An Open-Source, API-Based Framework for Assessing the Correctness of Code in CS50

Chad Sharp
University of Michigan
cmlsharp@umich.edu

Jelle van Assema
University of Amsterdam
jelle.van.assema@uva.nl

Brian Yu
Harvard University
brian@cs.harvard.edu

Kareem Zidane
Harvard University
kzidane@cs50.harvard.edu

David J. Malan
Harvard University
malan@harvard.edu

ABSTRACT

We present `check50`, an open-source, extensible tool for assessing the correctness of students' code that provides a simple, functional framework for writing checks as well as an easy-to-use API that abstracts away common tasks, among them compiling and running programs, providing their inputs, and checking their outputs. As a result, `check50` has allowed us to provide students with immediate feedback on their progress as they complete an assignment while also facilitating automatic and consistent grading, allowing teaching staff to spend more time giving tailored, qualitative feedback.

We have found, though, that since introducing `check50` in 2012 in CS50 at Harvard, students have begun to perceive the course's programming assignments as more time-consuming and difficult than in years past. We speculate that the feedback that `check50` provides prior to students' submission of each assignment has compelled students to spend more time debugging than they had in the past. At the same time, students' correctness scores are now higher than ever.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; **Student assessment**; • **Applied computing** → **Computer-assisted instruction**.

KEYWORDS

assessment, autograding, feedback

ACM Reference Format:
Chad Sharp, Jelle van Assema, Brian Yu, Kareem Zidane, and David J. Malan. 2020. An Open-Source, API-Based Framework for Assessing the Correctness of Code in CS50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20), June 15–19, 2020, Trondheim, Norway*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3341525.3387417

ITiCSE '20, June 15–19, 2020, Trondheim, Norway

ACM ISBN 978-1-4503-6874-2/20/06…$15.00
https://doi.org/10.1145/3341525.3387417

1 INTRODUCTION

We present an open-source, API-based framework for assessing correctness of code via functional testing, informed by our own years of experience with homegrown and third-party alternatives at Harvard in CS50, our introductory course for majors and non-majors alike. Implemented in Python, this framework provides teachers with an API via which to write *checks* (i.e., tests) and students with a command-line tool, `check50`, via which to execute those checks against their own code. The framework additionally provides an optional, containerized back-end environment in which students' code can be sandboxed, for cases where teachers prefer not to execute students' code locally. Via a JSON-based API can checks' results be incorporated into one's own learning management system (LMS) or other infrastructure.

Since its deployment in 2012, `check50` has been used by thousands of students on campus and tens of students online (via a MOOC). On campus, `check50` has not only equipped students with more-immediate feedback on work, it has also streamlined CS50's grading workflows, enabling teaching fellows (TFs) to spend more time on qualitative feedback for students. In the case of the MOOC, `check50` has enabled feedback itself for students off campus, albeit less personal.

However, the framework's deployment has not been without unforeseen side effects. Based on end-of-term surveys, students now report spending even more time on the course's programming assignments. That students now know, prior to submission, whether their work is correct, seems to be compelling even more effort. At the same time, we have found that some students have begun to rely too heavily on `check50`, effectively using it as their compiler (rather than, e.g., `clang` itself) in lieu of compiling or even testing their code themselves. We have thus begun to restrict the tool's use at term's start so that students have time to acquire some muscle memory. Similarly toward term's end have we begun to restrict use, instead advising students on how they might test their own code.

With that said, when students do have access to `check50`, upwards of 90% of students' submissions now receive perfect scores for correctness on the course's 5-point scale. To be fair, there might still be room for improvement in the submitted programs' design. And while there are occasionally cases that our own suites of checks do not catch initially, we iteratively eliminate those oversights each term as we augment our checks.

In the sections that follow, we elaborate on this framework's motivation, detail its design, reflect on results, and propose opportunities for next steps.

2 BACKGROUND

Our own workflow for assessment has evolved over time and has informed our design of `check50`. Years ago, we simply provided the course's TFs with a list of criteria against which to check students' code. That process, of course, was quite labor-intensive, with TFs manually compiling and executing students' code with prescribed inputs, after which they would record 0s and 1s in a spreadsheet. Quite quickly, some of the course's own TFs begin to automate that process with shell scripts, though each was homegrown for a particular assignment. Such "spaghetti" code too often proved unmaintainable year over year.

We considered standardizing on a unit-test framework (e.g., CUnit for assignments in C), though any such framework would require that we (or students themselves) instrument students' code for the particular framework. And for the course's earliest weeks, we were disinclined pedagogically to prescribe that students break up relatively small programs into even smaller functions solely for the sake of testing. That paradigm, we felt, could come later in the course.

Of particular interest, then, was to design a standardized framework via which the course's staff could automate assessment of correctness and feedback that would not impose conventions on students' own code. Similarly did we not want to impose a monolithic system or UI on fellow teachers who might adopt our framework themselves, and so another design goal was to ensure that all results could be extracted via simple API calls.

Indeed, we now use `check50` ourselves as just one component in our own grading pipeline. When students submit their work to a server, that triggers an automatic execution of `check50`, whose results are then stored, via an API call, in the course's own database. (Via a configuration file could those results be stored in any other teacher's database.) And an additional tool automatically lints students' code and provides feedback on style.

3 RELATED WORK

Courses in programming have long automated assessment thereof. Many automatic assessment systems exist and new ones are continuously created [1, 3, 8], not only for operational efficiency but educational value as well. Edwards [5], for instance, advocates for test-driven development, whereby students submit tests of their own along with their code. Of course, those tests should be, ideally, simple to write. Or, better yet, unnecessary to write.

Singh et al. [15], for instance, offer a tool for automated generation of feedback based on reference implementations of problems. Piech et al. [12] apply machine learning toward the same. And Dewey et al. [4] propose how to evaluate the efficacy of hand-generated tests.

Although other open-source (and commercial) frameworks exist for automated testing of students' code, among our goals in developing `check50` was to optimize the pedagogical usefulness of the system for less-comfortable students. Gradescope [6] and OK [11], two of the more popular code-checking systems that support the writing of arbitrary test code, display relatively limited information when student's code fails to pass tests. Indeed, when students' code fails to pass a test case, students are generally presented only with the assertion error that triggered the failure.

CodeLab by Turing's Craft [2], meanwhile, analyzes both the syntax of students' submissions via lexical analysis and the runtime behavior of students' code. It then offers suggestions and rhetorical questions in order to advise students on how to achieve desired behavior, not unlike `check50`'s own approach.

Vocareum [16], a service that offers grading automation as part of its learning management system, supports a simple user interface for generating tests that check for expected output and expected exit status, also allowing instructors to write arbitrary tests via their own grading scripts.

Autolab [13], meanwhile, offers functionality similar to `check50` but with the requirement of a back end.

A limitation of some of these tools, meanwhile, including Gradescope and OK, is that tests run independently, without support for some tests depending upon others. For several of our use cases in CS50, we found it valuable for some checks to depend upon the success and potential side effects of previously run checks. Accordingly, students can focus on just a limited set of failed checks rather than those that failed only because their dependencies did. For instance, in some cases, we wanted a check first to assess whether a student's code compiled and another check to assess some other behavior that's dependent on the code having passed that first check.

We also drew inspiration for the design of checks in `check50` from Python's built-in `unittest` framework, ultimately deciding to implement checks as Python methods, to use docstrings as descriptions of the functionality being tested, and to support "skipped" checks (that only run if their dependencies pass).

Unlike most other tools, `check50`'s own dependencies are minimal, and the tool itself can run locally. All that's required is a Python interpreter and Git. Teachers can write checks to test programs written in any language, so long as a compiler or interpreter for that language is installed locally as well.

Of course, a downside of any framework that uses human-written test cases, is the difficulty and human cost of writing test cases that cover a wide-enough spectrum of errors. To support easy updates, then, checks for `check50` can be hosted online on any repository on GitHub. The result is a collaborative workflow where teachers, TFs, and perhaps even students can independently create issues and submit contributions to the checks.

4 IMPLEMENTATION

In their systematic literature review, Ihantola et al. [8] describe eight features of automatic assessment tools. In Table 1, we list a description of `check50`'s support for those features.

In order to check their code for correctness, students simply run `check50` in the directory containing their code, passing as a command-line argument a unique *slug* (i.e., string) corresponding to the problem on which they are working. For example, a student taking CS50 at Harvard in Fall 2019 who was currently working on a problem called `cash` would have executed `check50` per Figure 1. The smile indicates that a check has passed, whereas the frown indicates that a check has failed.

All of `check50`'s checks are hosted on GitHub, and the slug serves to uniquely identify the repository and path in which checks are located. Adding checks to `check50` is thus as simple as adding to or creating a new GitHub repository. Indeed, `check50` expects a slug

Feature	check50
Programming Languages	Teachers can write checks to test programs written in any language, so long as a compiler or interpreter for that language is installed locally as well.
Learning Management Systems (LMS)	Via a JSON-based API can checks' results be incorporated into one's LMS.
Defining Tests	Checks are written by humans and hosted on GitHub.
Possibility for Manual Assessment	There is no support for manual assessment within the framework. That feature is left to the consumer of the framework's API.
Resubmissions	There are no limitations on resubmissions.
Sandboxing	The framework provides an optional, containerized back-end environment in which students' code can be sandboxed.
Distribution and Availability	check50 is available as open source at https://github.com/cs50/check50. Use of the tool, including its remote component, does not require approval from the authors.
Specialty	The framework can be extended independently from the main project with any pip-installable package, thereby allowing others to add support for specialty use cases, such as SQL, web programming, and GUIs, without needing to change existing code.

Table 1: check50's support for the features described by Ihantola et al. [8].

```
~/ $ check50 cs50/problems/2019/fall/cash
:) cash exists
:) cash compiles
:) input of 0.41 yields output of 4
:) input of 0.01 yields output of 1
:) input of 0.15 yields output of 2
:) input of 1.6 yields output of 7
:) input of 23 yields output of 92
:( input of 4.2 yields output of 18
    expected "18\n", not "22\n"
    did you forget to round your input to the nearest cent?
:) rejects a negative input like -1
:) rejects a non-numeric input of "foo"
:) rejects a non-numeric input of ""
```

Figure 1: Sample execution of check50 for a problem called cash in Fall 2019 of CS50.

to be formatted as `:organization/:repository/:branch/:path`, as in Figure 1. Even though a check's branch and path may contain arbitrarily many slashes, Git and, by extension, GitHub do not allow a repository to contain a branch that is a subpath of another. So each slug necessarily maps uniquely to a repository, branch, and path on GitHub.

check50 is implemented in Python, which itself is increasingly used in introductory programming courses [9]. Installation of check50 on a system with Python installed is as simple as installing any Python package: `pip install check50`. Checks themselves are implemented in Python, as in Figure 2.

Once installed, check50 can operate in two modes: local or remote. The default is remote, but if teachers or students wish to run check50 locally (e.g., for privacy concerns or lack of an internet connection), it suffices to add `--local` as a command-line argument. The behavior of check50 is the same in both modes; any check that passes or fails will pass or fail independent of the mode selected.

check50's remote mode is implemented by running check50 within a cloud-based container. This implementation detail is abstracted away, though, via the optional back end. To use check50

```
import check50
import check50.c

@check50.check()
def exists():
    """hello.c exists."""
    check50.exists("hello.c")

@check50.check(exists)
def compiles():
    """hello.c compiles."""
    check50.c.compile("hello.c")

@check50.check(compiles)
def world():
    """responds to stdin."""
    check50.run("./hello").stdin("world")
        .stdout("hello, world").exit()
```

Figure 2: An example of a check written in check50's Python format.

remotely, a student need only register for a (free) GitHub account and join an "organization" that's been set up by a course or university.

A principal design goal of check50 is to optimize the pedagogical usefulness of the system for less-comfortable students. In part, we achieve such by allowing teachers to control the output of check50, not only by defining a description for each check, but also defining, in advance, helpful hints for common mistakes, as in Figure 1. A common mistake that students make in CS50's cash problem, for instance, is not correctly rounding floating-point numbers, which results in an off-by-one error. check50 allows a check writer to add additional, case-specific help messages when a check fails, thereby empowering teachers to give helpful hints at just the right moment.

5 CHALLENGES

Implementation of check50 was not without challenges.

5.1 Performance

By default, check50 ships with a simple API that facilitates starting processes, sending standard input to them, and checking their standard output and exit code. Of course, this style of black-box testing results in a significant portion of time spent waiting on I/O. To combat such, check50 runs checks in parallel by default, in contrast with frameworks like unittest, pytest, and nose2, which do not support parallel tests natively but could through third-party extensions.

Inspired by the implementation of functional programming such as Haskell, check50 parallelizes the checks automatically where possible, albeit with the constraint that checks in check50 should be written as pure functions without side effects.

Of course, side effects are sometimes warranted. For instance, it's convenient to compile code once in one check and then pass the compiled program to other checks for functional testing. And so check50 indeed allows checks to depend on one other check, with the dependent check inheriting (a copy of) the other check's side effects. Parallel checking can also be disabled as needed.

5.2 Varying Technical Audiences

Another challenge that we faced when designing check50 was the varying technical comfort of teachers (i.e., prospective check writers). We consider our use of Python for check50 an implementation detail justified by its current popularity [9]. (And checks themselves can be written, in Python, to test programs written in any language.) We do, however, acknowledge that not all teachers are comfortable with Python. As such, in similar spirit to Codevolve's and Vocareum's tests, check50 supports a simpler, if limited, method for writing checks. Checks can alternatively be implemented in YAML, a text format based on key-value pairs, as in Figure 3. These YAML-based checks are more limited by design: they only support running programs, sending them input, and checking their output and exit code.

With that said, to ease the potential transition for teachers from YAML to richer Python-based checks, and to ensure equivalent behavior, check50 actually compiles YAML checks to human-readable Python checks, which are themselves then executed. As a result, transitioning from YAML to Python is as simple as changing a configuration option.

```
check50:
  checks:
    - run: python hello.py # run `python hello.py`
      stdin: world # input "world"
     stdout: hello, world # expect output "hello, world"
      exit: 0 # expect exit code 0
```

Figure 3: An example check written in check50's YAML format.

5.3 Extensibility

To accommodate other use cases, check50 itself can be extended with new features by simply installing additional Python packages via pip. check50 achieves such through a configuration option that takes a list of pip-installable dependencies, similar in spirit to Python's convention of including them in a file called requirements.txt. Such a list enables check50 to install all dependencies even when running remotely. Via this mechanism can check50 be extended with support for different programming languages or tools, without needing anyone's server-side approval or needing to change existing code.

For instance, at the University of Amsterdam, there is now a data science course that extended check50 with Jupyter Notebook. The extension automatically generates checks from test cells in Notebook and then runs all cells in a Jupyter kernel. The result is a lightweight alternative to nbgrader [7] for assessing code in notebooks, without the overhead of setting up a course in nbgrader or the need to host one's own server. The extension also enables students to add checks of their own that can then be tested by check50.

6 RESULTS

check50 has helped our course's staff improve both the efficiency and the consistency of grading of students' assignments. Whereas, previously, the course's staff would spend hours running a dozen or more tests manually on each of their students' submissions, check50's automation of the correctness-checking process has helped us ensure that the correctness of all students' code is evaluated on the same metrics. Moreover, it has freed up more human time for staff to spend on qualitative feedback and interactions with students.

However, when we first deployed check50 in 2012, we did observe that students began to spend more time on the course's assignments, so much so that they began to perceive them more negatively, based on end-of-term feedback. At the end of each term, students are asked to rate the course's assignments on a scale from 1 (unsatisfactory) to 5 (excellent). In 2011, the course's assignments averaged a score of 4.24, whereas in 2012, the assignments averaged a score of 3.86. (In both years did the course have hundreds of students.) We speculate that this change was, at least in part, the result of students perceiving assignments as more difficult. With check50, students were now empowered to check their own code before submission, discovering errors and corner cases that, in previous years, might have gone unnoticed. The course's assignments effectively became more difficult, but only insofar as students now knew, before submission, that their solutions were not yet quite right.

With that said, students in CS50 have taken significant advantage of the instantaneous feedback on code's correctness provided by check50 to verify that their code is, in fact, correct before submission, per Figure 4. In Fall 2017, for instance, when 10,951 submissions were made (by 671 students) for problems that supported check50 testing, 9,901 of those submissions—or 90% of the total—received perfect correctness scores. Another 458 submissions—or

4% of the total–received a score of 4 out of 5 on correctness, indicating nearly perfectly correct code, and fewer than 6% of submissions received a score of 3 or lower.

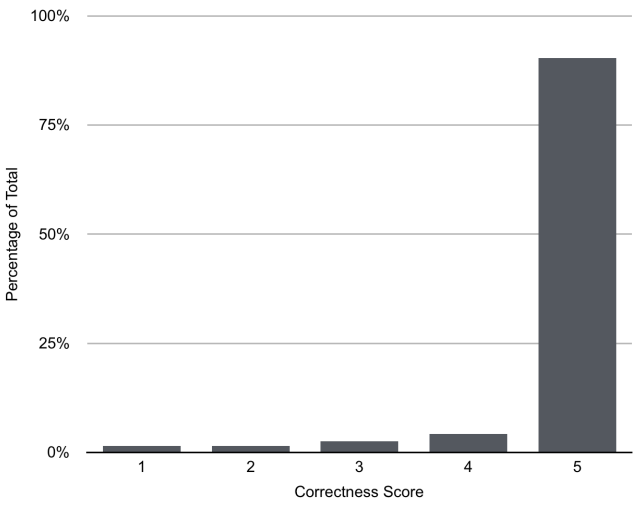

Figure 4: Distribution of correctness scores

However, even as we found that students were using `check50` to verify the correctness of their code before submission, we also observed that students were perhaps over-using `check50` because of its ease of use. Usage of `check50` averaged as few as 6.5 invocations on some problems and as many as 28.1 invocations on others. But during office hours (one-on-one opportunities for help for students), the course's staff also noticed that some students were frequently not even compiling their own code, choosing instead to run `check50` on their submission to see whether or not it would compile. For those students, `check50` unintentionally became a proxy of sorts for commands like `clang` (for C programs). Some staff members worried that, in providing such a readily available correctness-checking tool to students, students were failing to establish their own habit of compiling, running, testing, and debugging their own code for themselves.

To mitigate that risk, we have begun to restrict `check50`'s use at term's start, introducing it only after students have developed some muscle memory for compiling, running, and testing their code themselves. And we have also implemented support for "private" checks, which will enable us (and teachers more generally) to test submissions with checks that students do not themselves see, in order to motivate additional testing on their own.

7 FUTURE WORK

`check50` can be extended via any pip-installable package. However, there is no method to install arbitrary software, besides Python packages, in `check50`'s remote operation. In our testing, it proved too time-consuming to install other types of packages as each server-side container would have to download and re-install each package. Instead, we propose to add support for custom container images, giving complete control to teachers employing `check50` over the environment in which `check50` runs.

Currently, `check50` cannot handle data files that are larger than 2GB or heavy computing tasks that requires specialty hardware, both of which, we recognize, are common in data science classes. We propose, then, to add an ability to configure one's own back end for `check50`, effectively achieving extensibility in hardware as well.

Given our own experience, we do worry that the availability and ease of use of `check50` fosters all too much of a trial-and-error strategy for some students. We are encouraged to explore methods to have students establish their own habit of compiling, running, testing, and debugging. Indeed, others have tackled precisely this problem themselves. Edwards [5] suggests that students should submit tests along with their code, to foster a reflection-in-action strategy. In the Scheme-Robo automatic assessment system, Saikkonen et al. [14] purposefully lengthen the run time to ten minutes to encourage students to test their code before submitting. And Karavirta et al. [10] suggest both limiting the number of attempts and randomizing the inputs for the problem after each run.

8 CONCLUSION

Homegrown and third-party solutions for auto assessment abound, but we present in this work a framework that is, by design, open-source and extensible in hopes that we, as a community, can reinvent fewer wheels. While we ourselves use the framework to test code written in C and Python in CS50 at Harvard, the framework itself is language-agnostic, and extensions for additional languages are welcome. While `check50`'s deployment in CS50 seemed straightforward initially, it has not been without side effects, nearly all of them behavioral but manageable pedagogically. There is surely a balance to be struck, then, between such any tool's upsides and downsides, and with each passing term do we refine our own usage. And with the framework's cloud-based back end now accumulating so many submissions, we anticipate that we can ultimately provide automated feedback not only on correctness but also design, the latter informed by TFs' own qualitative assessment of past students' submissions that resemble those present.

ACKNOWLEDGEMENTS

Our thanks to Amazon Web Services and Google for their support of this work.

APPENDIX

`check50` is available at https://github.com/cs50/check50 as open-source software.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.
[2] CodeLab by Turing's Craft. 2020. https://www.turingscraft.com/
[3] Julio C Caiza and Jose M Del Alamo. 2013. Programming assignments automatic grading: review of tools and implementations. In *7th International Technology, Education and Development Conference (INTED2013)*. 5691.
[4] Kyle Dewey, Phillip Conrad, Michelle Craig, and Elena Morozova. 2017. Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE âĂŹ17)*. Association for Computing

Machinery, New York, NY, USA, 317âĂŞ322. https://doi.org/10.1145/3059009.3059051

[5] Stephen H. Edwards. 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE âĂŹ04)*. Association for Computing Machinery, New York, NY, USA, 26âĂŞ30. https://doi.org/10.1145/971300.971312

[6] Gradescope. 2020. https://www.gradescope.com/

[7] Jessica B Hamrick. 2016. Creating and grading IPython/Jupyter notebook assignments with NbGrader. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 242–242.

[8] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*. 86–93.

[9] Tiobe Index. 2018. https://www.tiobe.com/tiobe-index/.

[10] Ville Karavirta, Ari Korhonen, and Lauri Malmi. 2006. On the use of resubmissions in automatic assessment systems. *Computer science education* 16, 3 (2006), 229–240.

[11] OK. 2020. https://okpy.org/

[12] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. *CoRR* abs/1505.05969 (2015). arXiv:1505.05969 http://arxiv.org/abs/1505.05969

[13] Autolab Project. 2020. http://www.autolabproject.com/.

[14] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. 2001. Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*. 133–136.

[15] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2012. Automated Feedback Generation for Introductory Programming Assignments. arXiv:cs.PL/1204.1751

[16] Vocareum. 2020. https://help.vocareum.com/article/16-grading-on-vocareum