check50 Documentation

CS50

Aug 20, 2019

Contents:

CHAPTER 1

Running check50

1.1 Slug

check50 requires one positional argument, a so called slug. Something like this:

```
check50 cs50/problems/2018/x/caesar
```

Why? Well, anyone can write checks for check50 without needing to ask for permission. To achieve this goal check50, the tool, is decoupled from any of its checks. The checks themselves are hosted anywhere on a popular hosting platform for code, GitHub. All you need to do to add your own checks is to put them on anywhere on GitHub. With this flexibility comes a price, as check50 does not know where your checks live on GitHub. In order to uniquely identify a collection of checks on GitHub check50 needs the following:

- org
- repository
- branch
- path to problem

These four pieces of information separated by a / is what check50 calls a slug, a string that uniquely identifies a set of checks. For instance the slug `cs50/problems/2018/x/caesar` uniquely identifies the org `cs50`, the repository `problems`, the branch `2018/x` and the path `caesar`.

1.2 Operation modes

Check50 can run in four mutually exclusive modes of operation.

1.2.1 local

By default check50 runs locally. That means the checks run locally on the machine you run check50 on. The checks however are fetched remotely from GitHub.

1.2.2 offline

Running with `--offline` runs the checks locally and has check50 look for checks locally. check50 will not try to fetch checks remotely in this mode.

1.2.3 online

Running with `--online` runs the checks remotely and then waits for the results to come back.

1.2.4 dev

The `--dev` flag signals check50 to run in developer mode. This implies `--offline`. More on this mode in *Writing check50 checks*.

1.3 Additional output

By default check50 will try to keep its output concise in its `ansi` output mode. For each check you will see at most its description and rationale/help on why the check failed. check50 will by default hide tracebacks and log output. You can show both by running check50 with the `--verbose` flag, or just the log with the `--log` flag.

1.3.1 verbose

Running with `--verbose` lets check50 output both the log and any tracebacks in the `ansi` output mode.

1.3.2 log

Running check50 with `--log` will have check50 print out its logs.

1.4 Targeting checks

Check50 lets you target specific checks by name with the `--target` flags. This will have check50 run just these checks and their dependencies.

1.4.1 target

With `--target` you can target checks from a larger body of checks by name. check50 will only run and show these checks and their dependencies.

1.5 Output modes

check50 supports three output modes: `ansi`, `html` and `json`. In short, the `ansi` output mode is text-based output meant to be displayed in a terminal. `html` is an extension of `ansi` showing the same results but in a webpage. This allows for visual comparisons and more information to be displayed in general. Finally, the `json` output mode provides a machine readable form of output, that can for instance be used for automatic grading.

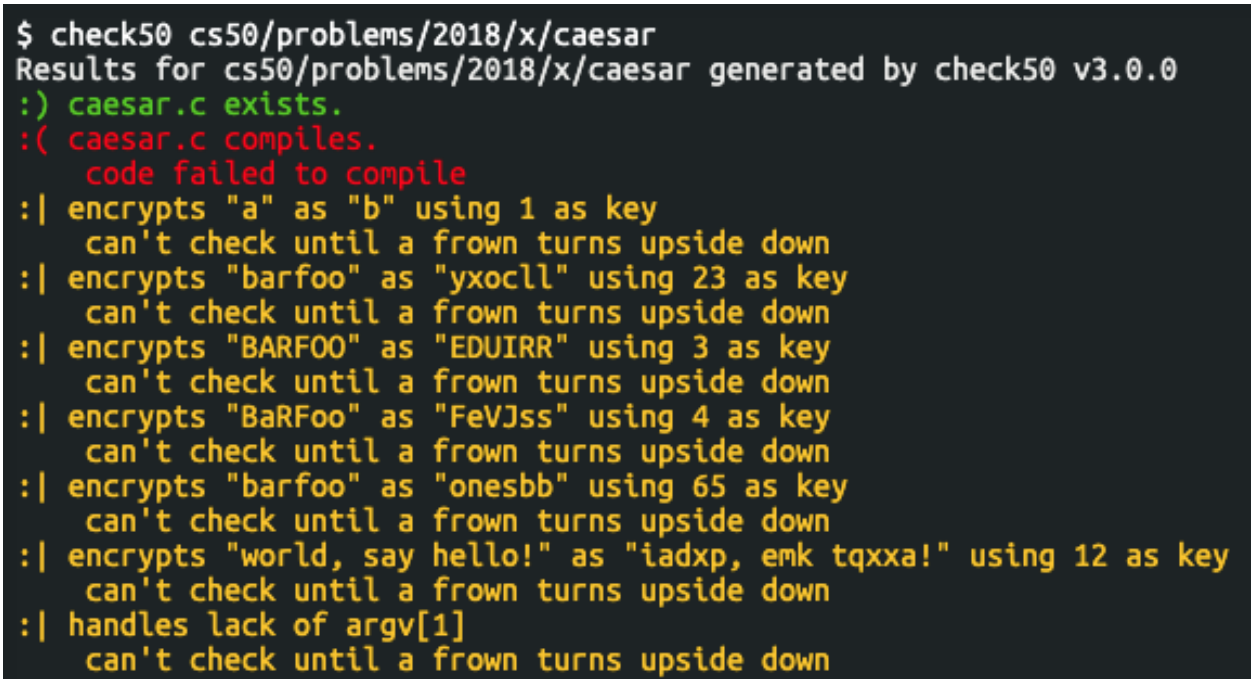
The output modes can be mixed and matched through the `--output` or `-o` flag.

```
check50 <slug> -o ansi
check50 <slug> -o ansi html # equivalent to check50 <slug>
check50 <slug> -o json
```

By default check50 shows both `ansi` and `html` output.

1.5.1 ansi

The `ansi` output mode will have check50 print the results from the checks to stdout. This output mode keeps students in the terminal, the environment in which they are running check50 in the first place. Limited by its nature, check50's `ansi` output mode is not suited for large pieces of text or visual comparisons. The output format is sufficient for showing which checks passed and failed, and offering short text based help or explanation on those checks.



1.5.2 html

In addition to `ansi`, check50 comes with a `html` output mode. This output mode allows check50 to show results side by side and to display more verbose information like the log by default. check50 creates a local self contained static html file in `/tmp` and will output the path to file in stdout.

check50

cs50/problems/2018/x/caesar

:) caesar.c exists.

Log
checking that caesar.c exists...

:(caesar.c compiles.

code failed to compile
Log
running clang caesar.c -o caesar -std=c11 -ggdb -lm -lcs50...
caesar.c:24:5: warning: implicit declaration of function 'f' is invalid in C99
[-Wimplicit-function-declaration]
f (argc != 2)
^
caesar.c:24:18: error: expected ';' after expression
f (argc != 2)
^
;
1 warning and 1 error generated.

:| encrypts "a" as "b" using 1 as key

can't check until a frown turns upside down

:| encrypts "barfoo" as "yxocll" using 23 as key

can't check until a frown turns upside down

:| encrypts "BARFOO" as "EDUIRR" using 3 as key

can't check until a frown turns upside down

:| encrypts "BaRFoo" as "FeVJss" using 4 as key

can't check until a frown turns upside down

:| encrypts "barfoo" as "onesbb" using 65 as key

can't check until a frown turns upside down

:| encrypts "world, say hello!" as "iadxp, emk tqxxa!" using 12 as key

can't check until a frown turns upside down

:| handles lack of argv[1]

can't check until a frown turns upside down

1.5.3 json

check50 can provide machine readable output in the form of `json`. By default this output mode will print to stdout, but like any other form of output check50 can write to a file with the `--output-file` command line option. For a

complete overview of the `json` output please refer to the *JSON specification*.


```
{
    "slug": "cs50/problems/2018/x/caesar",
    "results": [
        {
            "name": "exists",
            "description": "caesar.c exists.",
            "passed": true,
            "log": [
                "checking that caesar.c exists..."
            ],
            "cause": null,
            "data": {},
            "dependency": null
        },
        {
            "name": "compiles",
            "description": "caesar.c compiles.",
            "passed": false,
            "log": [
                "running clang caesar.c -o caesar -std=c11 -ggdb -lm -lcs50...",
                "caesar.c:24:5: warning: implicit declaration of function 'f' is␣
↪invalid in C99",
                "      [-Wimplicit-function-declaration]",
                "    f (argc != 2)",
                "    ^",
                "caesar.c:24:18: error: expected ';' after expression",
                "    f (argc != 2)",
                "                 ^",
                "                 ;",
                "1 warning and 1 error generated."
            ],
            "cause": {
                "rationale": "code failed to compile",
                "help": null
            },
            "data": {},
            "dependency": "exists"
        },
        {
            "name": "encrypts_a_as_b",
            "description": "encrypts \"a\" as \"b\" using 1 as key",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
            },
            "data": {},
            "dependency": "compiles"
        },
        {
            "name": "encrypts_barfoo_as_yxocll",
            "description": "encrypts \"barfoo\" as \"yxocll\" using 23 as key",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
```


(continues on next page)

(continued from previous page)


```
            },
            "data": {},
            "dependency": "compiles"
        },
        {
            "name": "encrypts_BARFOO_as_EDUIRR",
            "description": "encrypts \"BARFOO\" as \"EDUIRR\" using 3 as key",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
            },
            "data": {},
            "dependency": "compiles"
        },
        {
            "name": "encrypts_BaRFoo_FeVJss",
            "description": "encrypts \"BaRFoo\" as \"FeVJss\" using 4 as key",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
            },
            "data": {},
            "dependency": "compiles"
        },
        {
            "name": "encrypts_barfoo_as_onesbb",
            "description": "encrypts \"barfoo\" as \"onesbb\" using 65 as key",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
            },
            "data": {},
            "dependency": "compiles"
        },
        {
            "name": "checks_for_handling_non_alpha",
            "description": "encrypts \"world, say hello!\" as \"iadxp, emk tqxxa!\"␣
↪using 12 as key",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
            },
            "data": {},
            "dependency": "compiles"
        },
        {
            "name": "handles_no_argv",
            "description": "handles lack of argv[1]",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
            },
            "data": {},
```


(continues on next page)

(continued from previous page)

```
            "data": {},
            "dependency": "compiles"
        }
    ],
    "version": "3.0.0"
}
```

CHAPTER 2

API docs

CHAPTER 3

JSON specification

Check50 can create a machine readable output in the form of *json*. For instance, running check50 with *-o json* on a non compiling implementation of one of our problems called caesar:

```
check50 cs50/problems/2018/x/caesar -o json
```

Produces the following:

```
{
    "slug": "cs50/problems/2018/x/caesar",
    "results": [
        {
            "name": "exists",
            "description": "caesar.c exists.",
            "passed": true,
            "log": [
                "checking that caesar.c exists..."
            ],
            "cause": null,
            "data": {},
            "dependency": null
        },
        {
            "name": "compiles",
            "description": "caesar.c compiles.",
            "passed": false,
            "log": [
                "running clang caesar.c -o caesar -std=c11 -ggdb -lm -lcs50...",
                "caesar.c:24:5: warning: implicit declaration of function 'f' is␣
↪invalid in C99",
                "      [-Wimplicit-function-declaration]",
                "    f (argc != 2)",
                "    ^",
                "caesar.c:24:18: error: expected ';' after expression",
                "    f (argc != 2)",
```

(continues on next page)

(continued from previous page)


```
                "                   ^",
                "                   ;",
                "1 warning and 1 error generated."
            ],
            "cause": {
                "rationale": "code failed to compile",
                "help": null
            },
            "data": {},
            "dependency": "exists"
        },
        {
            "name": "encrypts_a_as_b",
            "description": "encrypts \"a\" as \"b\" using 1 as key",
            "passed": null,
            "log": [],
            "cause": {
                "rationale": "can't check until a frown turns upside down"
            },
            "data": {},
            "dependency": "compiles"
        },
    ],
    "version": "3.0.0"
}
```


3.1 Top level

Assuming *check50* is able to run successfully, you will find three keys at the top level of the json output: *slug*, *results* and *version*.

- **slug** (*string*) is the slug with which check50 was run, *cs50/problems/2018/x/caesar* in the above example.
- **results** (*[object]*) is a list containing the results of each run check. More on this key below.
- **version** (*string*) is the version of check50 used to run the checks.

If check50 encounters an error while running, e.g. due to an invalid slug, the *results* key will be replaced by an *error* key containing information about the error encountered.

3.2 results

If the results key exists (that is, check50 was able to run the checks successfully), it will contain a list of objects each corresponding to a check. The order of these objects corresponds to the order the checks appear in the file in which they were written. Each object will contain the following fields:

- **name** (*string*) is the unique name of the check (the literal name of the Python function specifying the check).
- **description** (*string*) is a description of the check.
- **passed** (*bool*, nullable) is *true* if the check passed, *false* if the check failed, or *null* if the check was skipped (either because the check's dependency did not pass or because the check threw some unexpected error).
- **log** (*[string]*) contains the log accrewed during the execution of the check. Each element of the list is a line from the log.

- **cause** (*object*, nullable) contains the reason that a check did not pass. If *passed* is *true*, *cause* will be *null* and *cause* will never be *null* if *passed* is not *true*. More detail about keys that may appear within *cause* below.
- **data** (*object*) contains arbitrary data communicated by the check via the *check50.data* API call. Checks could use this to add additional information such as memory usage to the results, but check50 itself does not add anything to *data* by default.
- **dependency** (*string*, nullable) is the name of the check upon which this check depends, or *null* if the check has no dependency.

3.2.1 cause

The cause key is *null* if the check passed and non-null. This key is by design an open-ended object. Everything in the *.payload* attribute of a *check50.Failure* will be put in the *cause* key. Through this mechanism you can communicate any information you want from a failing check to the results. Depending on what occurred, check50 adds the following keys to *cause*:

- **rationale** (*string*) is a stduent-facing explanation of why the check did not pass (e.g. the student's program did not output what was expected).
- **help** (*string*) is an additional help message that may appear alongside the rationale giving additional context.
- **expected** (*string*) and **actual** (*string*) are keys that always appear in a pair. In case you are expecting X as output, but Y was found instead, you will find these keys containing X and Y in the *cause* field. These appear when a check raises a *check50.Mismatch* exception.
- **error** (*object*) appears in *cause* when an unexpected error occurred during a check. It will contain the keys *type*, *value*, *traceback* and *data* with the same properties as in the top-level *error* key described below.

3.3 error

If check50 encounters an unexpected error, the *error* key will replace the *results* key in the JSON output. It will contain the following keys:

- **type** (*string*) contains the type name of the thrown exception.
- **value** (*string*) contains the result of converting the exception to a string.
- **traceback** (*[string]*) contains the stack trace of the thrown exception.
- **data** (*object*) contains any additional data the exception may carry in its *payload* attribute.

CHAPTER 4

Writing check50 checks

check50 checks live in a git repo on Github. check50 finds the git repo based on the slug that is passed to check50. For instance, consider the following execution of check50:

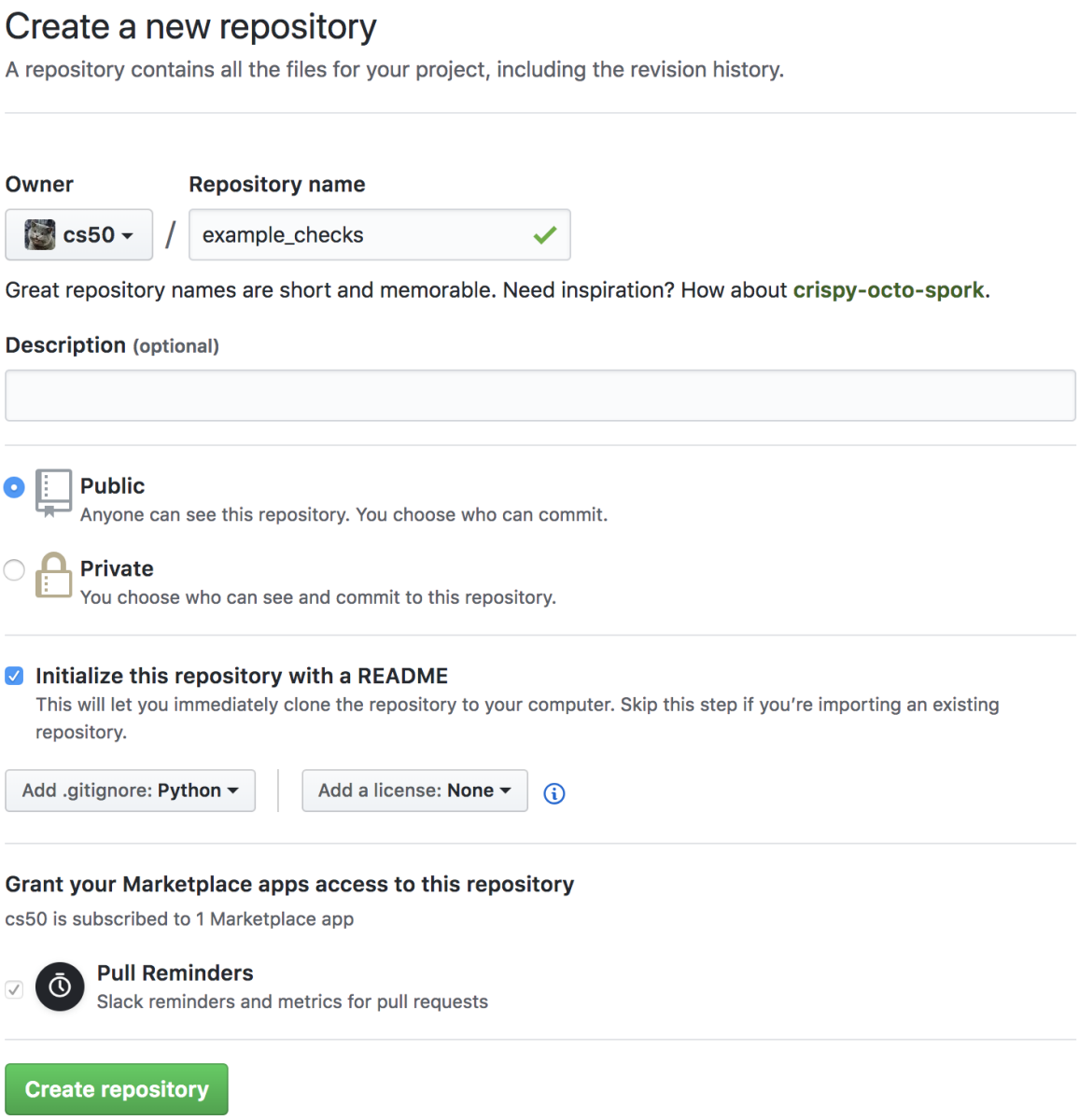
```
check50 cs50/problems/2018/x/hello
```

check50 will look for an owner called *cs50*, a repo called *problems*, a branch called *2018* or *2018/x* and a problem called *x/hello* or *hello*. The slug is thus parsed like so:

```
check50 <owner>/<repo>/<branch>/<problem>
```

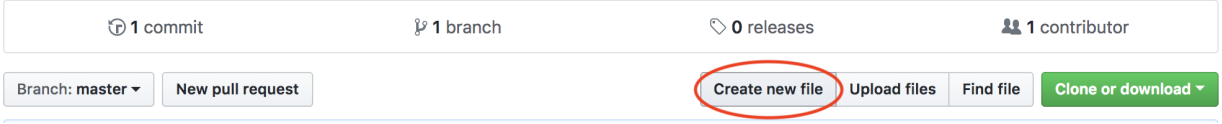
4.1 Creating a git repo

To get you started, the first thing you need to do is . Once you have done so, or if you already have an account with Github, . Make sure to think of a good name for your repo, as this is what students will be typing. Also make sure your repo is set to public, it is initialised with a *README*, and finally add a Python *.gitignore*. Ultimately you should have something looking like this:

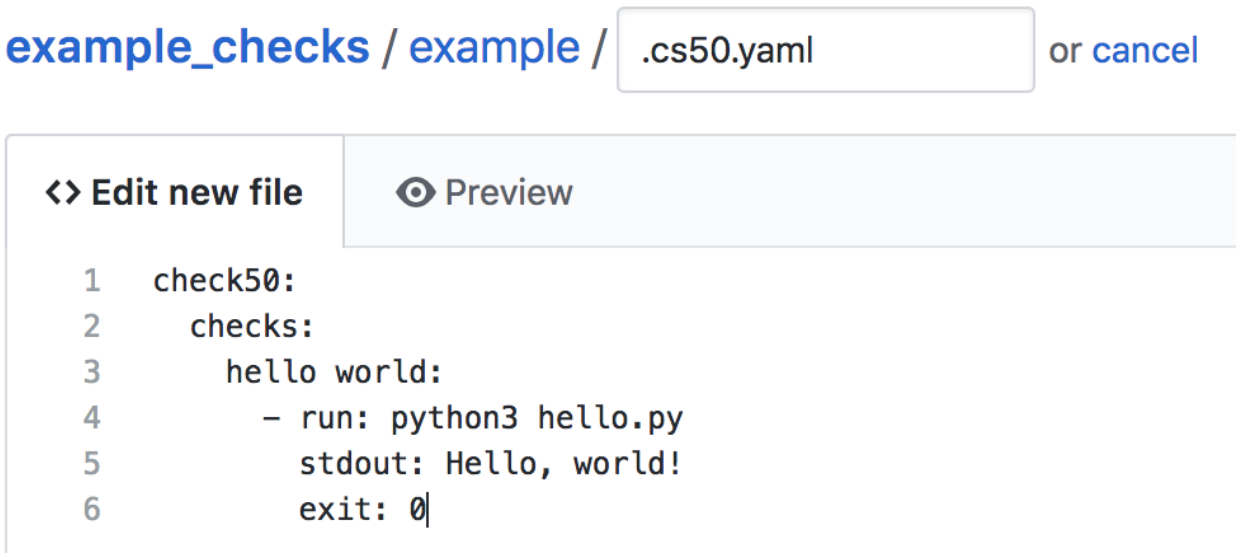


4.2 Creating a check and running it

Your new repo should live at `https://github.com/<user>/<repo>`, that is `https://github.com/cs50/example_checks` in our example. Once you have created your new repo, create a new file by clicking the *Create new file* button:



Then continue by creating the following *.cs50.yaml* file. All indentation is done by 2 spaces, as per YAML syntax.

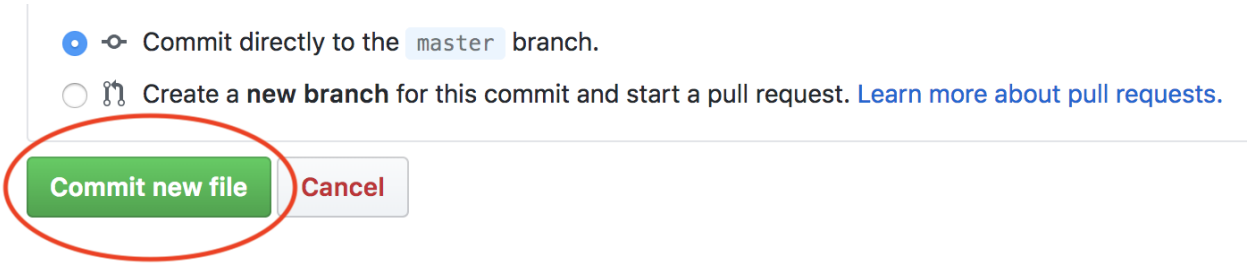


Or in text, if you want to quickly copy-paste:

Listing 1: **.cs50.yaml**

```yaml
check50:
  checks:
    hello world:
      - run: python3 hello.py
        stdout: Hello, world!
        exit: 0
```

Note that you should create a directory like in the example above by typing: *example/.cs50.yaml*. Once you have populated the file with the code above. Scroll down the page and hit the commit button:



That's it! You know have a repo that check50 can use to check whether a python file called *hello.py* prints `Hello, world!` and exits with a `0` as exit code. To try it, simply execute:

```
check50 <owner>/<repo>/master/example --local
```

Where you substitute *<owner>* for your own username, *<repo>* for the repo you've just created. Given that a file called *hello.py* is in your current working directory, and it actually prints `Hello, world!` when run, you should now see the following:

```
:) hello world
```

4.3 Simple YAML checks

To get you started, and to cover the basics of input/output checking, check50 lets you write simple checks in YAML syntax. Under the hood, check50 compiles these YAML checks to Python checks that check50 then runs.

YAML checks in check50 all live in *.cs50.yaml* and start with a top-level key called `check50`, that specifies a `checks`. The `checks` record contains all checks, where the name of the check is the name of the YAML record. Like so:

Listing 2: **.cs50.yaml**

```
check50:
  checks:
    hello world: # define a check named hello world
      # check code
    foo: # define a check named foo
      # check code
    bar: # define a check named bar
      # check code
```

This code snippet defines three checks, named `hello world`, `foo` and `bar` respectively. These checks should contain a list of `run` records, that can each contain a combination of `stdin`, `stdout` and `exit`. See below:

Listing 3: **.cs50.yaml**

```
check50:
  checks:
    hello world:
      - run: python3 hello.py # run python3 hello.py
        stdout: Hello, world! # expect Hello, world! in stdout
        exit: 0 # expect program to exit with exitcode 0
    foo:
      - run: python3 foo.py # run python3 foo.py
        stdin: baz # insert baz into stdin
        stdout: baz # expect baz in stdout
        exit: 0 # expect program to exit with exitcode 0
    bar:
      - run: python3 bar.py # run python3 bar.py
        stdin: baz # insert baz into stdin
        stdout: bar baz # expect bar baz in stdout
      - run: python3 bar.py # run python3 bar.py
        stdin:
          - baz # insert baz into stdin
          - qux # insert qux into stdin
        stdout:
          - bar baz # first expect bar baz in stdout
          - bar qux # then expect bar qux in stdout
```

The code snippet above again defines three checks: *hello world*, *foo* and *bar*.

The `hello world` check runs `python3 hello.py` in the terminal, expects `Hello, world!` to be outputted in stdout, and then expects the program to exit with exitcode `0`.

The `foo` check runs `python3 foo.py` in the terminal, inserts `baz` into stdin, expects `baz` to be outputted in stdout, and finally expects the program to exit with exitcode `0`.

The `bar` check runs two commands in order in the terminal. First `python3 bar.py` gets run, `baz` gets put in stdin and `bar baz` is expected in stdout. There is no mention of `exit` here, so the exitcode is not checked. Secondly, `python3 bar.py` gets run, `baz` and `qux` get put into stdin, and first `bar baz` is expected in stdout, then `bar qux`.

We encourage you to play around with the example above by copying its code to your checks git repo. Then try to write a *bar.py* and *foo.py* that make you pass these tests.

In case you want to check for multiline input, you can make use of YAML's `|` operator like so:

Listing 4: **.cs50.yaml**

```
check50:
  checks:
    multiline hello world:
      - run: python3 multi_hello.py
        stdout: | # expect Hello\nWorld!\n in stdout
          Hello
          World!
        exit: 0
```

4.4 Developing locally

To write checks on your own machine, rather than on the Github webpage, you can clone the repo via:

```
git clone https://github.com/<owner>/<repo>
```

Where `<owner>` is your Github username, and `<repo>` is the name of your checks repository. Head on over to the new directory that git just created, and open up *.cs50.yaml* with your favorite editor.

To run the cloned checks locally, check50 comes with a `--dev` mode. That will let you target a local checks repo, rather than a github repo. So if your checks live in `/Users/cs50/Documents/example_checks`, you would execute check50 like so:

```
check50 --dev /Users/cs50/Documents/example_checks/example
```

This runs the *example* check from `/Users/cs50/Documents/example_checks`. You can also specify a relative path, so if your current working directory is `/Users/cs50/Documents/solutions`, you can execute check50 like so:

```
check50 --dev ../example_checks/example
```

Now you're all set to develop new checks locally. Just remember to `git add`, `git commit` and `git push` when you're done writing checks. Quick refresher:

```
git add .cs50.yaml
git commit -m "wrote some awesome new checks!"
git push
```

4.5 Getting started with Python checks

If you need a little more than strict input / output testing, check50 lets you write checks in Python. A good starting point is the result of the compilation of the YAML checks. To get these, please make sure you have cloned the repo (via `git clone`), and thus have the checks locally. First we need to run the .YAML checks once, so that check50 compiles the checks to Python. To do this execute:

```
check50 --dev <checks_dir>/<check>
```

Where `<checks_dir>` is the local git repo of your checks, and `<check>` is the directory in which `.cs50.yaml` lives. Alternatively you could navigate to this directory and simply call:

```
check50 --dev .
```

As a result you should now find a file called `__init__.py` in the check directory. This is the result of check50's compilation from YAML to Python. For instance, if your `.cs50.yaml` contains the following:

Listing 5: **.cs50.yaml**

```
check50:
  checks:
    hello world:
      - run: python3 hello.py
        stdout: Hello, world!
        exit: 0
```

You should now find the following `__init__.py`:

Listing 6: **__init__.py**

```
import check50

@check50.check()
def hello_world():
    """hello world"""
    check50.run("python3 hello.py").stdout("Hello, world!", regex=False).exit(0)
```

check50 will by default ignore and overwrite what is in `__init__.py` for as long as there are checks in `.cs50.yaml`. To change this you have to edit `.cs50.yaml` to:

Listing 7: **.cs50.yaml**

```
check50: true
```

If the `checks` key is not specified (as is the case above), `check50` will look for Python checks written in a file called `__init__.py`. If you would like to write the Python checks in a file called `foo.py` instead, you could specify it like so:

Listing 8: **.cs50.yaml**

```
check50:
    checks: foo.py
```

To test whether everything is still in working order, run check50 again with:

```
check50 --dev <checks_dir>/<check>
```

You should see the same results as the YAML checks gave you. Now that there are no YAML checks in `.cs50.yaml` and check50 knows where to look for Python checks, you can start writing Python checks. You can find documentation in *API docs*, and examples of Python checks below.

4.6 Python check specification

A Python check is made up as follows:

Listing 9: **__init__.py**

```
1  import check50 # import the check50 module
2
3  @check50.check() # tag the function below as check50 check
4  def exists(): # the name of the check
5      """description""" # this is what you will see when running check50
6      check50.exists("hello.py") # the actual check
7
8  @check50.check(exists) # only run this check if the exists check has passed
9  def prints_hello():
10     """prints "hello, world\\n" """
11     check50.run("python3 hello.py").stdout("[Hh]ello, world!?\n", regex=True).exit(0)
```

check50 uses its check decorator to tag functions as checks. You can pass another check as argument to specify a dependency. Docstrings are used as check descriptions, this is what will ultimately be shown when running check50. The checks themselves are just Python code. check50 comes with a simple API to run programs, send input to stdin, and check or retrieve output from stdout. A check fails if a `check50.Failure` exception or an exception inheriting from `check50.Failure` like `check50.Mismatch` is thrown. This allows you to write your own custom check code like so:

Listing 10: **__init__.py**

```
1  import check50
2
3  @check50.check()
4  def prints_hello():
5      """prints "hello, world\\n" """
6      from re import match
7
8      expected = "[Hh]ello, world!?\n"
9      actual = check50.run("python3 hello.py").stdout()
10     if not match(expected, actual):
11         help = None
12         if match(expected[:-1], actual):
13             help = r"did you forget a newline ('\n') at the end of your printf string?
   ↪"
14         raise check50.Mismatch("hello, world\n", actual, help=help)
```

The above check breaks out of check50's API by calling `stdout()` on line 9 with no args, effectively retrieving all output from stdout in a string. Then there is some plain Python code, matching the output through Python's builtin regex module `re` against a regular expression with the expected outcome. If it doesn't match, a help message is provided only if there is a newline missing at the end. This help message is provided through an optional argument `help` passed to check50's `Mismatch` exception.

You can share state between checks if you make them dependent on each other. By default file state is shared, allowing you to for instance test compilation in one check, and then depend on the result of the compilation in dependent checks.

Listing 11: __init__.py

```
import check50
import check50.c

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

@check50.check(compiles)
def prints_hello():
    """prints "hello, world\\n" """
    check50.run("./hello").stdout("[Hh]ello, world!?\n", regex=True).exit(0)
```

You can also share Python state between checks by returning what you want to share from a check. It's dependent can accept this by accepting an additional argument.

Listing 12: __init__.py

```
import check50

@check50.check()
def foo():
    return 1

@check50.check(foo)
def bar(state)
    print(state) # prints 1
```

4.7 Python check examples

Below you will find examples of Python checks. Don't forget to for more examples. You can try them yourself by copying them to `__init__.py` and running:

```
check50 --dev <checks_dir>/<check>
```

Check whether a file exists:

Listing 13: __init__.py

```
import check50

@check50.check()
def exists():
    """hello.py exists"""
    check50.exists("hello.py")
```

Check stdout for an exact string:

Listing 14: __init__.py

```
@check50.check(exists)
def prints_hello_world():
```

(continues on next page)

(continued from previous page)

```
    """prints Hello, world!"""
    check50.run("python3 hello.py").stdout("Hello, world!", regex=False).exit(0)
```

Check stdout for a rough match:

Listing 15: __init__.py

```
@check50.check(exists)
def prints_hello():
    """prints "hello, world\\n" """
    # regex=True by default :)
    check50.run("python3 hello.py").stdout("[Hh]ello, world!?\n").exit(0)
```

Put something in stdin, expect it in stdout:

Listing 16: __init__.py

```
import check50

@check50.check()
def id():
    """id.py prints what you give it"""
    check50.run("python3 hello.py").stdin("foo").stdout("foo").stdin("bar").stdout(
↪"bar")
```

Be helpful, check for common mistakes:

Listing 17: __init__.py

```
import check50
import re

def coins(num):
    # regex that matches `num` not surrounded by any other numbers
    # (so coins(2) won't match e.g. 123)
    return fr"(?<!\d){num}(?!\d)"

@check50.check()
def test420():
    """input of 4.2 yields output of 18"""
    expected = "18\n"
    actual = check50.run("python3 cash.py").stdin("4.2").stdout()
    if not re.search(coins(18), actual):
        help = None
        if re.search(coins(22), actual):
            help = "did you forget to round your input to the nearest cent?"
        raise check50.Mismatch(expected, actual, help=help)
```

Create your own assertions:

Listing 18: __init__.py

```
import check50

@check50.check()
def atleast_one_match()
```

(continues on next page)

(continued from previous page)

```
    """matches either foo, bar or baz"""
    output = check50.run("python3 qux.py").stdout()
    if not any(answer in output for answer in ["foo", "bar", "baz"]):
        raise check50.Failure("no match found")
```

4.8 Configuring check50

Check50, and other CS50 tools like submit50 and lab50, use a special configuration file called `.cs50.yaml`. Here is how you can configure check50 via `.cs50.yaml`.

4.8.1 checks:

`checks:` takes a filename specifying a file containing check50 Python checks, or a record of check50 YAML checks. If not specified, it will default to `__init__.py`.

Listing 19: **.cs50.yaml**

```
check50:
  checks: checks.py
```

Only specifies that this is a valid slug for check50. This configuration will allow you to run `check50 <slug>`, by default `check50` will look for an `__init__.py` containing Python checks.

Listing 20: **.cs50.yaml**

```
check50:
  checks: "my_filename.py"
```

Specifies that this is a valid slug for check50, and has check50 look for `my_filename.py` instead of `__init__.py`.

Listing 21: **.cs50.yaml**

```
check50:
  checks:
    hello world:
      - run: python3 hello.py
        stdout: Hello, world!
        exit: 0
```

Specifies that this is a valid slug for check50, and has check50 compile and run the YAML check. For more on YAML checks in check50 see :ref:`check_writer`.

4.8.2 files:

`files:` takes a list of files/patterns. Every item in the list must be tagged by either `!include`, `!exclude` or `!require`. All files matching a pattern tagged with `!include` are included and likewise for `!exclude`. `!require` is similar to `!include`, however it does not accept patterns, only filenames, and will cause `check50` to display an error if that file is missing. The list that is given to `files:` is processed top to bottom. Later items in `files:` win out over earlier items.

The patterns that `!include` and `!exclude` accept are globbed, any matching files are added. check50 introduces one exception for convenience, similarly to how git treats .gitignore: If and only if a pattern does not contain a `/`, and starts with a `*`, it is considered recursive in such a way that `*.o` will exclude all files in any directory ending with `.o`. This special casing is just for convenience. Alternatively you could write `**/*.o` that is functionally identical to `*.o`, or write `./*.o` if you only want to exclude files ending with `.o` from the top-level directory.

Listing 22: **.cs50.yaml**

```
check50:
  files:
    - !exclude "*.pyc"
```

Excludes all files ending with `.pyc`.

Listing 23: **.cs50.yaml**

```
check50:
  files:
    - !exclude "*"
    - !include "*.py"
```

Exclude all files, but include all files ending with `.py`. Note that order is important here, if you would inverse the two lines it would read: include all files ending with `.py`, exclude everything. Effectively excluding everything!

Listing 24: **.cs50.yaml**

```
check50:
  files:
    - !exclude "*"
    - !include "source/"
```

Exclude all files, but include all files in the source directory.

Listing 25: **.cs50.yaml**

```
check50:
  files:
    - !exclude "build/"
    - !exclude "docs/"
```

Include everything, but exclude everything in the build and docs directories.

Listing 26: **.cs50.yaml**

```
check50:
  files:
    - !exclude "*"
    - !include "source/"
    - !exclude "*.pyc"
```

Exclude everything, include everything from the source directory, but exclude all files ending with `.pyc`.

Listing 27: **.cs50.yaml**

```
check50:
  files:
    - !exclude "source/**/*.pyc"
```

Include everything, but any files ending on `.pyc` within the source directory. The `**` here pattern matches any directory.

Listing 28: **.cs50.yaml**

```
check50:
  files:
    - !require "foo.py"
    - !require "bar.c"
```

Require that both foo.py and bar.c are present and include them.

Listing 29: **.cs50.yaml**

```
check50:
  files:
    - !exclude "*"
    - !include "*.py"
    - !require "foo.py"
    - !require "bar.c"
```

Exclude everything, include all files ending with `.py` and require (and include) both foo.py and bar.c. It is generally recommended to place any `!require``d files at the end of the ``files:`, this ensures they are always included.

4.8.3 dependencies:

`dependencies:` is a list of `pip` installable dependencies that check50 will install.

Listing 30: **.cs50.yaml**

```
check50:
  dependencies:
    - pyyaml
    - flask
```

Has check50 install both `pyyaml` and `flask` via `pip`.

Listing 31: **.cs50.yaml**

```
check50:
  dependencies:
    - git+https://github.com/cs50/submit50#egg=submit50
```

Has check50 `pip install` submit50 from GitHub, especially useful for projects that are not hosted on PyPi. See https://pip.pypa.io/en/stable/reference/pip_install/#vcs-support for more info on installing from a VCS.

4.9 Internationalizing checks

TODO

CHAPTER 5

Writing check50 extensions

Core to check50's design is extensibility. Not only in checks, but also in the tool itself. We designed `check50.c`, `check50.py` and `check50.flask` to be extensions of check50 that ship with check50. By design these three modules are all standalone and only depend on the core of check50, and no other part of check50 depends on them. In other words, you can remove or add any of these modules and check50 would still function as expected.

We ship check50 with three extensions because these extensions are core to the material cs50 teaches. But different courses have different needs, and we realize we cannot predict and cater for every usecase. This is why check50 comes with the ability to *pip install* other Python packages. You can configure this via the `dependencies` key in `.cs50.yaml`. Through this mechanism you can write your own extension and then have check50 install it for you whenever check50 runs. Host your extension anywhere *pip* can install from, for instance GitHub or PyPi. And all you have to do then is to fill in the `dependencies` key of `.cs50.yaml` with the location of your extension. check50 will make sure your extension is always there when the checks are run.

5.1 check50.internal

In addition to all the functionality check50 exposes, we expose an extra API for extensions in `check50.internal`. You can find the documentation in *API docs*.

5.2 Example: a JavaScript extension

Out of the box check50 does not ship with any JavaScript specific functionality. You can use check50's generic API and run a `.js` file through an interpreter such as `node`: `check50.run('node <student_file.js>')`. But we realize most JavaScript classes are not about writing command-line scripts, and we do need a way to call functions. This is why we wrote a small javascript extension for check50 dubbed check50_js at https://github.com/cs50/check50/tree/sample-extension.

5.2.1 check50_js

The challenge in writing this extension is that check50 itself is written in Python, so we need an interface between the two languages. This could be as simple as an intermediate JavaScript script that runs the students function and then outputs the results to stdout for check50 to read. But this approach does create indirection and creates quite some clutter in the checks codebase. Luckily, in case of Python and JavaScript (and PHP and Perl) we have access to a Python package called `python-bond`. This package lets us "bond" two languages together, and lets you evaluate code in another language's interpreter. Effectively creating a channel through which you can evaluate code and call functions from the other language. This is what we ended up doing for our JavaScript extension.

You can find example checks using check50_js and their solutions at:

- **hello.js**: checks solution
- **line.js**: checks solution
- **addition.js**: checks solution

To try any of these examples for yourself, simply run:

- **hello.js**:

```
wget https://raw.githubusercontent.com/cs50/check50/examples/solutions/hello_js/hello.
↪js
check50 cs50/check50/examples/js/hello
```

- **line.js**:

```
wget https://raw.githubusercontent.com/cs50/check50/examples/solutions/line_js/line.js
check50 cs50/check50/examples/js/line
```

- **addition.js**:

```
wget https://raw.githubusercontent.com/cs50/check50/examples/solutions/addition_js/
↪addition.js
check50 cs50/check50/examples/js/addition
```

check50 is a tool for checking student code. As a student you can use check50 to check your CS50 problem sets or any other Problem sets for which check50 checks exist. check50 allows teachers to automatically grade code on correctness and to provide automatic feedback while students are coding.

CHAPTER 6

Installation

First make sure you have Python 3.6 or higher installed. You can download Python .

check50 has a dependency on git, please make sure to if git is not already installed.

To install check50 under Linux / OS X:

```
pip install check50
```

Under Windows, please . Then install check50 within the subsystem.

CHAPTER 7

Usage

To use check50 to check a problem, execute check50 like so:

```
check50 <owner>/<repo>/<branch>/<check>
```

For instance, if you want to check you call:

```
check50 cs50/problems/2018/x/caesar
```

You can choose to run checks locally by passing the `--local` flag like so:

```
check50 --local <owner>/<repo>/<branch>/<check>
```

For an overview of all flags run:

```
check50 --help
```

CHAPTER 8

Design

- **Write checks for code in code** check50 uses pure Python for checks and exposes a small Python api for common functionality.
- **Extensibility in checks** Anyone can add checks to check50 without asking for permission. In fact, here is a tutorial to get you started: *Writing check50 checks*
- **Extensibility in the tool itself** We cannot predict everything you need, nor can we cater for every use-case out of the box. This is why check50 provides you with a mechanism for adding your own code to the tool, once more without asking for permission. This lets you support different programming languages and add new functionality. Jump to *Writing check50 extensions* to learn more.
- **PaaS** check50 can run online. This guarantees a consistent environment and lets you check code for correctness without introducing your own hardware.

CHAPTER 9

Checks

In check50 the checks are decoupled from the tool. You can find CS50's set of checks for CS50 problem sets at . If you would like to develop your own set of checks such that you can use check50 in your own course jump to *Writing check50 checks*.

Under the hood, checks are naked Python functions decorated with the `@check50.check` decorator. check50 exposes several functions, see *API docs*, that allow you to easily write checks for input/output testing. check50 comes with three builtin extensions: `c`, `py` and `flask`. These extensions add extra functionality for C, Python and Python's Flask framework to check50's core.

By design check50 is extensible. If you want to add support for other programming languages / frameworks and you are comfortable with Python please check out *Writing check50 extensions*.