COMP226 Assignment 1

Continuous Assessment Number	1 (of 2)
Weighting	15%
Assignment Circulated	Tuesday 22 February 2022
Deadline	17:00 Tuesday 15 March 2022
Submission Mode	Submit a single R file "solution.R" to the CodeGrade Assignment on Canvas
Learning Outcomes Assessed	Have an understanding of market microstructure and its impact on trading.
Goal of Assignment	Reconstruct a limit order book from order messages; compute quantities based on the limit order book
Marking Criteria	Pre-deadline visible CodeGrade tests of correctness of 6 functions (70%); Post-deadline CodeGrade tests of 4 further functions (30%)
Submission necessary in order to satisfy module requirements	No
Late Submission Penalty	Standard UoL policy; **resubmissions after the deadline may not be considered**.
Expected time taken	Roughly 8-12 hours

Warning

Submissions are automatically put through a plagiarism and collusion detection system. Students found to have plagiarized or colluded will likely receive a mark of zero. Do not discuss or show your work to others, and do not search for solutions to the assignment online. In previous years, students have had their studies terminated and left without a degree because of plagiarism or collusion.

Rscript from Rstudio

In this assigment, we use `Rscript` (which is provided by R) to run our code, e.g.,

```
Rscript main.R template.R input/book_1.csv input/empty.txt
```

In R studio, you can call Rscript from the "terminal" tab (as opposed to the "console"). On Windows, use Rscript.exe not Rscript:

```
Rscript.exe main.R template.R input/book_1.csv input/empty.txt
```

Distributed code and sample input and output data

As a first step, please download `comp226_a1.zip` via the assignment page on Canvas. Then unzip `comp226_a1.zip`, which will yield the following contents in the directory `comp226_a1`:

```
comp226_a1
├── common.R
├── input
│   ├── book_1.csv
│   ├── book_2.csv
│   ├── book_3.csv
│   ├── empty.txt
│   ├── message_a.txt
│   ├── message_ar.txt
│   ├── message_arc.txt
│   ├── message_ex_add.txt
│   ├── message_ex_cross.txt
│   ├── message_ex_reduce.txt
│   └── message_ex_same_price.txt
├── main.R
├── output
│   ├── book_1-message_a.out
│   ├── book_1-message_ar.out
│   ├── book_1-message_arc.out
│   ├── book_2-message_a.out
│   ├── book_2-message_ar.out
│   ├── book_2-message_arc.out
│   ├── book_3-message_a.out
│   ├── book_3-message_ar.out
│   └── book_3-message_arc.out
└── template.R

2 directories, 23 files
```

Brief summary

You are provided with three `.R` files, **two complete, which should not be edited**:

- `main.R` is the file you will run, e.g. with `Rscript`, by specifying several command line arguments described below (see example above);
- `common.R` contains complete working functions that are used by `main.R` in conjunction with the incomplete functions in `template.R`;

and **one incomplete file**:

- `template.R` is the file that you will edit -- the distributed version contains empty functions. It contains 10 empty functions.

If you run `main.R` using `template.R` as it is distributed, it runs without error, but does not produce the desired output because the first 6 functions in `template.R` are provided empty. To get 70%, you will need to correctly complete these 6 functions; if your answer is only partially correct you will get a mark less than 70%. If you have correctly completed all these 6 functions, you can then -- and only then -- get marks for the 4 "extra" functions, which together account for 30% of the marks.

You should submit a single R file that contains your implementation of some or all of these 10 functions. Your submission will be marked via extensive **automated tests of correctness** across a wide range of example cases, with some chosen randomly at test time:

- The tests for the first 6 functions, which give up to 70% of the marks will run at the time of submission and are fully visible **pre-deadline** on CodeGrade (detailed guidance on using CodeGrade to improve your mark can be found at the end of this document);
- The tests for the final 3 functions will only run **post-deadline**, and **only if you got full marks for the first 6 functions**.

You can (and if required should) submit multiple times to repeatedly use the CodeGrade pre-deadline tests; for a correct solution CodeGrade will show that you pass all tests and have thus achieved the first 70% of marks. It probably does not make sense for you to work much on the final 4 functions until you have achieved this and submitted completely correct versions of the first 6 functions.

In addition to the visible pre-deadline tests on CodeGrade, for the first 6 functions, correct sample output is provided so that you can check correctness of your implementations "offline" (without submitting to CodeGrade), for example with a tool like diff (https://en.wikipedia.org/wiki/Diff) to compare the output that you produce with the correct output. Offline testing is quick to do once you have set it up, and if you match all the offline examples then chances are that you will also pass the CodeGrade tests (but **make sure that you check this to avoid nasty surprises**).

template.R versus solution.R

Throughout the rest of this handout, we show example output from the incomplete template.R as well as using a full solution file "solution.R" that contains a correct implementation of all the functions. Obviously, you are not given the file solution.R, you need to create it from template.R.

The first 6 functions to implement

The first 6 functions, which are worth 70% of the marks, are broken down into two groups. The percentages in square brackets show the breakdown of the marks by function.

Order book stats:

1. `book.total_volume <- function(book)` **[5%]**
2. `book.best_prices <- function(book)` **[5%]**
3. `book.midprice <- function(book)` **[5%]**
4. `book.spread <- function(book)` **[5%]**

These first 4 functions are intentionally very easy, and are meant to as get you used to the format of `book`. The next 2 functions are more involved and relate to reconstructing the order book from an initial book and a file of messages.

Reconstructing the limit order book:

5. `book.reduce <- function(book, message)` **[15%]**
6. `book.add <- function(book, message)` **[35%]**

Running main.R with template.R

An example of calling main.R with template.R is as follows.

```
Rscript main.R template.R input/book_1.csv input/empty.txt
```

As seen in this example, main.R takes as arguments the path to **three input files** (the order of the arguments matters):

1. an R file with the functon implementations (`template.R` in the example)
2. an initial order book (`input/book_1.csv` in the example)
3. order messages to be processed (`input/empty.txt` in the example)

Let's see the source code of `main.R` and the output that it produces.

```
options(warn=-1)
args <- commandArgs(trailingOnly = TRUE); nargs = length(args)
log <- (nargs == 4) # TRUE is there are exactly 4 arguments

arg_format <- "<--log> <solution_path> <book_path> <messages_path>"

if (nargs < 3 || nargs > 4)  # check that there are 3 or 4 arguments
    stop(paste("main.R has 3 required arguments and 1 optional flag:", arg_format))

if (nargs == 4 && args[1] != "--log") # if 4 check that --log is the first
    stop(paste("Bad arguments format, expected:", arg_format))

solution_path <- args[nargs-2]
book_path     <- args[nargs-1]
messages_path <- args[nargs]

if (!all(file.exists(c(solution_path, book_path, messages_path))))
    stop("File does not exist at path provided.")

source(solution_path); source("common.R") # source common.R from pwd

book <- book.load(book_path)
book <- book.reconstruct(data.load(messages_path), init=book, log=log)
book.summarise(book)
```

In short, `main.R`:

- checks that the command line arguments are ok
- assigns them to variables (`solution_path`, `data_path`, and `book_path` respectively)
- sources `common.R` and the file at `solution_path`; loads the initial book
- reconstructs the book according to the messages
- prints out the resulting book; prints out the book stats

Let's see the output for the example above:

```
$ Rscript main.R template.R input/book_1.csv input/empty.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100
```

```
Total volume:
Best prices:
Mid-price:
Spread:
```

Now let's see what the output would look like for a correct implementation:

```
$ Rscript main.R solution.R input/book_1.csv input/empty.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100

Total volume: 100 100
Best prices: 95 105
Mid-price: 100
Spread: 10
```

You will see that now the order book stats have been included in the output, because the four related functions that are empty in `template.R` have been implemented in `solution.R`.

The initial order book

Here is the contents of `input/book_1.csv`, one of 3 provided initial book examples:

```
oid,side,price,size
a,S,105,100
b,B,95,100
```

Let's justify the columns to help parse this input:

oid	side	price	size
a	S	105	100
b	B	95	100

The first row is a header row. Every subsequent row contains a limit order, which is described by the following fields:

- `oid` (order id) is stored in the book and used to process (partial) cancellations of orders that arise in "reduce" messages, described below;
- `side` identifies whether this is a bid ('B' for buy) or an ask ('S' for sell);
- `price` and `size` are self-explanatory.

Existing code in common.R will read in a file like `input/book_1.csv` and create the corresponding two (possibly empty) orders book as two data frames that will be stored in the list `book`, a version of which will be passed to all of the functions that you are required to implement.

Note that if we now change the message file to a non-empty one, `template.R` will produce the same output (since it doesn't use the messages; you need to write the code for functions 5 and 6 to process them):

```
$ Rscript main.R template.R input/book_1.csv input/message_a.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100

Total volume:
Best prices:
Mid-price:
Spread:
```

If correct message parsing and book updating is implemented, `book` would be updated according to `input/message_a.txt` to give the following output:

```
$ Rscript main.R solution.R input/book_1.csv input/message_a.txt
$ask
  oid price size
8   a   105  100
7   o   104  292
6   r   102  194
5   k    99   71
4   q    98  166
3   m    98   88
2   j    97  132
1   n    96  375

$bid
  oid price size
1   b    95  100
2   l    95   29
3   p    94   87
4   s    91  102

Total volume: 318 1418
Best prices: 95 96
Mid-price: 95.5
Spread: 1
```

Before we go into details on the message format and reconstructing the order book, let's discuss the first four functions that compute the book stats.

Computing limit order book stats

The first four of the functions that you need to implement compute limit order book stats, and can be developed and tested without parsing the order messages at all. In particular, you can develop and test the first four functions using an empty message file, `input/empty.txt`, as in the first example above.

The return values of the four functions should be as follows (where, as usual in R, single numbers are actually numeric vectors of length 1):

- `book.total_volumes` should return a list with two named elements, `bid`, the total volume in the bid book, and `ask`, the total volume in the ask book;

- `book.best_prices` should return a list with two named elements, `bid`, the best bid price, and `ask`, the best ask price;
- `book.midprice` should return the midprice of the book;
- `book.spread` should return the spread of the book.

Check that the example outputs above are what you expect them to be.

Reconstructing the order book from messages

We now move on to the reconstructing the order book from the messages in the input message file. You do not need to look into the details of the (fully implemented) functions `book.reconstruct` or `book.handle` in `common.R` that manage the reconstruction of the book from the starting initial book according to the messages (but you can if you want).

In the next section, we describe the **two types of message**, "Add" messages and "Reduce" messages. All you need to know to complete the assignment is that messages in the input file are processed in order, i.e., line by line, with "Add" messages passed to `book.add` and "Reduce" messages passed to `book.reduce`, along with the current `book` in both cases.

Message Format

The message file contains one message per line (terminated by a single linefeed character, '`\n`'), and each message is a series of fields separated by spaces. Here's an example, which contains an "Add" message followed by a "Reduce" message:

```
A c S 97 36
R a 50
```

An "Add" message looks like this:

```
'A' oid side price size
```

- '`A`': fixed string identifying this as an "Add" message;
- `oid`: "order id" used by subsequent "Reduce" messages;
- `side`: '`B`' for a buy order (a bid), and an '`S`' for a sell order (an ask);
- `price`: limit price of this order;
- `size`: size of this order.

A "Reduce" message looks like this:

```
'R' oid size
```

- '`R`': fixed string identifying this as a "Reduce" message;
- `oid`: "order id" identifies the order to be reduced;
- `size`: amount by which to reduce the size of the order (*not* its new size); if `size` is equal to or greater than the existing size, the order is removed from the book.

Processing messages

A "Reduce" message affects at most one existing order. An "Add" message will either:

- **not cross the spread** and then add a single row to the book (orders at the same price are stored separately to preserve their distinct "oid"s);

- **cross the spread** and in that case can affect any number of orders on the other side of the book (and may or may not result in a remaining limit order for residual volume).

The provided example message files are split into cases that include crosses and those that don't. This allows you to develop your code incrementally and test it on inputs of differing difficulty. We do an example of each case, one by one. In each example we start from `input/book_1.csv`; we only show this initial book in the first case.

Example of processing a reduce message

```
$ Rscript main.R solution.R input/book_1.csv input/empty.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100

Total volume: 100 100
Best prices: 95 105
Mid-price: 100
Spread: 10
```

```
$ cat input/message_ex_reduce.txt
R a 50
```

```
$ Rscript main.R solution.R input/book_1.csv input/message_ex_reduce.txt
$ask
  oid price size
1   a   105   50

$bid
  oid price size
1   b    95  100

Total volume: 100 50
Best prices: 95 105
Mid-price: 100
Spread: 10
```

Example of processing an add (non-crossing) message

```
$ cat input/message_ex_add.txt
A c S 97 36
```

```
$ Rscript main.R solution.R input/book_1.csv input/message_ex_add.txt
$ask
  oid price size
2   a   105  100
1   c    97   36
```

```
$bid
  oid price size
1   b    95  100

Total volume: 100 136
Best prices: 95 97
Mid-price: 96
Spread: 2
```

Example of processing a crossing add message

```
$ cat input/message_ex_cross.txt
A c B 106 101
```

```
$ Rscript main.R solution.R input/book_1.csv input/message_ex_cross.txt
$ask
[1] oid   price size
<0 rows> (or 0-length row.names)

$bid
  oid price size
1   c   106    1
2   b    95  100

Total volume: 101 0
Best prices: 106 NA
Mid-price: NA
Spread: NA
```

The NAs in the last example are the expected output for certain fields when at least one side of the book is empty and the corresponding quantity is undefined.

Sample output

We provide sample output for 9 cases, namely all combinations of the 3 initial books (book_1.csv, book_2.csv, book_3.csv) and 3 message files, all found in the `input` subdirectory. The 3 message files are called:

file	
messages_a.txt	add messages only, i.e., requires `book.add` but not `book.reduce`; for all three initial books, none of the messages cross the spreed
messages_ar.txt	add and reduce messages, but for the initial book `book_3.csv`, no add message crosses the spread
messages_arc.txt	add and reduce messages, with some adds that cross the spread for all three initial books

The 9 output files can be found in the `output` subdirectory of the `comp226_a1` directory.

```
output
├── book_1-message_a.out
├── book_1-message_ar.out
├── book_1-message_arc.out
├── book_2-message_a.out
├── book_2-message_ar.out
├── book_2-message_arc.out
├── book_3-message_a.out
├── book_3-message_ar.out
└── book_3-message_arc.out

0 directories, 9 files
```

Hints for order book stats

For `book.spread` and `book.midprice` a nice implementation would use `book.best_prices`.

Hints for `book.add` and `book.reduce`

A possible way to implement `book.add` and `book.reduce` that makes use of the different example message files is the following:

- First, do a partial implementation of `book.add`, namely implement add messages that do not cross. Check your implementation with `message_a.txt`.
- Next, implement `book.reduce` fully. Check your combined (partial) implementation of `book.add` and `book.reduce` with `message_ar.txt` and `book_3.csv` (only this combination with `message_ar.txt` has no crosses).
- Finally, complete the implementation of `book.add` to deal with crosses. Check your implementation with `message_arc.txt` and any initial book or with `message_ar.txt` and `book_1.csv` or `book_2.csv`.

Hint on `book.sort`

```
$ Rscript main.R solution.R input/book_1.csv input/message_ex_same_price.txt
$ask
  oid price size
2   j   105  132
1   a   105  100

$bid
  oid price size
1   b    95  100
2   k    95   71

Total volume: 171 232
Best prices: 95 105
Mid-price: 100
Spread: 10
```

Note that **earlier messages are closer to the top of the book**. This is due to **price-time precedence**, according to which orders are executed:

- Best price first, but **when two orders have the same price, the earlier one is executed first**

We provide `book.sort` that respects price-time precedence. It relies on the fact that the order ids increase as follows:

```
a < k < ab < ba
```

where < is indicating **"comes before"** in the message files.

```
book.sort <- function(book, sort_bid=T, sort_ask=T) {
    if (sort_ask && nrow(book$ask) >= 1) {
        book$ask <- book$ask[order(book$ask$price,
                                   nchar(book$ask$oid),
                                   book$ask$oid,
                                   decreasing=F),]
        row.names(book$ask) <- 1:nrow(book$ask)
    }

    if (sort_bid && nrow(book$bid) >= 1) {
        book$bid <- book$bid[order(-book$bid$price,
                                   nchar(book$bid$oid),
                                   book$bid$oid,
                                   decreasing=F),]
        row.names(book$bid) <- 1:nrow(book$bid)
    }

    book
}

book.init <- function() {
    book <- list(
         ask=data.frame(matrix(ncol=3, nrow=0)),
         bid=data.frame(matrix(ncol=3, nrow=0))
    )

    colnames(book$ask) <- c("oid", "price", "size")
    colnames(book$bid) <- c("oid", "price", "size")

    return(book)
}
```

This method will ensure that limit orders are sorted first by price and second by time of arrival (so that for two orders at the same price, the older one is nearer the top of the book). You are encouraged to use `book.sort` in your own implementations. In particular, by using it you can avoid having to find exactly where to place an order in the book.

Hint on using logging in `book.reconstruct`

In `common.R` a logging option has been added to `book.reconstruct`:

```
book.reconstruct <- function(data, init=NULL, log=F) {
    if (is.null(init)) init <- book.init()
    if (nrow(data) == 0) return(init)

    book <- Reduce(
        function(b, i) {
            new_book <- book.handle(b, data[i,])
            if (log) {
                cat("Step", i, "\n\n")
                book.summarise(new_book, with_stats=F)
                cat("====================\n\n")
            }
            new_book
        },
        1:nrow(data), init,
    )

    book.sort(book)
}
```

If you want to use this for debugging, you can turn it on with the `--log` flag, e.g.:

```
Rscript main.R --log solution.R input/book_1.csv input/message_arc.txt
```

Then `book.summarise` is used to give intermediate output after each message is processed.

Hint on `stringsAsFactors=FALSE`

Notice the use of `stringsAsFactors=FALSE` in `book.load` (and `data.load`) in `common.R`.

```
book.load <- function(path) {
    df <- read.table(
        path, fill=NA, stringsAsFactors=FALSE, header=TRUE, sep=','
    )

    book.sort(list(
        ask=df[df$side == "S", c("oid", "price", "size")],
        bid=df[df$side == "B", c("oid", "price", "size")]
    ))
}
```

Its use here is not optional, it is necessary and what ensures that the `oid` columns of `book$bid` and `book$ask` have type `character`.

It is crucial that you ensure that the types of your `oid` columns in your books remain `character` rather than `factors`. The following examples will explain the use of `stringsAsFactors` and help you to achieve this. First we introduce a function that will check the type of this column on different data frames that we will construct:

```
check <- function(df) {
    checks <- c("is.character(df$oid)",
                "is.factor(df$oid)")
    for (check in checks)
        cat(sprintf("%20s: %5s", check, eval(parse(text=check))), '\n')
}
```

Now we use this function to explore different cases. First we look at reading in a csv file.

```
> check(read.csv('input/book_1.csv'))
is.character(df$oid): FALSE
   is.factor(df$oid):  TRUE

> check(read.csv('input/book_1.csv', stringsAsFactors=FALSE))
is.character(df$oid):  TRUE
   is.factor(df$oid): FALSE
```

What about creating a `data.frame`?

```
> check(data.frame(oid="a", price=1))
is.character(df$oid): FALSE
   is.factor(df$oid):  TRUE

> check(data.frame(oid="a", price=1, stringsAsFactors=FALSE))
is.character(df$oid):  TRUE
   is.factor(df$oid): FALSE
```

What about using `rbind`?

```
> empty_df <- data.frame(oid=character(0), price=numeric(0))
> non_empty_df <- data.frame(oid="a", price=1, stringsAsFactors=FALSE)
```

```
> check(rbind(empty_df, data.frame(oid="a", price=1)))
is.character(df$oid): FALSE
   is.factor(df$oid):  TRUE

> check(rbind(empty_df, non_empty_df))
is.character(df$oid):  TRUE
   is.factor(df$oid): FALSE

> check(rbind(non_empty_df, data.frame(oid="a", price=1)))
is.character(df$oid):  TRUE
   is.factor(df$oid): FALSE
```

Note: when a `data.frame` becomes empty the type of the `oid` column is malleable and it is crucial to use `stringsAsFactors=FALSE`. We see this issue if we `rbind` a list with a `data.frame`.

```
> check(rbind(empty_df, list(oid="a", price=1)))
is.character(df$oid): FALSE
   is.factor(df$oid):  TRUE

> check(rbind(empty_df, list(oid="a", price=1), stringsAsFactors=FALSE))
is.character(df$oid):  TRUE
   is.factor(df$oid): FALSE

> check(rbind(non_empty_df, list(oid="a", price=1)))
is.character(df$oid):  TRUE
   is.factor(df$oid): FALSE
```

So it is crucial to use `stringsAsFactors=FALSE` when the `data.frame` is empty; I suggest to use it in every case.

Extra problems for final 30%

> ***Warning***
>
> The final 30% of marks are only available if CodeGrade gives you the full first 70% of marks. Please only focus on the extra problems once you have achieved this.

For these final 30%, there are **four functions** that you are asked to implement. You can get marks for any one of these independently. The marks available are as follows:

1. `book.extra1 <- function(book,size)` **[6%]**
2. `book.extra2 <- function(book,size)` **[6%]**
3. `book.extra3 <- function(book)` **[6%]**
4. `book.extra4 <- function(book,k)` **[12%]**

The functions are defined by a full specification, given below, but we intentionally do not give you explicit test cases (you need to create them for yourself if you want) and you cannot find out your mark before the deadline. We will also not offer detailed help on solving these parts, as they are meant to be more challenging, with part of that challenge being to solve them on your own.

If you have achieved the full first 70%, then you can get extra marks for fully or partially correct implementations of any of the 4 extra functions, independent of each other (e.g., you can complete `book.extra3` and not the other extra functions and still get marks).

The first three require you to compute an *expectation* of a discrete random variable. If you need a refresher, go back to COMP111 Introduction to Artificial Intelligence. For our applicaton such an expectation is just the average (since the probability distribution is uniform) over all the possible values of the random variable.

Assumption

For the extra tests, you can assume that bid book is not empty, so that the starting mid-price is only NA if the ask book has no orders in it.

Extra problem 1

`book.extra1` has two parameters, `book` (the order book), and `size`, which is an integer size between `1` and `M`, where `M` is the total volume in the ask book (you can assume that your function will only be tested on such values of `size`).

The function should return the expected value of the midprice after execution of a **buy** limit order with size `size` and price drawn uniformly at random from the **set** of prices in `book$ask`.

Extra problem 2

`book.extra2` has exactly the same two parameters as `book.extra1`. The function should return the expected value of the midprice after execution of a **buy** limit order with size `size` and price drawn uniformly at random from the integer prices (the tick size is 1) between the best ask price and the highest ask price in the book.

Hint: For the first 2 extra problems, if `size` is equal to `M` then the correct answer is NA.

Extra problem 3

`book.extra3` only takes `book` as an argument. The function should return the expected value of the midprice after execution of a **buy** market order with size `s` is executed, assuming that `s` is drawn uniformly at random from the set `{1,....,(M-1)}`, where `M` is the total volume in the ask book.

Hint: For first 3 extra problems, one unified approach is to simulate the relevant orders using functions that you implemented for `book.reconstruct`.

Extra problem 4

`book.extra4` has two parameters, `book` (the order book), and `k` a non-negative number that will be interpreted as a percentage, e.g., if `k=0.4` then `k` corresponds to 0.4%.

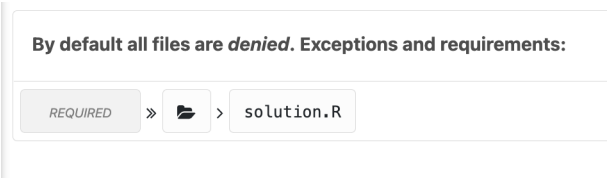
The function should return: 0 if the ask book has no orders in it; otherwise the largest amount of buy volume `v` such that a buy market order with size `v` causes the mid-price to increase by no more than `k` % (so an order with size `v+1` would either cause the midprice to increase by more than `k` % or would leave no asks left in the book which means that the mid-price is NA).

Note: the return value should be an integer between 0 and the total ask volume in `book`.

Hint: For all 4 extra problems, to increase the chance that your implementations are correct, do one or two examples where you compute the correct expectation by hand on and then check that your code produces the correct answer.

Submission

Submission is via CodeGrade on Canvas. Remember to call the file that you submit "solution.R", as this is the only thing that will be accepted by CodeGrade:

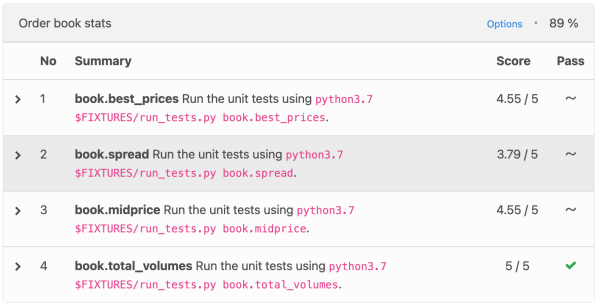


Using CodeGrade to improve your solution

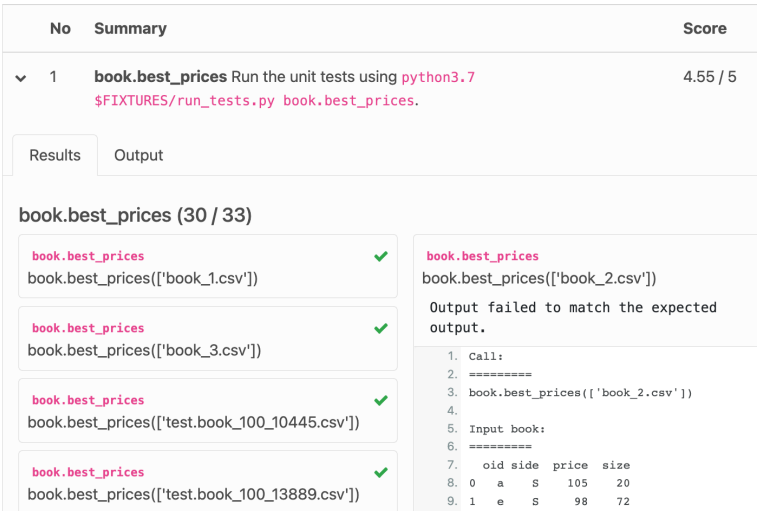
For the first 70% of marks, after you submit the tests runs and then:

1. a provisional mark is visible;
2. any tests that you fail are visible on CodeGrade and can be used to help you improve your solution.

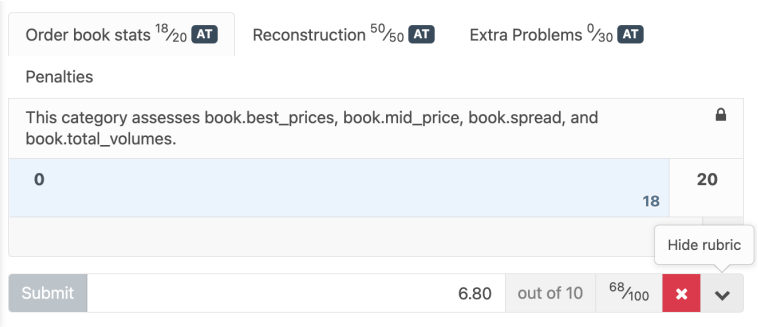
For example, I have edited the correct solution of `book.best_prices` to give the wrong answer 10% of the time at random and submitted this as a test student. After waiting a short while for all the tests to run, one can then see the result of the tests. Here they are for "Order book stats", for the first 4 functions:



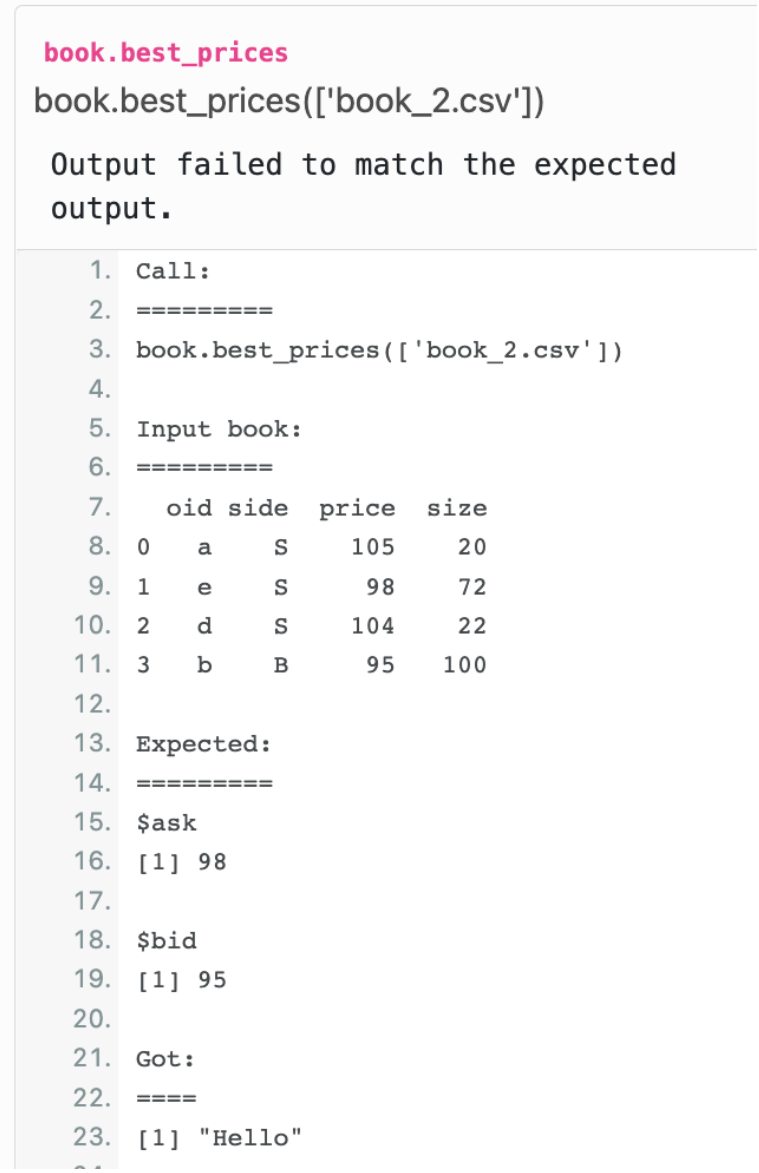
Not only is `book.best_prices` failing some tests, but so is `book.spread` and `book.midprice` because these happen to use `book.best_prices` in this implementation. Let's click on `book.best_prices` to inspect the individual tests:



Note that if you find that the area of the interface that shows the test results is too small, it may be because you are showing the rubric, which can be hidden (see the "Hide rubric" arrow in the bottom right of the screenshot):

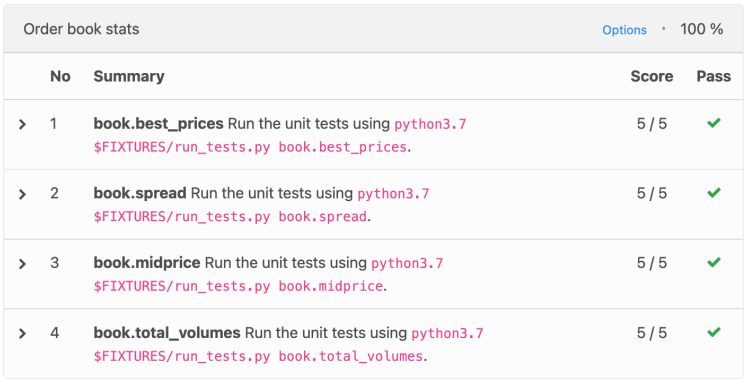


Let's zoom in on one of the failed tests for `book.best_prices`, which will allow us to see which inputs were used:

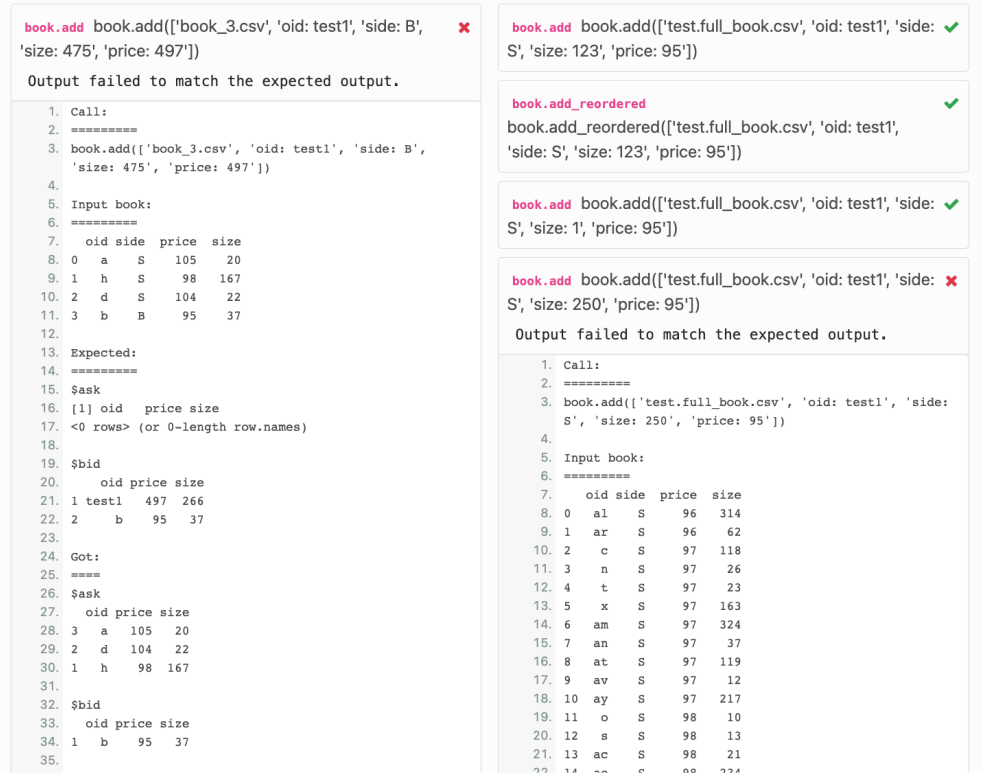


In this case, `book_2.csv` was used; the first few tests in each category use the examples books that you have been given, which makes it easy to investigate these failed tests.

When you have got all of the "Order book stats" functions correct and all the tests pass, it would look like this:

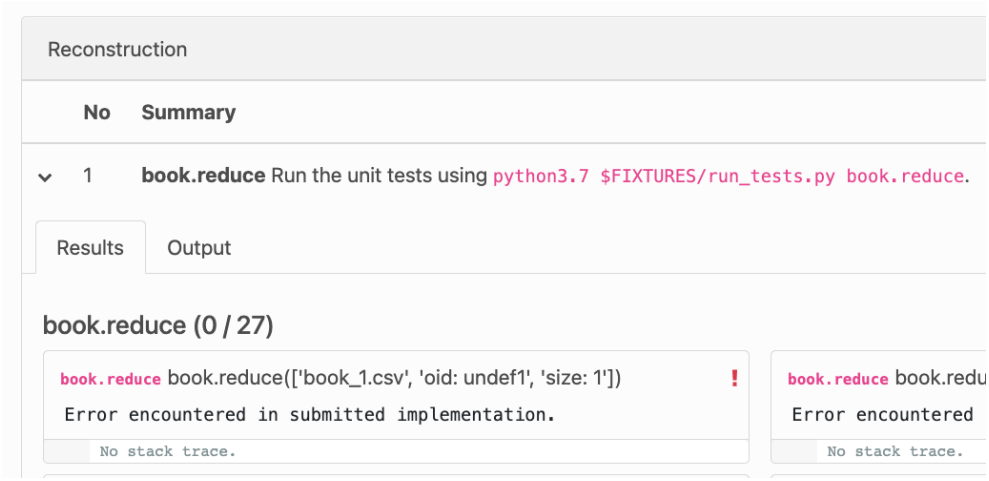


Let's do one more example, using `book.add`, where there is a message input as well as the book:

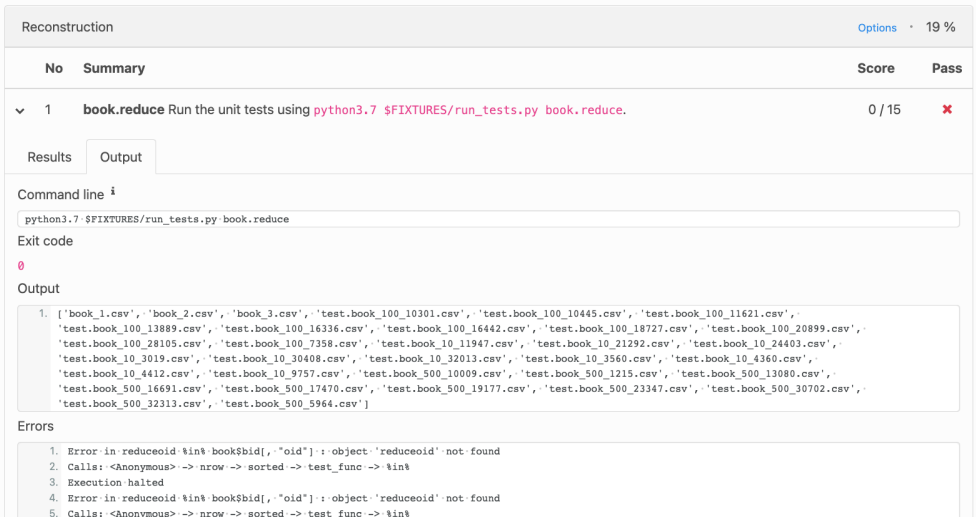


Finally, there is a second way that a test can fail. The above examples showed failures as "red crosses", meaning that output was generated but that it didn't match the expected output. A second way that a test can fail is with a "red exclamation mark":

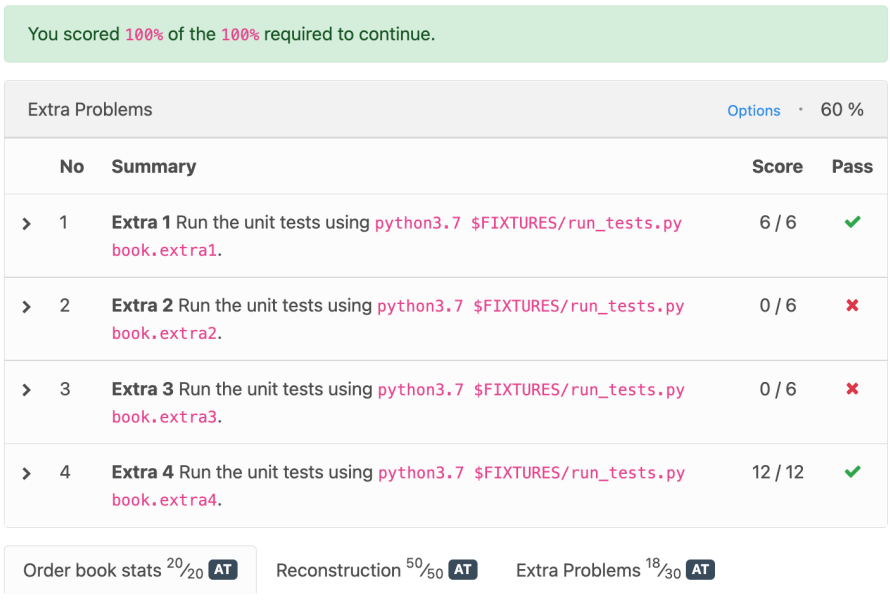
There are two tabs, "Results" and "Output". As a default, CodeGrade starts on the results tab, which is used in the examples above to see the details of tests that failed with a red cross:



To see any error messages that occured, which is relevant for tests that failed with a red exclamation mark, switch to the output tab:



Finally, a comment on the post-deadline tests. You will get similar test output for the extra functions, provided that you got the first 6 functions totally correct, **but only after the deadline** (this is completely intentional):



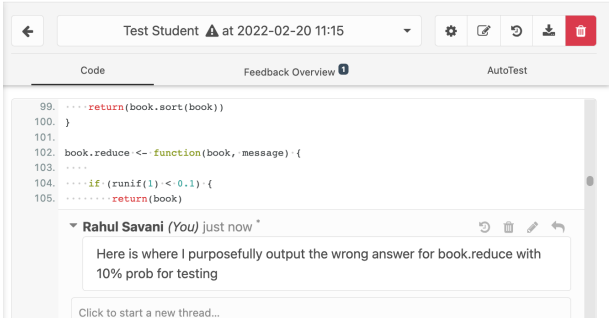
Tests

The first few tests in each category intentionally use book_1.csv, book_2.csv, and book_3.csv, so that it is easy for you investigate if a test fails (since you have these csv files, and can use the empty message file or create a simple message file yourself for testing).

Many of the tests use more complicated csv files, which are not available to you as csvs directly, but you could create csvs if you wanted. That should not be necessary in most cases though since if your code works on the example input/output files it should pass all the tests on CodeGrade.

Finally, please note that some tests are randomly generated. This is required to reduce the appeal of hardcoding answers (if there was a small number of non-random tests, students could just hardcode the expected answers rather than solve the general problem). This means that you may experience a small amount of variance in the mark you see when you resubmit the same wrong code (correct code will get full marks every time).

If you need help with the assignment we expect you to have submitted to CodeGrade. That way we can both see how your code does on the tests, which will help us to help you, and we can enter comments directly on the submitted source code, as in he following example:



Good luck with the assignment.

THE END