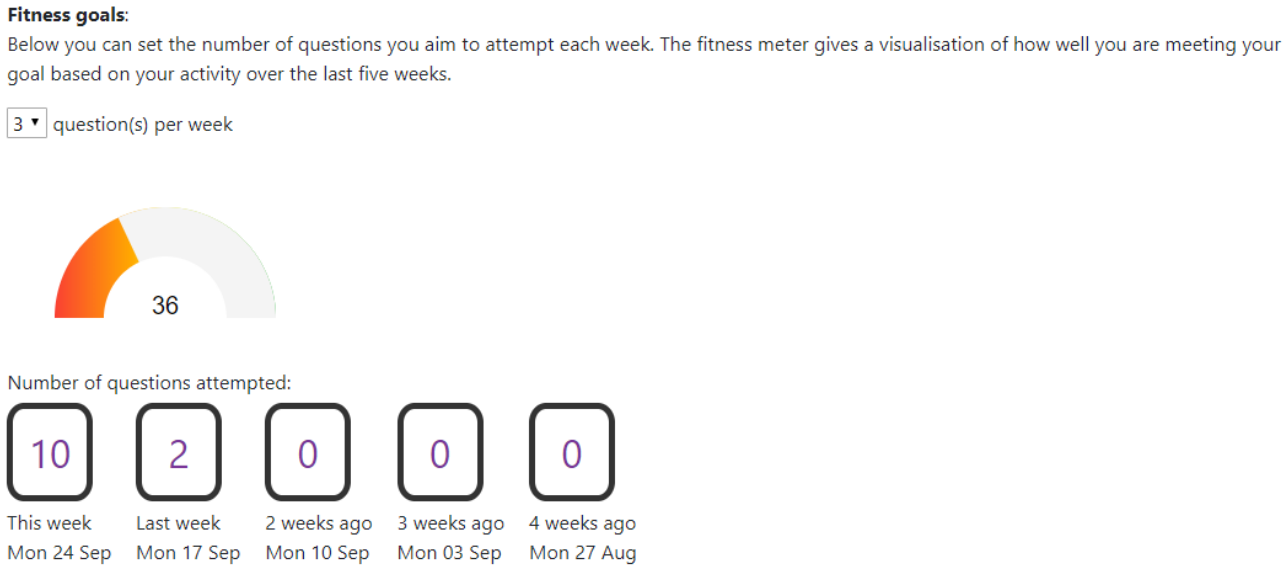
The profile page displays most information which is stored about the user, with the exception of the password, which instead has a link to allow the user to change their password. Not all of this information needs to be displayed – for example, the days logged in may not be particularly interesting or helpful to an ordinary user.

The Achievements section only has a very basic level of CSS applied to it. In future it would be ideal if these badges were presented in a more aesthetic format, perhaps with an icon associated with each.

The next part of the profile page displays a section where the user can monitor their recent activity. They can set a personal goal of how many questions they want to do each week. Currently the allowed goal range is between 1 to 7 questions per week, and it can be updated at any time. The number of questions completed each week for the last five weeks is displayed in a row of boxes.

The fitness section also displays a gauge to visually represent how well the user is meeting their fitness goal. This gauge was made using kuma-gauge, a JQuery plugin with a range of customisations available [12]. Kuma-gauge demonstrates a risk of using

open-source third-party plugins. There is a small amount of documentation associated with it, but it is otherwise not very well supported – the last update to the plugin was four years ago. I found a bug in it where the value was set to 50 if the gauge needle was not visible. I had disabled the needle because the needle did not move with the value and it looked better without. This was an easy bug to find and fix, but a more subtle bug in a third-party library might have caused a negative impact. However, a benefit of kuma-gauge being open-source was that the bug could be fixed.



The formula for the fitness score is experimental and could be easily changed at a later date if it is found to be unsuitable. Currently it is based on the last five weeks. Each week is given a different weighting – 0.5, 2, 1.5, 0.75, 0.25 from most to least recent. How close the user was to achieving their goal that week is also taken into account. The final score is between 0 and 100.

The weights are based on the idea that more recent activity contributes more to fitness than older activity. The exception to this is for the current week, where the weight is 0.5. The weight is less here because there is still time left for the user to complete their goal for that week. For example, if it is Tuesday and the last time the user completed questions was Sunday (the day before the current week started), their fitness should not have dropped significantly, but should still have dropped a little to remind them that they haven't completed their goal for the week. Ideally the formula should be visible to the user, but I am unsure how this information could be presented simply and concisely.

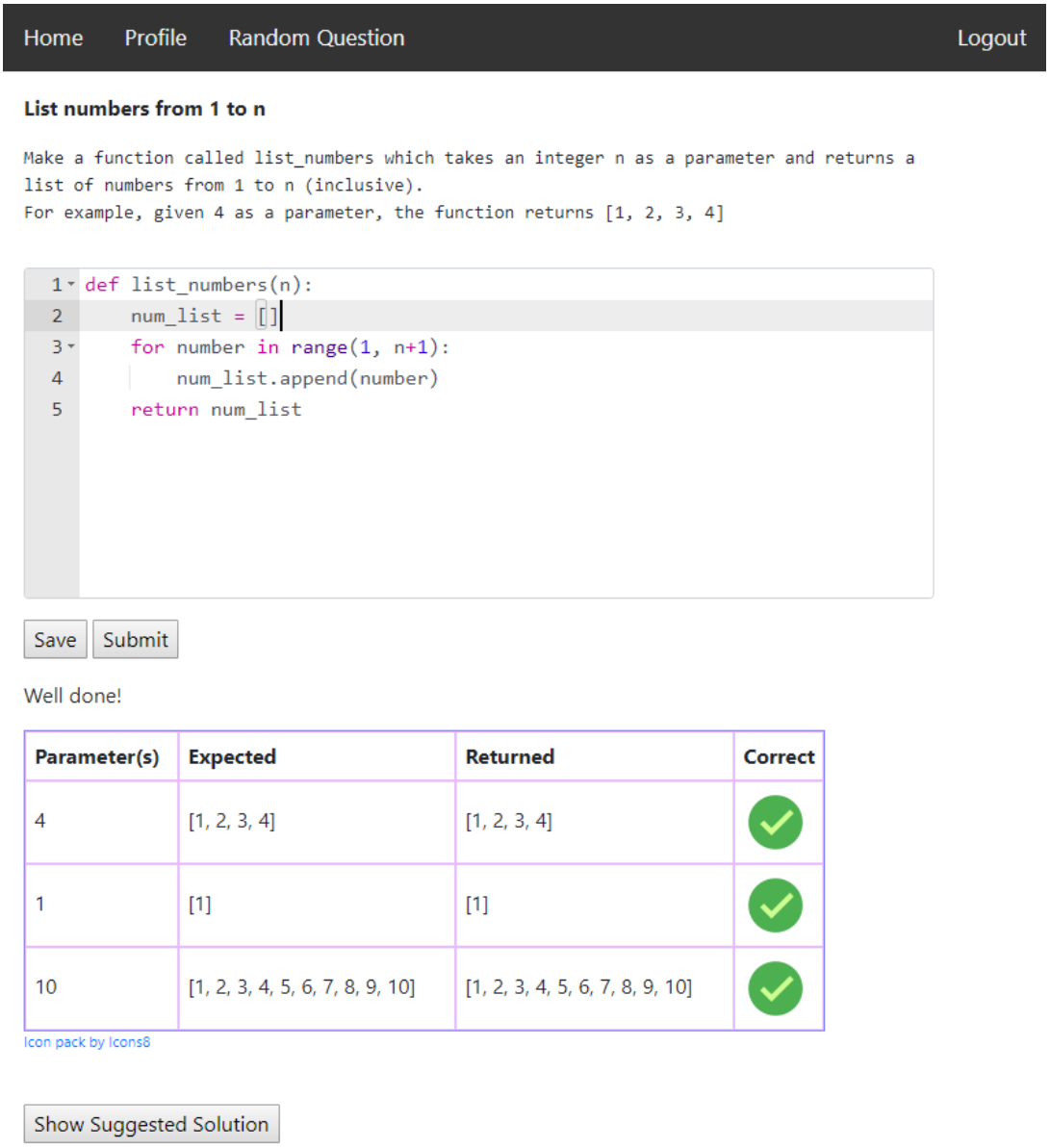


Figure 6: Question submission for programming type question.

The question page has been designed to look similar to CodeRunner quiz questions. The user enters their code in an editor with syntax highlighting. There were several choices for which to use, the most promising of which were Ace [13], CodeMirror [14], and Skulpt [15]. Skulpt was disregarded due to its complexity relative to the other two, although it could be an option to consider once SphereEngine is no longer being used. There did not appear to be a significant difference between Ace and CodeMirror. Both are open-source projects which have recent commits and a reasonable amount of documentation. They both support over 100 languages and have customisable themes. Ultimately, Ace was chosen based on the fact that CodeRunner uses Ace so it would be familiar to users and developers who have experience with the university's quiz server.

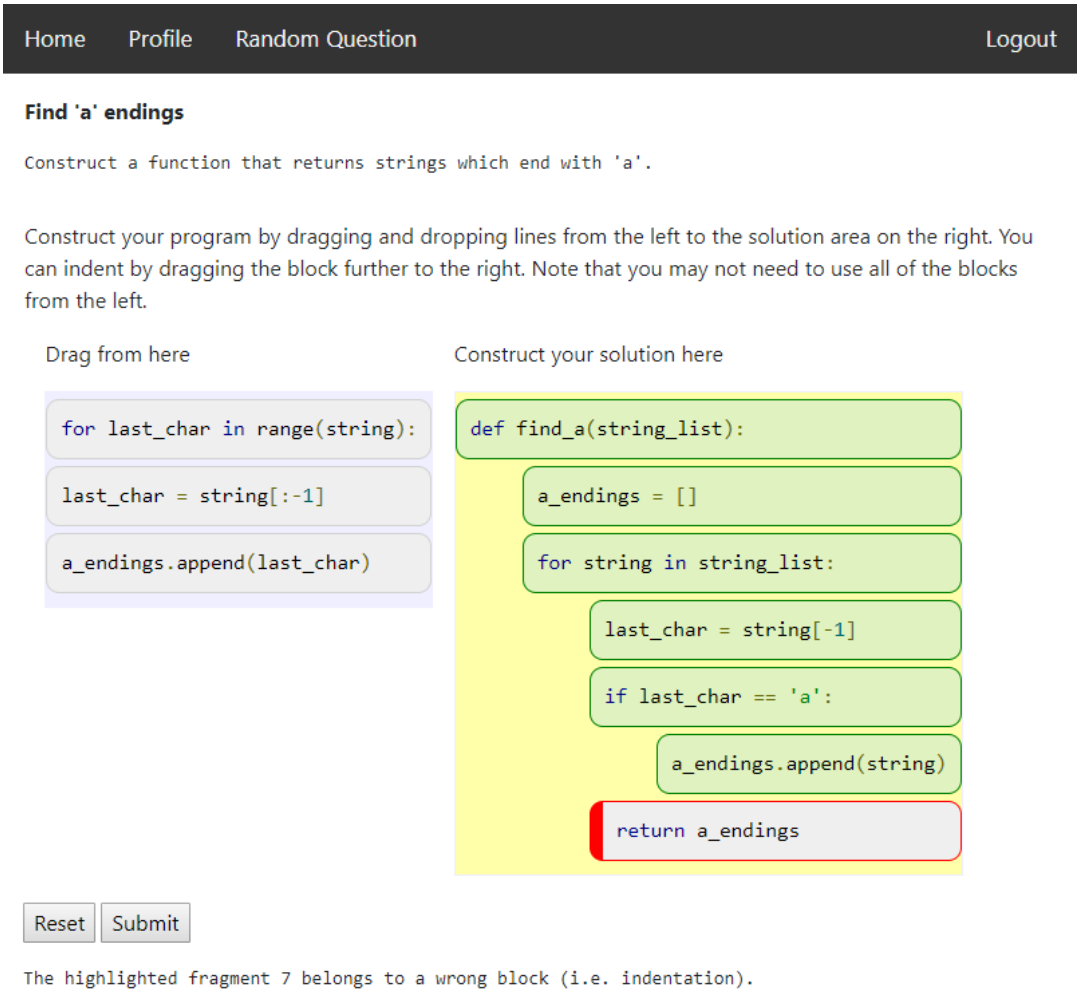


Figure 7: Parsons problem where the user has submitted a solution with incorrect indentation.

The second question type to be implemented were Parsons problems. This question type provides every line of the correct solution, but in a mixed-up order. Unnecessary lines called distractors can also be present. It is the task of the user to assemble the right blocks into the correct order. The blocks must also be indented correctly if the program involves indentation. Parsons problems are beneficial because they involve low cognitive load while still providing learning gains [16]. They also take significantly less time than writing code from scratch [17].

I used an open-source JavaScript library called js-parsons to implement Parsons problems [18]. This was a slightly risky decision because the library is not well documented. The website claims that documentation is under construction but the repository has not been contributed to in several years. Fortunately, it was easy to set up, and had all the key features I required. Little is dependent on this specific implementation, so it should be simple to switch if a better implementation is later discovered.

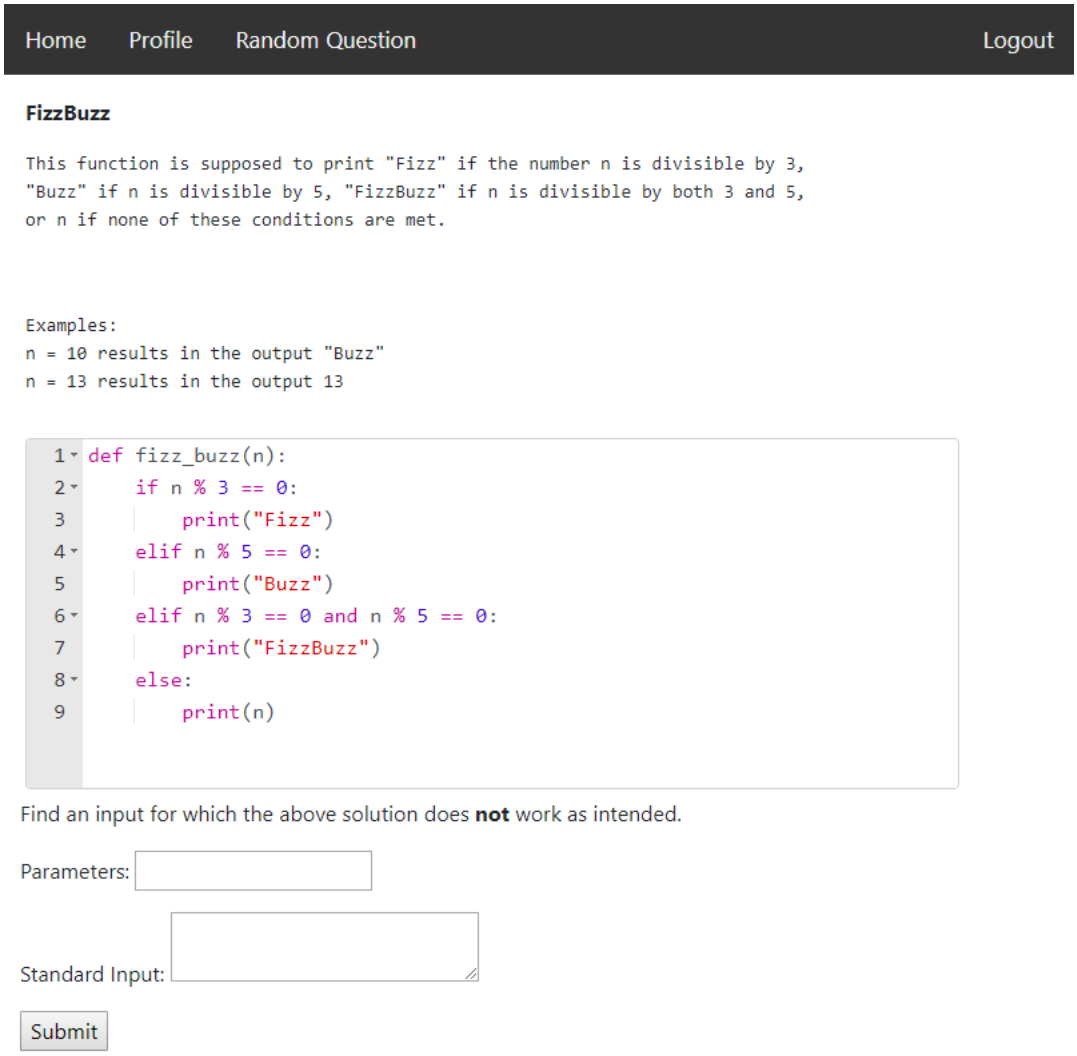


Figure 8: Debugging type question.

Another question type which I expect to be useful for teachers is the debugging question type. This question type presents the user with a buggy program or function and asks them to provide an input which will make the buggy solution behave unexpectedly. I do not know of any other websites which offer these kinds of questions. It is fairly similar to a code tracing question – finding the solution usually involves some amount of tracing. Code tracing has been found to improve code writing skills pertaining to both syntax and semantics [19], while utilising different skills than programming or Parsons problems [20]. Like Parsons problems, the debugging question type involves a lower cognitive load than developing a solution from scratch. This question type can be useful for raising awareness of common mistakes. It is intended to encourage teachers to think critically and consider edge cases – a skill which could be useful when helping students make their programs more robust.

Screenshots of pages not shown in this section are given in Appendix 6.

3.6 Admin Interface

One of the key reasons Django was chosen as the web framework for this project was because of its built-in admin interface. This allows the content of the database to be viewed and managed from the '/admin/' URL, which is only accessible by superusers. During development and testing, all models were registered in the admin interface, meaning everything could be viewed, created, edited, and deleted. In future, when the site is live and the question makers have access to the admin interface, the interface should only contain the question, and test case models. This could be done by creating another user group (staff) which has fewer permissions than the developer group (superuser).

For the most part, the default admin interface was considered sufficient. However, towards the end of the project, I made several improvements to increase its usability from the perspective of a staff user. Firstly, help text was added to the top of the Parsons problem question page in order to advise the user on the syntax for setting up a Parsons problem. Help text can also be set for other question types but is currently empty.

Due to the way the question and test case models were set up in the database, parts of the admin interface had to be modified to make creating and editing questions easier. For example, I made a modification so that staff can create test cases inline on the same page as they create a question, instead of creating them separately and having to manually link the test cases to the question.

A downside of the Django admin interface is that while it provides many useful features out of the box, it is far less flexible than the main website. An example of this was when adding another column to the question list view. The new column shows the question type, calculated based on the actual subclass of each instance. Unfortunately because the interface can only search and sort by model fields (or columns calculated from a model field), the new column cannot be searched or sorted.

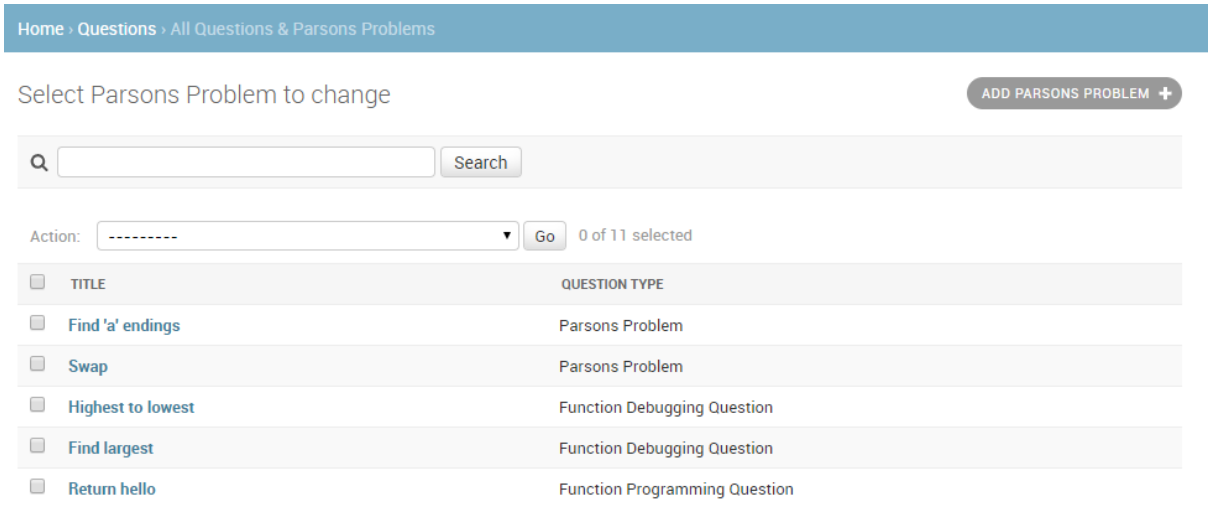


Figure 9: Parson problem list view. Due to the Parsons problem model being the same as the generic parent question model, this screen shows all question types.

Another issue with the default interface was that editing questions from the edit page of any parent question class only showed and allowed editing of the fields on the parent table. This could be confusing if the user was editing a function-type programming question but the function name field was missing.

My initial idea for how to solve this problem was to hide instances of child models from the parent question list view. However, I decided it would be useful for staff to be able to see all questions in the parent list view. Otherwise they might get confused about the number of questions on the site if they had created some programming questions but their newly created questions didn't appear in the parent list view.

Therefore I decided to disable the edit page if the question the user wanted to edit belonged to a subclass as well as the parent class.

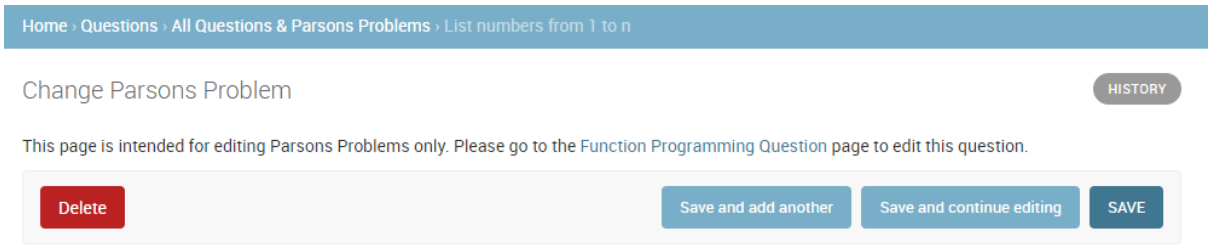


Figure 10: An error message along with a link to the appropriate page is provided.

3.7 Testing

Django includes by default a testing framework based on Python's unittest. Three types of unit tests were performed: model tests, form tests, and view tests. Each type has its own file which is split further into different classes for every model, form, or view. Coverage.py (a tool for measuring code coverage of Python programs [21]) reported that model coverage is 93%, form coverage is 100%, and view coverage is 67%. View coverage should be higher – this is what I would focus on in future. See Appendix 5 for a list of currently implemented test cases.

Django's built in Client class is useful for unit-testing views. However, for integration tests, Selenium is more practical. These tests take longer to run than the unit tests so are kept in a separate file so that they can be run independently.

A limitation to the question tests is that they send code to SphereEngine. This would not be ideal if the subscription was pay-per-request. In addition, the tests take a while to run. Unfortunately there was no easy way to avoid this. I could have used mocking in order to unit test the server-side functions used for running questions. However, I could not do the same for the actual Python templates used for judging because the templates need to be rendered then run. This is what the server-side functions already do so it was simpler to do integration tests rather than reimplement this behaviour. Given that integration tests included the server logic, I did not feel it was necessary to unit test those particular functions in this scenario.

Test-driven development was not used extensively as a development strategy for this project. Some of the tests were written before development on that feature was carried out, but the majority were added afterwards in order to ensure the feature worked as expected and continued to work into the future.

The testing strategy mostly focused on blue sky scenarios along with several tests of common alternate paths. Where testing of more complex features was performed, equivalence partitioning and boundary value analysis was used. For example, some of the tests included testing a badge which is supposed to be awarded when the user has logged in for 3 consecutive days. The boundary value was 3, so I tested 2, 3, and 5 days. Another partition was consecutive runs including the current day, and consecutive runs not including the current day. Therefore I tested that a badge was not awarded on the user's 2nd consecutive day (including today) but was on the 3rd. Then I tested whether the badge would be awarded for a run of 2, 3, or 5 days from a week ago.

The majority of the testing was done manually as features were developed. The automated tests proved very useful when the database was redesigned and code was re-engineered, but it was important to continue to perform some manual tests alongside the automated ones to ensure that the tests were not incorrectly passing.

For manual tests I used a Chrome browser. Selenium integration tests run using Firefox. Older browsers such as Internet Explorer have not been tested but should be, because some teachers may not be using modern software.

In terms of user testing, six different users were asked for feedback at various stages of the project. Two of these were Python beginners (the main intended audience), while the others had at least a year of Python experience and were comfortable developing in Python. The users were asked to look around the site while unregistered. Then they were asked to sign up and complete some questions, as well as have a look at their profile page. At each stage in this process I prompted them for feedback and watched how they interacted with the website. The actual feedback given by these users is discussed in a later section of this report.

More user acceptance testing with the intended audience needs to be performed. It was originally planned that the teachers of the EDEM665 would be asked for feedback on the website. However, due to this course being a remote course, a public server for the website would need to be set up. By the time the project was in a state where user acceptance testing would be the most helpful, there was not enough time to set up the server, send out the link to teachers, and collect their feedback. I could have asked for feedback earlier on in the project, and again after the project was completed, but it was expected that teachers would be less likely to respond to the second request for feedback if they had already given feedback once. Therefore it was better to wait until after the project was completed.

4 Discussion

4.1 Learning Outcomes

I had never used Django before starting this project so I have learned a lot about this framework and web development in general. Django's clear and detailed documentation was very helpful to my learning – in particular the tutorial which I started out with. I also gained a better understanding of the pros and cons of working with a dynamically typed language on a long-term project. For example, I ran into a few issues due to not knowing the types of variables, especially when refactoring.

I also learned that it is important to take the time to plan the database design carefully, because changing key elements of the database can be expensive due to the amount of code and tests that rely upon specific models. It is not reasonable to expect that I could have planned the database perfectly in my initial design. However, reviewing the design of the question table before adding the functionality for the two new question types (Parsons and debugging) would have been a good idea.

While it took a significant amount of time to rewrite the tests which broke whenever a database model was altered, having those tests was still very useful for regression testing. That process also helped me learn which kinds of tests were more helpful than others. For example, I initially had a test which checked that a model field had a certain length, but because that constraint was not relied upon by any part of the codebase, that test was not worth keeping. By contrast, another field's values being zero could cause division issues in client-side code, so that did need a test associated with it.

4.2 User Experience and Feedback

The final look and feel of the site has not yet been designed. At the moment only a basic level of CSS has been applied to the site. The main purpose of this was so that development could be more flexible. Elements could be added, removed, or moved around on a page (or to a different page) without having to worry about fixing the CSS every time. Ideally, before design decisions are made, more research needs to be done into which aesthetics teachers would prefer and what kind of flow would motivate them to continue practicing questions on the site.

The test users provided some useful feedback with regards to user experience, along with suggestions of features which could improve the quality of the website. The most common feedback from test users was that the interface for Parsons problems could be easier to use. In particular, the boxes should be bigger so that drag and drop would be more lenient. Several users missed the fact that not all blocks were required in the solution despite the note above the solution area. This note could perhaps be bolded so that it would stand out more. Otherwise the user could be presented with a modal dialogue box when they first see this type of question – this would draw more attention to the note as it would mean they have to acknowledge the modal before proceeding.

One of the beginner test users suggested that hints would be a useful feature because they thought they would get stuck with the more difficult questions and having a hint available would lessen the likelihood of them giving up. They also suggested that a further explanation of the correct answer and how the function worked would be useful for both the Parsons problems and debugging question types, because it is possible to guess the answer for these question types without fully understanding what is going on.

Another suggestion was to have a warning if the code style in the user's solution was not recommended (for example, too long or short variable names). This came up when the test user noticed that their solution differed from the suggested solution. Perhaps a note would also be useful here, letting the user know that their solution being different from the suggested solution does not necessarily mean it is worse in any way.

I received positive feedback on the fitness gauge and question history sections from nearly all test users, although one felt that this did not belong on the same page as account information. Another said that it would be good if points were visible on the home page or on a sidebar while completing questions. That user said that the core concepts of the site were nicer and more motivating than similar sites they were familiar with, but that the interface could be improved visually. Additionally, the user suggested that there be levels or ranks based on the number of points accumulated, and reaching higher levels could unlock aesthetic rewards such as special site themes.

4.3 Future Work

The project met most of its objectives. Some aspects may require fine tuning after more user feedback is gathered. In particular, the fitness score and points calculations could be altered based on further research involving real users. The CSS of the website should also be refined.

The largest planned feature which was not implemented was the intelligent tutoring system. This system was intended to be able to tell which types of questions a user often gets incorrect and recommend problems to them accordingly. Another feature which could have been good to have was dynamic hints. The idea of this was to automatically generate hints to offer to the user based on the skills involved in the question, rather than hints having to be defined manually for every question. Having the option for a hint should help prevent users getting frustrated at not being able to solve some questions and losing confidence and motivation. Both this hint system and the intelligent tutoring feature would depend upon a skill tree being created. This is a simple task but it would involve a significant amount of research in order to be done properly.

More question types which help teachers with skills they might need in the classroom could also be added in future. For example, a question type which asks them to select the line which contains the bug in a piece of code. The interface for this could look similar to a Parsons problem. The buggy code could be automatically generated from correct solutions in order to increase the number of possible questions, as writing questions manually can be a time consuming process. Other desired question types

include questions which ask whether two solutions are identical, and questions which require the user to assign a value to a particular variable (useful for beginners).

5 References

[1] Bell, T. (2014). Establishing a nationwide CS curriculum in New Zealand high schools. *Communications of the ACM, 57*(2), 28–30.

[2] Sentance, S., & Csizmadia, A. (2017). Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies, 22*(2), 469–495.

[3] Lobb, R. & Harlow, J. (2016). Coderunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads, v.7 n.1*, 47-51.

[4] 3bits elearning solutions. (2018). *Adaptable - Home*. Available: https://adaptable.ws/

[5] United for Wildlife. (2017). *United for Wildlife Online Courses*. Available: https://learn.unitedforwildlife.org/

[6] Lobb, R. (2018, Apr). *JOBE*. Available: https://github.com/trampgeek/jobe

[7] SphereEngine. (2018). *Documentation*. Available: https://developer.sphere-engine.com/

[8] SQLite. (2018). *Appropriate Uses for SQLite*. Available: http://www.sqlite.org/whentouse.html

[9] Bissex, P. (2016, Jan). *Is it difficult to switch from sqlite3 to Postgres in Django?* Available: https://www.quora.com/Is-it-difficult-to-switch-from-sqlite3-to-Postgres-in-Django-Im-just-starting-out

[10] Meyer, C. (2015). *Model Managers*. Available: https://django-model-utils.readthedocs.io/en/latest/managers.html

[11] Ronacher, A. (2008). *Jinja2 Documentation*. Available: http://jinja.pocoo.org/

[12] Bellen, S. (2014). *Creating Animated Gauges Using jQuery and Raphael.js – kumaGauge.* Available: https://www.jqueryscript.net/chart-graph/Creating-Animated-Gauges-Using-jQuery-Raphael-js-kumaGauge.html

[13] Ajax.org B.V. (2018). *Ace – The High Performance Code Editor for the Web*. Available: https://ace.c9.io/

[14] CodeMirror (2018). *CodeMirror*. Available: https://codemirror.net/index.html

[15] Miller, B. (2018). *Skulpt*. Available: http://www.skulpt.org/

[16] Ericson, B. (2018). Evaluating the effectiveness and efficiency of Parsons problems and dynamically adaptive Parsons problems as a type of low cognitive load practice problem. *Doctoral dissertation, Georgia Institute of Technology*. Available: https://smartech.gatech.edu/handle/1853/59890

[17] Ericson, B., Margulieux, L., & Rick, J. (2017). Solving parsons problems versus fixing and writing code. *Proceedings of the 17th Koli Calling International*

Conference on Computing Education Research (Koli Calling '17). ACM, New York, NY, USA, 20-29.

[18] Karavirta, V., Ihantola, P., Helminen, J., & Hewner, M. (2014). *js-parsons - a JavaScript library for Parsons Problems*. Available: https://github.com/js-parsons/js-parsons

[19] Kumar, A. (2015). Solving Code-tracing Problems and its Effect on Code-writing Skills Pertaining to Program Semantics. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*, 314-319.

[20] Denny, P., Luxton-Reilly, A., Simon, B.. (2008). Evaluating a new exam question: Parsons problems. *Proceedings of the Fourth international Workshop on Computing Education Research (ICER '08)*, 113-124.

[21] Batchelder, N. (2018). *Coverage.py*. Available: https://coverage.readthedocs.io/en/coverage-4.5.1/

6 Appendix

Appendix 1

List of user stories generated as part of initial requirement research. Blue items represent stories which were completed over the course of the project. Purple items have been partially implemented, or implemented in a different way than intended.

Teachers

- I want to have an account so that my progress can be saved
- I want to have the option to be emailed if I haven't done any practice problems in a while so that I don't forget
- I want to write Python code in the browser so that I don't have to use an IDE
- I want to be able to save my code so that I do not lose what I have done if I leave the page
- I want immediate feedback telling me if my solution is correct or incorrect so that I know if my code needs to be fixed
- I want to see hints if I have attempted a problem several times without success so that I can find out how to solve a problem if I have no idea how to do it
- I want to be given problems at my level so that I am neither out of my depth nor bored
- I want to know which problems I have already done so that I don't unintentionally repeat the same problem twice
- I want to review problems I have done so that I can remember how to do those kinds of problems
- I want to collect badges for achievements so that I have some reward for my efforts
- I want to practice problem solving so that I am more confident with programming
- I want to practice using common Python built in functions so that I don't get stuck if a student uses one and I don't know what it does
- I want to practice finding bugs in code so that I have the ability to help my students find their bugs
- I want to practice determining whether a solution is equivalent to another solution so that I don't tell a student their solution is wrong when it is correct (or vice versa)
- I want to see alternative solutions to problems so that I can consider different ways of solving problems
- I want to have the option of being anonymous so that others don't judge me for my coding abilities
- I want to be able to connect with friends so that we can motivate each other
- I want to be able to post my achievements on Facebook so that my friends can see
- I want to be able to choose how many problems to do so that I can set appropriate goals for myself
- I want to see my progress so that I am motivated to continue
- I want to know what I need more practice on so that I can focus on that area
- I want a random question button so that I don't have to choose what to attempt
- I want to visit the forums so that I can discuss with other teachers

Administrators

- I want to be able to add questions so that users have a larger range of problems to solve
- I want to be able to remove questions so that users don't do unsuitable questions
- I want to be able to update questions so that I can fix mistakes in them
- I want the interfaces for adding, deleting, and updating questions to be easy to use to that it doesn't take me long
- I want to see site usage statistics so that I can track how the platform is being used
- I want to identify how difficult each question seems based on the effort/attempts required by users to answer them

List of user stories generated as part of initial requirement research. Blue items represent stories which were completed over the course of the project.

Appendix 2

List of technical requirements created from user stories in Appendix 1 as well as new ideas which came up during development. Blue items represent tasks which were completed as part of the project.

Core

- Check user functions (involving params and return) against test cases
- Check user programs (involving stdin and stdout) against test case

Account (Required)

- Login to user account
- Logout from user account
- Registration page (ask for username, email, password, first name, last name)

Account (Basic)

- Allow people to solve problems without an account, just don't show profile page or save any data
- Display weeks the user has logged in with number of different questions attempted for the last 5 weeks
- Profile page shows basic details (name, email)
- Profile page shows history of questions done (question name, n_attempts, date)
- Fitness gauge
- Allow user to set goal of x questions per week (dropdown from 1 to 7)
- Allow user to change password

Account (Advanced)

- Display list of achievements and whether the user has completed them
- Check when user satisfies badge condition
- Badge to reward account creation
- Badge to reward logging in for consecutive days
- Badge to reward solving questions
- Badge to reward logging in for consecutive weeks
- Badge to reward maintaining/surpassing fitness goal
- Give user 1 point for attempting a question once, 1 for next two attempts, or 10 if they get it right
- Don't give points for correctly answering if user viewed suggested solution

Storage

- Store user code in database if user presses save button
- Store user code in database after each submission
- Load latest attempt if one exists
- Visual indication that saving was successful
- Button to show full solution
- Store question maker's solution
- Store history of attempts linked to user and question (date, code, hint used, correct)
- Store user points
- Allow admin to create test cases inline on question page

- Question help text in admin interface

Intelligent Tutoring

- Give user less points if they get question right using a hint
- Button to show a hint
- Next question button to go to another problem in skill area (requires info on which skill area the user is in which is difficult considering random question button)
- Sort failures into skill categories and store count of failures in each skill category
- Suggest problem to practice which is in a skill area which they have failed at before - don't suggest an area with too many failures

User Interface

- Navigation bar with Home | Profile | Random Question | Logout etc
- Button on nav bar to go to random question
- Display whether code passed all the test cases
- IDE in browser with syntax highlighting
- Show 5 questions which haven't been attempted on index page
- In skill area and question history sections, show questions as green that have already been completed, yellow if attempted

Question Generation

- Support debuggy question type (find input which causes bug)
- Support Parsons problem question type
- Support 'are these identical solutions' question type
- Generate identical solutions
- Support click which line the bug is on (like Parsons problem) question type
- Generate buggy solutions from correct solutions

Other

- Test cases
- Basic CSS
- Script to fill db
- Design skill tree with corresponding hints

Non-priority

- SMTP server can send user email to reset password
- SMTP server emails user if they haven't logged in for user-specified period of time
- Show max 20 (unsolved) problems for each skill area
- Let user set their goal activity level and whether to email them if they don't stick to it
- Post badges to Facebook
- Button to cycle through alternative solutions after correct submission
- Allow users to login with Facebook, Gmail etc (django-allauth)
- Search for and add friends
- Option to share progress with friends
- Forums to chat about problems / programming / teaching in general
- Allow user to make themselves anonymous (while registering?)

Appendix 3

List of similar programming websites researched

- Grok Learning
- Code Avengers
- Codingbat
- Hackerrank
- Exercism
- Project Euler
- Code wars
- Codecademy
- Advent of code
- Codingame
- Practice Python

Appendix 4

List of identified bugs or potential issues to look into later. Issues that were put on this list but later fixed have been removed.

- Could support different versions of print eg print(x, y, z) print(x, y, sep="-")
- There is a visible delay when changing from cross to tick on the results table
- Could update attempt with new info instead of adding a new one each time and storing all past attempts
- In question description lines should wrap instead of scroll if no newlines
- Gracefully handle debug input which is str when expecting numbers etc
- Line numbers in tracebacks are not correct from point of view of user (use try except to catch errors in user code)
- Disable non-editable models eg login days in admin
- Buggy page console.logs errors - there shouldn't be errors for this type but could be, these should be handled better
- Not allowed triple quotes in expected output of test case
- Return None as string to client instead of null
- Don't show the box (params or stdin) that you're not meant to put anything in (debug question)
- Auto scroll when results box comes up
- Save icon's width is based on % not px so it looks stretched sometimes
- Skill area page should display which skill area is being looked at
- Make parsons drag/drop boxes bigger
- Draw more attention to note saying tha not all Parsons blocks need to be used
- Show further explanation (after correct attempt) for Parsons and debug questions
- Warn user if using poor code style
- Let user know that their solution being different from the suggested solution is not necessarily a bad thing
- Add a level up system
- Unlock themes for the site when user gets higher number of points

Appendix 5

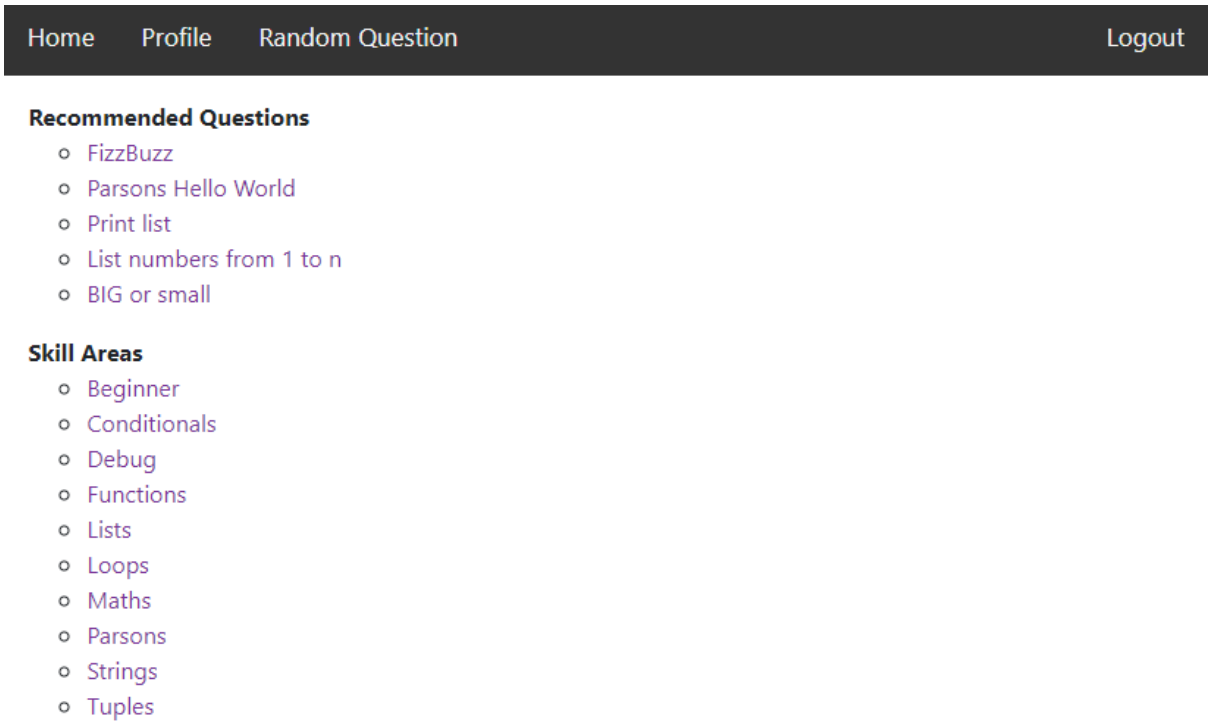
Current Test Cases

- Profile view
    - Redirect to login page if not logged in
    - Profile url exists
    - Profile url displays expected template
- Badge view
    - Create account badge awarded after user creation
    - Create account badge doesn't aware twice
    - Adding unknown badge type doesn't cause error in badge view
    - Consecutive login of 2 days including today does not award login-3 badge
    - Consecutive login of 3 days including today awards login-3 badge
    - Consecutive login of 3 days from 1 week ago awards login-3 badge
    - Consecutive login of 5 days from 1 week ago awards login-3 badge
    - Consecutive login of 2 days from 1 week ago does not award login-3 badge
    - Completing 1 question awards solve-1 badge
    - Attempting 1 question does not award solve-1 badge
- Buggy question view
    - Buggy program with stdin and stdout works as expected
    - Buggy function with parameters and return (no stdin or stdout) works as expected
    - Buggy function with parameters and return and stdin and stdout works as expected
- Question view
    - Question url exists when not logged in
    - Question url exists when logged in
    - send_code function returns submission id
    - Program with stdout works as expected
    - Program with stdout and multiple test cases works as expected
    - Program with stdout and stdin but blank stdin test case works as expected
    - Program using tab to indent works as expected
    - Program using 4 spaces to indent works as expected
    - Program using escaped newline works as expected
    - Function with return and single param works as expected
    - Function with stdout and single param but no return works as expected
    - Function with stdout and return and single param and multiple test cases works as expected
    - Function with return and single param and multiple test cases including empty string param works as expected
    - Function with return and multiple params works as expected
    - Semantically incorrect function with return and multiple params works as expected (fails test case)
- Sign up form
    - Email help text is provided
- Token model
    - Creating token with same name fails (name field unique)
- Badge model
    - Creating badge with same id_name fails (id_name field unique)
- Profile model
    - New profile starts with 0 points
    - New profile starts with goal level as 1
    - Updating goal level to 4 succeeds (goal must be between 1 and 7)
    - Updating goal level to 0 fails (goal must be between 1 and 7)

        - Updating goal level to 8 fails (goal must be between 1 and 7)
- Question model
    - Question text label name
    - Solution label name
    - Question __str__ is same as question title
- Programming function question model
    - Is instance of Question parent class
    - Is instance of Programming parent class
    - Is instance of ProgrammingFunction child class
    - Not instance of Buggy class
    - Not instance of BuggyFunction class
    - Programming function question __str__ is same as question title
- Programming function question model
    - Is instance of Question parent class
    - Not instance of Programming class
    - Not instance of ProgrammingFunction class
    - Is instance of Buggy class
    - Not instance of BuggyFunction child class
    - Buggy question __str__ is same as question title
- Sign up integration
    - Registering with all fields valid succeeds
    - Registering with poor password fails with appropriate error message
- Question integration
    - Random question button takes user to question page

Appendix 6

Home



Skill Area

Account management pages