1. Overview

Code Activator is a web-based application intended to support evaluation and improvement of code reading skills. The application takes a sequence of computer programs, each displayed as an HTML web form with a few actual inputs or outputs in the code replaced by text fields. The student completes the fields; the solution is then checked automatically and feedback is provided immediately.

1.1 Who is the intended audience for this document?

There are three user roles for Code Activator.

- *Students* take quizzes and should read the **Game play** section of this document.
- Question *authors* write questions and assemble them into quizzes. Question authors should read the **Game play**, **Question generation**, and **Quiz specification** sections. The person responsible for grading students on their performance in Code Activator should also read **Log files and analysis**.
- The *webmaster*, responsible for running the Code Activator server and installing the content produced by question authors, should read **Server management**.

1.2 What is the purpose of this document?

This document specifies the required observable behaviour of the Code Activator system.

1.3 What are the prerequisites for the reader?

- Students reading this document should be familiar with the programming languages of the questions and with the terms *command-line arguments* (`argv`), *standard input* (`stdin`), and *standard output* (`stdout`).
- In addition to the knowledge assumed for students, question authors need a good understanding of Python strings, lists and dictionaries, as question templates and quiz specifications are written using these Python constructs.
- Webmasters should be familiar enough with the operating system on which the Code Activator server will be run to make any needed adjustments (creating firewall rules, running the web2py server under a restricted user, setting directory permissions, etc.). A moderate understanding of Python (enough to understand the web2py documentation) is an asset, but is not strictly required. Webmasters administrators should have an understanding of basic web server concepts and HTML.

2. Game play

CodeAct quizzes run in several modes.

Each quiz runs in either *practice mode* or *marked mode*. In practice mode, a student may check each answer for correctness; in marked mode, the student is given no information about answer correctness.

In *standalone mode*, there is a single question. Standalone mode is used primarily for tutorials.

2.1 Display layout

A Code Activator question displays a game board with the following cells:

- *code cell*: source code of a program,
- *argv cell*: text available from command-line arguments,
- *stdin cell*: text available as standard input, and
- *stdout cell*: text written as standard output.

Each cell may contain a mix of fixed text and text fields. The student's job is to complete the text fields.

The following buttons are displayed:

- A "Check answer" button is displayed in practice mode; A "Submit answer" button is displayed in marked mode.
- Except in standalone mode, "Previous" and "Next" buttons are displayed.

2.2 Questions and answers

- *Input-output*: When the the check/submit answer button is pressed, the code is executed. The actual output of the execution is saved. The answer is correct if the actual output from this execution is the same as the contents of the stdout cell.
- *Bullseye*: The code cell contains one or more highlighted lines. When the check/submit answer button is pressed, the code is executed. The answer is correct if each of the highlighted lines was executed at least once.
- *Liar-liar*: The code cell contains one or more `assert()` statements. When the check/submit answer button is pressed, the code is executed. The answer is correct if each of the assertions was false at least once during execution.
- *Find-the-failure*: The code cell contains both a specification and an implementation of a function, and an invocation of that function. When check/submit answer button is pressed, the code is executed. The answer is correct if the return value of the function is incorrect, with respect to the specification

2.3 Further check answer semantics

Each text field has an associated type: integer, floating point or string. In addition, text fields in the argv cell may contain only digits, upper and lower case letters, hyphens, or underscore. If the value entered is not of the correct type, then the code is not executed and the answer is incorrect.

In I/O questions, the contents of the stdout cell and the output of the executed code are compared. Before this comparison is performed, all newlines are removed from the end of both outputs.

3. Question generation

3.1 What files comprise a generated question?

Each of the four codeAct question types has a set of code library files. Each of the `.txt` files in the set is

a template corresponding to a display cell and contains a mix of fixed text and text field markers. The text field markers such as `$x` are at the positions of text fields in the `.txt` files. We briefly describe these files for each of the question types:

1. Input-output:
    - `types.py`: A Python file that contains two variables:
        - `game_type`: indicates the type of the question. The following are the legal values:
            - `'input_output'`
            - `'bullseye'`
            - `'liar_liar'`
            - `'find_the_failure'`
        - `parameter_types`: a nested dictionary. For the display cells as the key, the value is another dictionary that a text field marker is the key and the value is the text field type.

      An example `types.py` file, from the question `py_min_io_stdin_0` .
    - `display.txt`: contents of the code cell. An example `display.txt`, from the question `py_min_io_stdin_0` .
    - `argvs.txt`: contents of the argv cell. An example `argvs.txt`, from the question `py_min_io_stdin_0` .
    - `stdin.txt`: contents of the stdin cell. An example `stdin.txt`, from the question `py_min_io_stdin_0` .
    - `stdout.txt`: contents of the stdout cell. An example `stdout.txt`, from the question `py_min_io_stdin_0` .
    - `run.sh`: a shell script that executes the program with command-line arguments. An example `run.sh`, from the question `py_min_io_stdin_0` .

    Python only:
    - `execute.py`: A Python program. An example `execute.py`, from the question `py_min_io_stdin_0` .

    C only:
    - `execute.c`: source code of the program in the code cell with lines to facilitate answer checking inserted. An example `execute.c` file, from the question `min_io_forward_0` .
    - `execute`: a binary executable compiled from `execute.c`
2. Bullseye: A Bullseye question has the same code library files as Input-output except `stdout.txt`
3. Liar-liar: A Liar-liar question has the same code library files as Input-output except `stdout.txt`
4. Find-the-failure: A Find-the-failure question has the same code library files as Input-output except `stdout.txt` . In addition, a faulty version of the source code, and if applicable, an executable compiled from the faulty version source code are included.
    - `faulty.sh`: a shell script that executes the faulty version of the program with the contents of the argv cell. An example `faulty.sh`, from the question `py_min_ff_0` .

    Python only:
    - faulty.py: the faulty version of the Python program as in the code cell with lines to facilitate answer checking inserted. An example `faulty.py`, from the question `py_min_ff_0` . Compare `execute.py`, from the same question.

    C only:
    - `faulty.c`: the source code of the program in the code cell of a Find-the-failure question with lines to facilitate answer checking inserted.
    - `faulty`: executable compiled from `faulty.c`

3.2 Where is the code library?

Generated questions are stored in subdirectories of `code_library/` . Multiple question directories can be generated from a single template file.

3.3 How do you write a question template?

A template file is a python module with the following variables, their use described for each question type:

1. Input-output:
    - Question specification variables:
        - `game_type`: a string that indicates the question type. Used to generate `types.py` . The following are the legal values:
            - `'input_output'`
            - `'bullseye'`
            - `'liar_liar'`
            - `'find_the_failure'`
        - `parameter_list`: a nested list of text field marker and text field type pairs for declaring text field markers. The allowed types are `int`, `float`, and `string`. Used to generate `types.py` .
        - `tuple_list`: a list of question groups. The first element of a question group is a group prefix string followed by tuples. Each tuple corresponds to a question and the tuple elements are the values of the text field markers. Each tuple in a question group is used to generate a code library directory named with the group prefix followed by the tuple's order number, starting at 0.
    - Template variables are strings containing mix of fixed text and the text field markers:
        - `global_code_template` and `main_code_template`: contain the global code and the main function code of the program in the question. Each line in the templates starts with a prefix followed by a tab. The allowed prefix letters are:
            - `d`: this line should be present in `display.txt`
            - `x`: this line should be present in the source file (e.g. `execute.c`)

          Combinations of prefix letters are allowed, and the most common prefix is '`dx`', indicating that the line is in both `display.txt` and the source file.
        - `argv_template`, `stdin_template`, and `stdout_template`: the contents templates of the argv cell, stdin cell, and stdout cell. Used to generate `argvs.txt`, `stdin.txt`, and `stdout.txt`, respectively.

    An example Input/output question template.
2. Bullseye: In addition to the markers allowed for Input-output, text field marker type `target` is allowed in the `parameter_list` . The corresponding tuple element in the `tuple_list` has only the value `True` or `False`. When generating display.txt, the `target` type text field markers in `global_code_template` and `main_code_template` are replaced with a highlight marker (the string '`ca_highlight`') if the corresponding tuple element is `True` and removed otherwise. When the question is displayed, this highlight marker is further replaced by HTML code that changes the text colour of the line. An example Bullseye question template.
3. Liar-liar: In addition to the markers allowed for Input-output template file, a text field marker in the `parameter_list` can be the `assert` type that has a string containing an expression or an

empty string in the corresponding tuple element in the `tuple_list`. When generating `display.txt`, the `assert` type text field markers in the `global_code_template` and `main_code_template` are replaced with a highlight marker and an `assert()` containing the expression from that tuple element. If the corresponding tuple element is the empty string, the marker is removed. An example Liar-liar question template.

4. Find-the-failure: The allowed prefixes at each line in `global_code_template` and `main_code_template` are `d`, `x`, `X`, and any combination of them. The `X` prefix letter indicates that the line should be present in the faulty source file. The correct lines follow and have only the prefix '`x`'. An example Find-the-failure question template.

3.4 How do you run the question template verifier?

The `template_verifier.py` script in the `generate/` directory takes the name of a template file as its only argument. The verifier prints error messages for common problems in a template file. If the given template file is well formed, the verifier produces no output.

For example, to verify a template called `example.py`, run:

```
python template_verifier.py example.py
```

3.5 How do you run the generator script?

To build a question template file, run `generate.py` with two arguments: the template file, and the destination code library. For example, to build questions from a template file named `example.py` into the standard code library directory, run:

```
python generate.py example.py ../code_library
```

4. Quiz Specification

A quiz is defined by a single Python file, which is normally stored in the `quiz_specs/` directory. A quiz specification consists of a number of variable definitions.

4.1 What variables define a quiz specification?

- `code_lib_path`: the path to the code library (the directory containing questions). Normally `'../code_library/'`. The trailing slash is is optional.
- `question_list`: a list of tuples. Each tuple corresponds to one or more questions in the quiz. Each tuple has three components:
  - `mark`: an integer that specifies how many points each question from this tuple is worth.
  - `count`: an integer that specifies how many questions should be randomly chosen from this tuple's list of directories, i.e., how many questions this tuple will create.
  - `directories`: a list of one or more directory names or patterns (globbing patterns containing `*`, `?`, and character ranges, e.g., `[0-9]`, are valid). These patterns should match question directories present in the code library specified by `code_lib_path` .
- `practice_mode`: a boolean. If `True`, the quiz does not count for marks. Other differences between normal and practice mode are explained below.
- `standalone`: a boolean. If `True`, the quiz is a standalone question. The full details of standalone

mode are explained below.

- `logged`: a boolean. Indicates whether or not the quiz should be logged. A quiz can only be logged if:
  - `log_dir` is a valid directory and the server has permission to write into it.
  - The student taking the quiz has logged in with some username.
- `log_dir`: the directory into which log files should be written, for example, `'quiz_logs/'` . Can be the empty string (`''`) if `logged` is set to `False` .

4.2 Example quiz specification file

An example of a valid quiz specification file. The first question will be any one of the directories matched by the pattern `'min_io_forward_*'`. The second question will be one of `min_bull_1` , `min_bull_2` , or `min_bull_4` , but not `min_bull_3` . The quiz is in practice mode and its results will not be logged.

4.3 Quiz specification verifier

There is a script that can check quiz specification files for common errors, including a check to make sure the patterns and directory names given in each tuple's `directories` list exist. It takes the name of the quiz specification to be checked as its only argument.

For example, to check a quiz specification file called `example.py`, run:

```
python qspec_verifier.py example.py
```

4.4 What are the differences between the different quiz modes?

Practice = true vs. false

Setting `practice_mode = True` causes the following changes to a quiz:

- The quiz can be played anonymously (non-practice quizzes require that a student be logged in, so that marks can be recorded).
- The student is informed whether answers are correct or incorrect.
- The "Submit Answer" button is labelled "Check Answer" instead.

Standalone = true vs. false

A standalone quiz comprises only one question. In a standalone quiz the "Previous" and "Next" buttons are hidden, and only the first tuple from `question_list` is used; subsequent tuples (if any are present) are ignored.

Note that the question displayed in a standalone quiz is only definite if there is a single entry in the `directories` list — if there are multiple directories (or a globbing pattern expands to multiple directory names), the quiz's only question is chosen randomly from the `directories` list.

5. Log files and analysis

5.1 What tools are there for analysing log files?

There is a tool, `log_analysis.py`, for processing a directory of quiz log files into a tab-separated list of student usernames and their score for that quiz. It takes the log directory as its only argument. For example, to print a list of student marks for a log directory called `example_quiz/`, run:

```
python log_analysis.py example_quiz/
```

6. Server management

6.1 How do you start/stop server?

To start the server, `cd` into the `web2py/` directory and run `./start.sh` .

Listening on loopback only

By default, the server listens on `127.0.0.1` (localhost) on port `8080`. To ensure that this is the case, inspect the last line of `start.sh` . It should read:

```
python web2py.py -i 127.0.0.1 -p 8080 -a password
```

Listening on other IP addresses and ports

To set the IP address and port that the server listens on, change the `-i` and `-p` parameters, respectively, on the last line of `start.sh` . The `-a` parameter is the web2py administrator password.

Example

For example, to listen on the IP address `192.168.2.55` on port `80`, with `s3cret` as the web2py administrator password, change the last line of `start.sh` to:

```
python web2py.py -i 192.168.2.55 -p 80 -a s3cret
```

6.2 How do you install and access a plain HTML page P?

Plain HTML pages are HTML files that should be served exactly as they are, with no special processing by the server.

What changes need to be made to P?

Links to other static content (other plain HTML pages, style sheets, images, etc.) need no special treatment: references to other files in `static/` are ordinary relative links.

Relative links to a particular quiz should have the form:

```
../ca_controller/quiz?quiz_spec=../quiz_specs/Q
```

where `Q` is the name of the quiz specification.

Where does P go?

Plain HTML pages go in the `web2py/applications/codeAct/static/` directory.

What is the URL of P?

Assuming the file's name is `P.html`, `http://localhost:8080/codeAct/static/P.html` .
(localhost:8080 should be replaced with the hostname and port your server is running on.)

6.3 How do you install and access a quiz with quiz specification Q?

Suppose a quiz specification file called `Q.py` :

Where does Q go?

Put `Q.py` in the `quiz_specs/` directory.

What is the URL of Q?

`http://localhost:8080/codeAct/ca_controller/quiz?quiz_spec=../quiz_specs/Q`

Question: should we modify ca_controller so that "../quiz_specs/" can be omitted from the above URL? It is the same for all quiz specs.

7. Appendix

The webmaster may also be interested in learning more about the web2py framework, on which Code Activator is written. Web2py documentation is available from the official web2py book. Of particular interest is Chapter 4, which details the API.