StepRepl Guide

Table of Contents

[Introduction 2](#_Toc105011539)

[Installation 2](#_Toc105011540)

[Starting StepRepl 2](#_Toc105011541)

[The command box 3](#_Toc105011542)

[Adjusting the font size 3](#_Toc105011543)

[Selecting a project 3](#_Toc105011544)

[Running code 4](#_Toc105011545)

[Running code from the command box 4](#_Toc105011546)

[Command output 4](#_Toc105011547)

[Scrolling output 4](#_Toc105011548)

[Stopping or pausing a running program 5](#_Toc105011549)

[Editing and reloading code 5](#_Toc105011550)

[Editor support 5](#_Toc105011551)

[Warnings 5](#_Toc105011552)

[Singleton variables 5](#_Toc105011553)

[Uncalled tasks 5](#_Toc105011554)

[Programmatic control of the UI 5](#_Toc105011555)

[Making buttons 6](#_Toc105011556)

[Hot keys 6](#_Toc105011557)

[Debugging tools 6](#_Toc105011558)

[Stack traces 6](#_Toc105011559)

[Reading a stack trace 6](#_Toc105011560)

[Showing variable values 7](#_Toc105011561)

[Breakpointing and single-stepping 7](#_Toc105011562)

[Unit testing 7](#_Toc105011563)

[Burn-in testing 8](#_Toc105011564)

[Sampling and statistical analysis 8](#_Toc105011565)

[Magic stuff **Error! Bookmark not defined.**](#_Toc105011566)

[Environment options **Error! Bookmark not defined.**](#_Toc105011567)

[Appendix: Key Bindings 9](#_Toc105011568)

[Appendix: StepRepl Commands 10](#_Toc105011569)

[Appendix: User-defined tasks with special meanings 10](#_Toc105011570)

[User interface 10](#_Toc105011571)

[Documentation 10](#_Toc105011572)

[Debugging 11](#_Toc105011573)

[Other 11](#_Toc105011574)

# Introduction

*StepRepl* is an interactive execution environment for *Step* code. It is a read-eval-print-loop (REPL), meaning you type commands, it runs them, and prints the results. It supports standard debugging tools such as breakpointing and single-stepping, as well as providing tools for profiling, sampling, and burn-in testing. This guide describes its capabilities and user-interface.

# Installation

To install StepRepl, download the appropriate binary for your operating system from the current github release. If you are running on MacOS, the binary will look like an application, and you can just run it. If you are running on Windows, it will look like a folder, and you will run the StepRepl.exe file inside the folder. In either case, you can place them wherever you like; they do not need to be in any special location.

**Important:** if you are running on MacOS, it will likely ask you whether to allow it access to your documents. If you do not allow it access, then MacOS will silently fail file operations and *Step* will be unable to access your projects.

# Starting StepRepl

To launch StepRepl simply open the application (MacOS) or run StepRepl.exe (Windows). You can exit either by clicking the close box or typing quit in the command box. You should see something like this:

Text

Description automatically generated

The green text box at the top is in the **command box**, where you can type code to run. Beneath it, you may see a **button bar** containing buttons to run code defined by the project you are running. Then at the bottom you will find two columns of text. The left column is where output from the text commands you run will appear. The right column (shown in yellow here) is where debugging information will be displayed (exceptions and stack traces) when debugging. The columns resize, so the system can use the full screen for text output when no debugging information is being displayed.

## The command box

The command box is the large green text box at the top of the screen. You can type any command here and it will execute it. The command can either be calls to step tasks or special commands to the StepRepl environment (see Appendix: StepRepl Commands).

## Adjusting the font size

As of this writing, Unity’s UI scaling is behaving strangely. The current system starts with a font size of 24 “points", which is readable for the standalone build, but may be smaller than you like. You can change the size by typing:

size *number*

in the command box. If 24 is too small, 30 is usually a good size. If you’re demonstrating in a classroom, students in back might prefer 40.

## Selecting a project

You cannot use StepRepl to enter new tasks and methods, only to call existing tasks and methods. These are loaded from your current project folder. Your project folder should be in either the Step folder or GitHub folder of your Documents folder. On Windows, this would be in one of:

C:\Users\*username*\Documents\Step\*projectName* C:\Users\*username*\Documents\GitHub\*projectName*

Or on MacOS, it would be in one of:

/Users/*username*/Documents/Step/*projectName*  
 /Users/*username*/Documents/GitHub/*projectName*

You can select a project directory by typing:

project *projectName*

in the command box. *StepRepl* will search the Step and GitHub folders for a subfolder named *projectName*. It will then load all .step and .csv files within that folder. If you modify the code, you can reload it by typing **control-R**.

# Running code

StepRepl is primarily an application to let you load and run Step code. It provides a number of ways to call tasks, and basic tools for debugging them.

## Running code from the command box

The most versatile way to run code is to type it in the green command box. You cannot type new methods here, but you can call any tasks defined in your project or built into *Step*. When typing a call to execute, the square brackets around it are optional. However when running multiple calls in one command, they must each be enclosed in brackets. Thus:

* Story  
  Runs the Story task with no arguments, assuming you’ve written such a task.
* [Story]  
  Does the same thing.
* Write “Hi there”  
  Runs the Write task with the argument “Hi there.”
* [Write “Hi there”]  
  Does the same thing.
* [Story] [Paragraph] The end.  
  Calls Story, then starts a new paragraph and prints “The end.”

## Command output

When you run a task, the system displays its output in the left column. If the call you performed contained local variables (local variables begin with ? characters), the system will also print the final values of those variables, in the form: *variable* = *value*. If the variable ended with a value (a number, string, tuple, etc.), then that’s what will be displayed for *value*. If it either had no value or was unified with some other variable, then *value* will be that variable.

*Step* prints variable names with uids (serial numbers) appended so you can distinguish between different variables with the same name. Thus a variable named ?x in the source might display as ?x3178.

If the code throws an uncaught exception, then the program will stop and StepRepl will show the exception followed by a Stack trace in the right column.

## Scrolling output

You can scroll the output using the **page up** and **page down** keys.

## Stopping or pausing a running program

You can kill the running Step program by pressing ESC. This will stop the program, showing it output (left column) and stack trace (right column) as of the time you stopped it.

You can pause the execution of the program without killing it by pressing the **pause** key. You can then read the stack trace and either abort, continue, or single step it.

# Editing and reloading code

StepRepl does not include a code editor. So if you need to change the code, edit it externally, then save the files and type **control-R** (or type **reload** in the command box). If there are errors or warnings, they will appear in the right (debug) column.

## Editor support

The preferred editing environment for Step is [Visual Studio Code](https://code.visualstudio.com/). Install VS Code, click the extensions manager in the toolbar on the left, and enter “Step syntax highlighting.” Find the step highlighting package and press Install. If you intend to use CSV files as well as .step files for your source code, you can either edit them with a spreadsheet program, or load one of the CSV editing extensions into VS Code.

# Warnings

After loading your code, *Step* runs a simple static analysis on it and prints warnings related to common errors. The common cases are listed here.

## Singleton variables

Since local variables aren’t explicitly declared in *Step*, a typo in a variable name will generally cause Step to treat it as a valid, but completely different variable. The calling card of such mistakes is a variable that is used only once, and so *Step* reports warnings for these so-called singleton variables.

As with other logic programming languages, *Step* uses a naming convention to override the warning. To override the warning, either change the name of the variable to either ? or place an underscore after the ? of the existing name, that is rename ?unused to ?\_unused.

## Uncalled tasks

Tasks that are defined by never called occur when:

* The task is an **entry point** to be called from outside the step code (e.g. by the user in *StepRepl* or by the game engine)
* A **typo** in the task name in the method in any calls to it
* The task is **dead code** that has become obsolete

The first of these is not an error, but the other two are. So Step issues a warning for uncalled tasks. You can override it by annotating the task with *[main]* in the source code.

# Programmatic control of the UI

*StepRepl* allows you to add items to the UI declaratively by including methods for the predicates Button and HotKey.

## Making buttons

To make a button, add a declaration of the form:

Button *label code*.

This tells the system to add a button with the specified label that runs the specified code when pressed. For example, the declaration:

Button “Make a story” [Story].

tells the system to include a button that says “Make a story” and that runs [Story] when you press it.

## Hot keys

To define a keyboard shortcut for running a particular code fragment, add a declaration of the form:

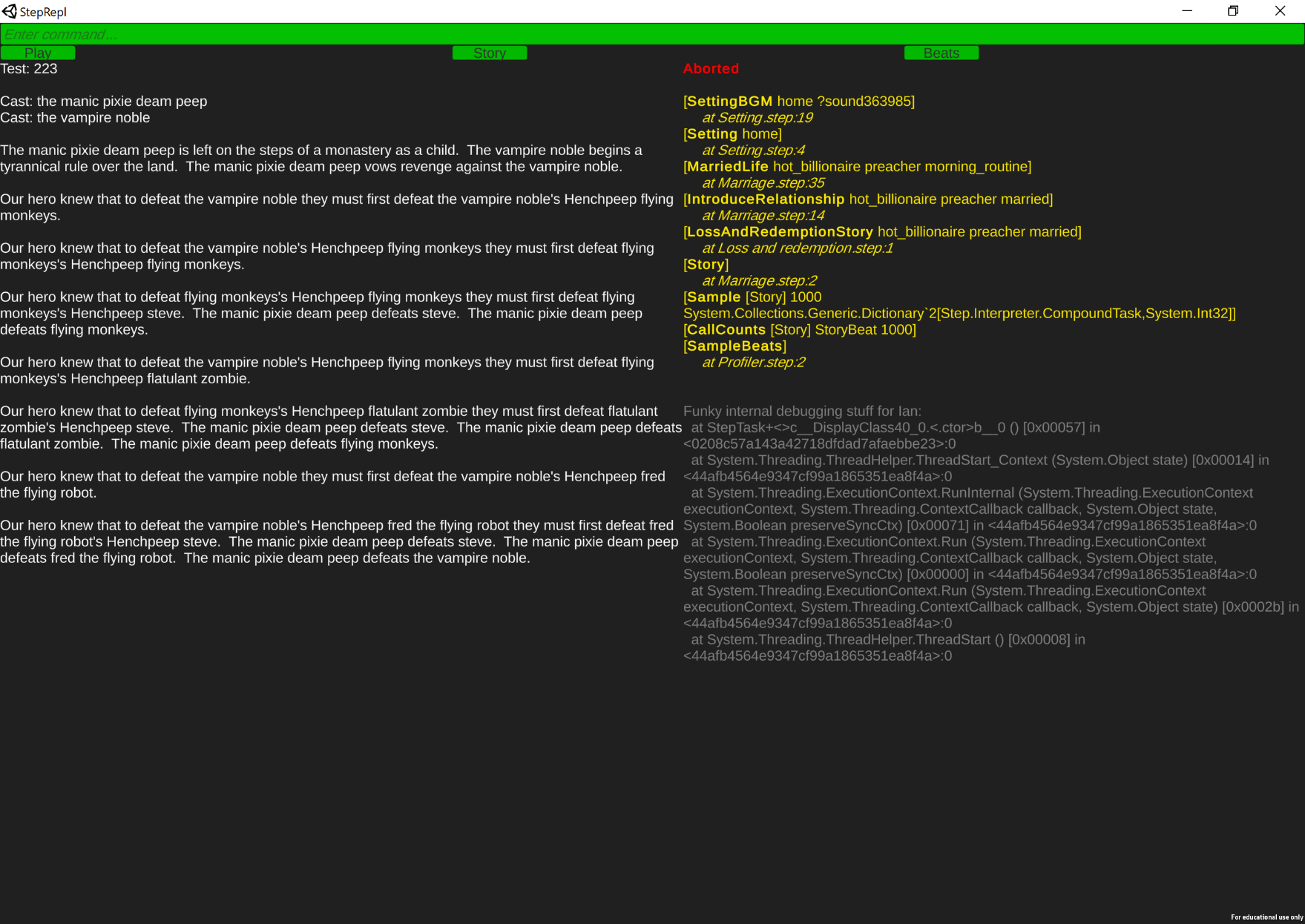
HotKey *key documentation code*.

This tells the system that when *key* is pressed, *code* should be run. Documentation is just text that is displayed when the system displays the available hotkeys.

# Debugging tools

*StepRepl* provides the standard kinds of debugging tools you would expect in a development environment.

## Stack traces

When a program pauses or fails, the system displays the currently running task, followed by the task that called it, its caller, and so on. This is known as a stack trace.[[1]](#footnote-1) Each task is shown as two lines. The first shows the call itself – the task name and arguments, and the second shows the file name and line number of the method that performed the call. For multiline methods, the line number shown is the line number of the start of the method, not of the specific call.

### Reading a stack trace

Each task is printed together with its arguments. Arguments are often local variables. As mentioned previously, when local variables are printed in the trace, a serial number is appended so that you can tell whether two different variables named ?x are the same ?x or different ones. If an argument is a local variable that has been unified with a value, that value is printed instead of the name of the variable. If you call [MyTask ?x], it will initially display in a trace as something like:

MyTask ?x259

Then, if ?x is unified with some other value, its appearance in the stack can change. If it is unified with another variable that doesn’t have a value, then the name of either variable can be displayed in the trace. If ?x is unified with a constant, or if some other variable unified with ?x is unified with a constant, then that constant will appear in the stack trace rather than the name ?x. Thus, the call my go from appearing in the trace as shown above to simply saying:

MyTask 7

If ?x had been unified with 7.

## Showing variable values

As mentioned already, the stack trace substitutes the values for local variables that have values. Any local variable in the running program will appear somewhere in the stack trace, and so you can determine the values of local variables from the trace.

However, global variables do not appear in the trace. To display the values of the global variables, press the **F4** key. This prints a listing of all global variables that have been given values

## Breakpointing and single-stepping

Although *StepRepl* doesn’t support source-level debugging, it does support the standard single-stepping operations found in most debuggers, using the same key bindings as Visual Studio:

* Step over (F10 key)  
  Run the call at the top of the stack to completion and paise
* Step Into (F11 key)  
  Step through the different calls within the current method
* Step Out (Shift-F11)  
  Run to completion of the caller of the top of stack
* Continue (F5 key or on-screen Continue button)  
  Allow the program to run without pausing

The program can be started in single-step mode by holding the **Control** key while pressing return in the command box. A running program can be paused either by pressing the **Pause** key. Breakpoints are signaled via the Break primitive task:

* [Break]  
  Pause execution at this point in the code. Display the stack trace and allow the programmer to either single-step or continue execution.
* [Break *message*]  
  Same, but *message* is displayed in the debugger when execution pauses.
* [Debug *code*]  
  Simply runs [Break] and then *code*.

Note that single stepping doesn’t presently stop for primitive tasks. That could be added if needed, but there hasn’t been a big call for it and it makes single-stepping even more laborious than usual.

## Unit testing

StepRepl include a simple-minded unit testing framework. More complicated systems can be built, but this will be sufficient for most users.

* TestCase *code*.  
  Include this declaration in your code to tell the system to run *code* as a unit test, when running unit tests.
* [RunTestCases]  
  Runs all the TestCases and displays any that fail.

## Burn-in testing

Since Step code typically makes extensive use of randomization, it is often useful to simply run a generation a few thousand times to make sure it doesn’t crash.

* *code* [Fail]  
  Runs code, but immediately rejects the solution, forcing it to backtrack and find the “next” solution. This continues until all solutions are generated, at which point, it fails completely. This will systematically generate every possible output of *code*, and then still fail. The point of this test is to ensure that no execution paths of *code* throw exceptions. If this fails rather than generating some other exception, then you know all possible execution paths are valid. This can also be included as a TestCase for automated testing. Note however that if code has many possible execution paths, it may take an unrealistically long time to execute. In addition, some recursive generators may have an infinite number of possible outputs, or at least an infinite number of execution paths. When this approach isn’t practical, try:
* [Test *code count*]  
  Runs *code*, *count* times. Each execution of *code* is independent of the previous ones. That is, it isn’t backtracking to try the “next” solution, it completely restarts the execution process. This only makes sense for *code* that involves randomization, since non-random *code* will always produce the same execution path. In addition, this will display the output of each iteration on the screen. It can be strangely satisfying to watch your code generate millions of outputs very quickly.

You can also invoke Test by holding **alt**/**option** while pressing return in the command box. This will cause the system to run Test on your command with *count*=10000.

## Sampling and statistical analysis

Complex random generators can behave very unpredictably. In particular, certain outputs or parts of outputs can unintentionally be generated too frequently or not at all. It can be useful to run the generator a large number of times and gather statistics on the derivations used by the system. This can be done using the Sample task:

* [Sample *code count ?sampling*]  
  Runs *code*, *count* times, using independent runs as with Test. For each run, it counts the number of times each user-defined task was used in that run’s final execution path. It then returns in *?sampling* a dictionary mapping tasks to the total number of times each task was used across all runs.
* [DisplayCallCount *sampling task divisor*]  
  Prints the number of times *task* was called in *sampling*, divided by *divisor*. To get absolute numbers, use *divisor*=1. To get averages per run, set *divisor* to the number of runs.
* [CallCounts *code taskPredicate count*]  
  Runs *code*, *count* times as above. Then displays the call counts for any tasks that satisfy *taskPredicate*. The taskPredicate is desirable since it is rare to want to know every task that is called. You usually have particular tasks that represent, e.g. plot points in a story generator, and you only want to know the statistics for those tasks. So you define a predicate over tasks that tells you whether a task implements a plot point, then just display the call counts for those tasks.
* [Uncalled *code taskPredicate count*]  
  Same as CallCounts, but only prints the tasks defined by *taskPredicate* that were never called in any run, or more properly, never appeared in the final execution path of any run.

# EnvironmentOption

The EnvironmentOption task allows you to control the behavior of the host system (*StepRepl* or your game) *Step* is running inside of. It provides a safe way of including calls in your *Step* code that are meaningful to one or another host system and are ignored when the code is run in other systems.

StepRepl supports the following environment options:

* [EnvironmentOption retainState]  
  Normally each command is run with a fresh, empty dynamic state, meaning that any state changes made by previous commands are undone for the next command, and all global variables returned to their initial values. The retainState option causes StepRepl to run each subsequent command using the end state of the previous command. This allows global state changes to accumulate across commands. State retention is disabled before loading a project, so as to prevent it being too sticky.
* [EnvironmentOption discardState]  
  This undoes the effect of retainState, returning the system to running commands from empty dynamic states.

# Appendix: Key Bindings

The following hotkeys are built into StepRepl:

* **F1**  
  Shows the help page
* **Page up**/**Page down**  
  Scrolls the output text and debug display, if any.

## Debugging

* **ESC**  
  Terminates the running command
* **Pause**  
  Pauses the running command as if it had hit a breakpoint
* **F4**Shows the values of all global variables not set to their default values.
* **F5**Continues execution of a paused/breakpointed/single-stepped command
* **F10**Step over
* **F11**Step into
* **Shift-F11**Step out
* **Control-R**  
  Reloads the project source files. The equivalent of the reload command.

## Modifiers for commands

The following modifiers keys can be used when pressing return/enter in the command box:

* **Control**  
  Runs [Break] *command* rather than just *command*, allowing you to step through the code.
* **Alt/option**Runs [Test *command*] rather than just *command*. This runs the command 10000 times to test different random execution paths.

# Appendix: StepRepl Commands

The following are the special commands that StepRepl will recognize in the command box:

* project *projectName*  
  Changes the current project folder whose code we are running.
* reload  
  Reloads the .step and .csv files from the current project.
* size *number*  
  Sets the font size. The current default is 32.
* quit  
  Exits the application.

# Appendix: User-defined tasks with special meanings

A number of tasks are consulted by the system if they are defined. All are optional, however. Projects do not need to define any of them.

## User interface

* Button *label code*.  
  Declaration telling *StepRepl* to display a button with the specified *label* text and that runs *code* when pressed.
* HotKey *key documentation code*.  
  Declaration telling *StepRepl* that when the user presses *key*, it should run *code*. *Documentation* is a string that is displayed as part of the documentation of the keybindings when the project is loaded.

## Documentation

If these tasks are defined, they are run by StepRepl after loading a project and their results are printed.

* Author: *text*.  
  States the identity of the author of the project.
* Description: *text*.  
  Description of the project.

## Debugging

* TestCase *code*.  
  States that *code* should run and succeed when unit testing, i.e. *code* is a unit test.

## Other

* [Mention *object*]  
  Called as part of the process of substituting variables into output text. This defaults to calling Write, if no methods are provided by the user. See the *Step tutorial* or *Step language guide* for more information.

1. Note for Prolog programmers: StepRepl displays the stack differently than languages such as Prolog. The actual low-level execution stack in a non-deterministic language includes not only the currently pending tasks, but also all the ones that have succeeded along the current execution path. Prolog displays everything in chronological order. This makes it very long and it makes it very hard to tell what called what. To the extent that what you usually want to know when reading a stack trace is what failed and what its caller was, I find it easier to just display that, as if the language were something more like C or Python. [↑](#footnote-ref-1)