APIs

1  Running check50                  1
2  API docs                        9
3  JSON specification              19
4  Writing check50 checks          23
5  Writing check50 extensions      37
6  Installation                    39
7  Usage                           41
8  Design                          43
9  Checks                          45
10 cli50                           47
11 compare50                       51
12 render50                        57
13 style50                         61
14 submit50                        65
15 cs50/baseimage                  69
16 cs50/check                      71
17 cs50/cli                        73
18 cs50/ide                        75
19 cs50/sandbox                    77
20 cs50/server                     79
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.

```
$ check50 cs50/problems/2018/x/caesar
Results for cs50/problems/2018/x/caesar generated by check50 v3.0.0
.:) caesar.c exists.
.:) caesar.c compiles.
   code failed to compile
.:) encrypts "a" as "b" using 1 as key
   can't check until a frown turns upside down
.:) encrypts "barfoo" as "yxccll" 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
```

### 1.5.2 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.

```json
{
  "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)

1.5. Output modes
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Passed</th>
<th>Log</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>encrypts_BARFOO_as_EDUIRR</td>
<td>encrypts &quot;BARFOO&quot; as &quot;EDUIRR&quot; using 3 as key</td>
<td>null</td>
<td>[]</td>
<td>can't check until a frown turns upside down</td>
</tr>
<tr>
<td>encrypts_BaRFoo_FeVJss</td>
<td>encrypts &quot;BaRFoo&quot; as &quot;FeVJss&quot; using 4 as key</td>
<td>null</td>
<td>[]</td>
<td>can't check until a frown turns upside down</td>
</tr>
<tr>
<td>encrypts_barfoo_as_onesbb</td>
<td>encrypts &quot;barfoo&quot; as &quot;onesbb&quot; using 65 as key</td>
<td>null</td>
<td>[]</td>
<td>can't check until a frown turns upside down</td>
</tr>
<tr>
<td>checks_for_handling_non_alpha</td>
<td>encrypts &quot;world, say hello!&quot; as &quot;iadxp, emk tqxxa!&quot; using 12 as key</td>
<td>null</td>
<td>[]</td>
<td>can't check until a frown turns upside down</td>
</tr>
<tr>
<td>handles_no_argv</td>
<td>handles lack of argv[1]</td>
<td>null</td>
<td>[]</td>
<td>can't check until a frown turns upside down</td>
</tr>
</tbody>
</table>

(continues on next page)
"data": {},
  "dependency": "compiles"
},

"version": "3.0.0"
2.1 check50

check50.import_checks(path)

Import checks module given relative path.

Parameters path (str) – relative path from which to import checks module

Returns the imported module

Raises

- FileNotFoundError – if path / .check50.yaml does not exist
- yaml.YAMLError – if path / .check50.yaml is not a valid YAML file

This function is particularly useful when a set of checks logically extends another, as is often the case in CS50’s own problems that have a “less comfy” and “more comfy” version. The “more comfy” version can include all of the “less comfy” checks like so:

```python
less = check50.import_checks("../less")
from less import *
```

Note: the __name__ of the imported module is given by the basename of the specified path (less in the above example).

check50.data(**kwargs)

Add data to the check payload

Params kwargs key/value mappings to be added to the check payload

Example usage:

```python
check50.data(time=7.3, mem=23)
```
check50.exists(*paths)

Assert that all given paths exist.

**Params** paths  files/directories to be checked for existence

**Raises** check50.Failure – if any path in paths does not exist

Example usage:

```python
check50.exists("foo.c", "foo.h")
```

check50.hash(file)

Hashes file using SHA-256.

**Parameters** file (str) – name of file to be hashed

**Return type** str

**Raises** check50.Failure – if file does not exist

ccheck50.include(*paths)

Copy files/directories from the check directory (check50.internal.check_dir), to the current directory

**Params** paths  files/directories to be copied

Example usage:

```python
check50.include("foo.txt", "bar.txt")
assert os.path.exists("foo.txt") and os.path.exists("bar.txt")
```

class check50.run(command, env={})

Run a command.

**Parameters**

- command (str) – command to be run
- env (dict) – environment in which to run command

By default, the command will be run using the same environment as check50, these mappings may be overriden via the env parameter:

```python
check50.run("./foo").stdin("foo").stdout("bar").exit(0)
check50.run("./foo", env={ "HOME": "/" }).stdin("foo").stdout("bar").exit(0)
```

ccheck50.exit(code=None, timeout=5)

Wait for process to exit or until timeout (5 sec by default) and asserts that process exits with code. If code is None, returns the code the process exited with.

..note:: In order to ensure that spawned child processes do not outlive the check that spawned them, it is good practice to call either method (with no arguments if the exit code doesn’t matter) or .kill() on every spawned process.

**Parameters**

- code (int) – code to assert process exits with
- timeout (int / float) – maximum number of seconds to wait for the program to end

**Raises** check50.Failure – if code is given and does not match the actual exitcode within timeout

Example usage:
```python
check50.run("./hello").exit(0)

code = check50.run("./hello").exit()
if code != 0:
    raise check50.Failure(f"expected exit code 0, not {code}"

kill()

Kill the process.

Child will first be sent a SIGHUP, followed by a SIGINT and finally a SIGKILL if it ignores the first two.

reject(timeout=1)

Check that the process survives for timeout. Useful for checking whether program is waiting on input.

Parameters

- **timeout** *(int / float)* – number of seconds to wait

Raises

- **check50.Failure** – if process ends before timeout

stdin(line, prompt=True, timeout=3)

Send line to stdin, optionally expect a prompt.

Parameters

- **line** *(str)* – line to be send to stdin
- **prompt** *(bool)* – boolean indicating whether a prompt is expected, if True absorbs all of stdout before inserting line into stdin and raises check50.Failure if stdout is empty
- **timeout** *(int / float)* – maximum number of seconds to wait for prompt

Raises

- **check50.Failure** – if prompt is set to True and no prompt is given

stdout(output=None, str_output=None, regex=True, timeout=3)

Retrieve all output from stdout until timeout (3 sec by default). If output is None, stdout returns all of the stdout outputted by the process, else it returns self.

Parameters

- **output** *(str)* – optional output to be expected from stdout, raises check50.Failure if no match
- **str_output** *(str)* – what will be displayed as expected output, a human readable form of output
- **regex** *(bool)* – flag indicating whether output should be treated as a regex
- **timeout** *(int / float)* – maximum number of seconds to wait for output

Raises

- **check50.Mismatch** – if output is specified and nothing that the process outputs matches it
- **check50.Failure** – if process times out or if it outputs invalid UTF-8 text.

Example usage:

```python
check50.run("./hello").stdout("[Hh]ello, world!?", "hello, world").exit()

output = check50.run("./hello").stdout()
if not re.match("[Hh]ello, world!?", output):
    raise check50.Mismatch("hello, world", output)
```
check50.log(line)
Add to check log

Parameters line (str) – line to be added to the check log

The check log is student-visible via the --log flag to check50.

exception check50.Failure(rationale, help=None)
Exception signifying check failure.

Parameters

• rationale (str) – message to be displayed capturing why the check failed
• help (str) – optional help message to be displayed

Example usage:

```python
out = check50.run("./cash").stdin("4.2").stdout()
if 10 not in out:
    help = None
if 11 in out:
    help = "did you forget to round your result?"
raise check50.Failure("Expected a different result", help=help)
```

exception check50.Mismatch(expected, actual, help=None)
Exception signifying check failure due to a mismatch in expected and actual outputs.

Parameters

• expected – the expected value
• actual – the actual value
• help (str) – optional help message to be displayed

Example usage:

```python
from re import match
expected = "[Hh]ello, world!\n"
actual = check50.run("./hello").stdout()
if not match(expected, actual):
    help = None
    if match(expected[:-1], actual):
        help = r"did you forget a newline (\n') at the end of your printf string? →"
raise check50.Mismatch("hello, world\n", actual, help=help)
```

check50.check(dependency=None, timeout=60)
Mark function as a check.

Parameters

• dependency (function) – the check that this check depends on
• timeout (int) – maximum number of seconds the check can run

When a check depends on another, the former will only run if the latter passes. Additionally, the dependent check will inherit the filesystem of its dependency. This is particularly useful when writing e.g., a compiles check that compiles a student’s program (and checks that it compiled successfully). Any checks that run the student’s program will logically depend on this check, and since they inherit the resulting filesystem of the check, they will immediately have access to the compiled program without needing to recompile.

Example usage:
@check50.check() # Mark 'exists' as a check
def exists():
    """hello.c exists"
    check50.exists("hello.c")

@check50.check(exists) # Mark 'compiles' as a check that depends on 'exists'
def compiles():
    """hello.c compiles"
    check50.c.compile("hello.c")

@check50.check(compiles)
def prints_hello():
    """prints "Hello, world!\n"
    # Since 'prints_hello', depends on 'compiles' it inherits the compiled binary
    check50.run("./hello").stdout("\[Hh\]ello, world!?\n", "hello, world\n").exit()

exception check50.EOF (value)
    Raised when EOF is read from a child. This usually means the child has exited.

2.2 check50.c

ccheck50.c.CC = 'clang'
Default compiler for check50.c.compile()

ccheck50.c.CFLAGS = {'ggdb': True, 'lm': True, 'std': 'c11'}
Default CFLAGS for check50.c.compile()

ccheck50.c.compile(*files, exe_name=None, cc='clang', **cflags)
Compile C source files.

Parameters

• files  – filenames to be compiled
• exe_name  – name of resulting executable
• cc  – compiler to use (check50.c.CC by default)
• cflags  – additional flags to pass to the compiler

Raises

• check50.Failure  – if compilation failed (i.e., if the compiler returns a non-zero exit status).
• RuntimeError  – if no filenames are specified

If exe_name is None, check50.c.compile() will default to the first file specified sans the .c extension:

ccheck50.c.compile("foo.c", "bar.c") # clang foo.c bar.c -o foo -std=c11 -ggdb -lm

Additional CFLAGS may be passed as keyword arguments like so:

ccheck50.c.compile("foo.c", "bar.c", lcs50=True) # clang foo.c bar.c -o foo -
   -std=c11 -ggdb -lm -lcs50

In the same vein, the default CFLAGS may be overridden via keyword arguments:
check50.c.compile("foo.c", "bar.c", std="c99", lm=False) # clang foo.c bar.c -o
→ foo -std=c99 -ggdb

check50.c.valgrind(command, env={})
Run a command with valgrind.

Parameters
• command (str) – command to be run
• env (str) – environment in which to run command

Raises check50.Failure – if, at the end of the check, valgrind reports any errors

This function works exactly like check50.run(), with the additional effect that command is run through valgrind and valgrind’s output is automatically reviewed at the end of the check for memory leaks and other bugs. If valgrind reports any issues, the check is failed and student-friendly messages are printed to the log.

Example usage:
check50.c.valgrind("./leaky").stdin("foo").stdout("bar").exit(0)

Note: It is recommended that the student’s code is compiled with the -ggdb flag so that additional information, such as the file and line number at which the issue was detected can be included in the log as well.

2.3 check50.flask

class check50.flask.app(path='application.py', app_name='app')
Spawn a Flask app.

Parameters
• path (str) – path to python file containing Flask app
• app_name – name of Flask app in file

Example usage:
check50.flask.app("application.py").get("/").status(200)

content (output=None, str_output=None, **kwargs)
Searches for output regex within HTML page. kwargs are passed to BeautifulSoup’s find function to filter for tags.

get (route=None, data=None, params=None, follow_redirects=True)
Send GET request to app.

Parameters
• route (str) – route to send request to
• data (dict) – form data to include in request
• params – URL parameters to include in request
• follow_redirects (bool) – enable redirection (defaults to True)

Returns self
Raises `check50.Failure` – if Flask application throws an uncaught exception

Example usage:

```python
check50.flask.app("application.py").get("/buy", params={"q": "02138"}).content()
```

`post (route, data=None, params=None, follow_redirects=True)`

Send POST request to app.

Parameters

- `route (str)` – route to send request to
- `data (dict)` – form data to include in request
- `params` – URL parameters to include in request
- `follow_redirects (bool)` – enable redirection (defaults to True)

Raises `check50.Failure` – if Flask application throws an uncaught exception

Example usage:

```python
check50.flask.app("application.py").post("/buy", data={"symbol": "GOOG", "shares": 10}).status(200)
```

`raw_content (output=None, str_output=None)`

Searches for `output` regex match within content of page, regardless of mimetype.

`status (code=None)`

Check status code in response returned by application. If `code` is not None, assert that `code` is returned by application, else simply return the status code.

Parameters `code (int)` – code to assert that application returns

Example usage:

```python
check50.flask.app("application.py").status(200)
status = check50.flask.app("application.py").get("/").status()
if status != 200:
    raise check50.Failure(f"expected status code 200, but got {status}")
```

2.4 check50.py

`check50.py.append_code (original, codefile)`

Append the contents of one file to another.

Parameters

- `original (str)` – name of file that will be appended to
- `codefile (str)` – name of file that will be appended

This function is particularly useful when one wants to replace a function in student code with their own implementation of one. If two functions are defined with the same name in Python, the latter definition is taken so overwriting a function is as simple as writing it to a file and then appending it to the student’s code.

Example usage:
# Include a file containing our own implementation of a lookup function.
check50.include("lookup.py")

# Overwrite the lookup function in helpers.py with our own implementation.
check50.py.append_code("helpers.py", "lookup.py")

check50.py.compile(file)
Compile a Python program into byte code

    Parameters file – file to be compiled

    Raises check50.Failure – if compilation fails e.g. if there is a SyntaxError

check50.py.import_(path)
Import a Python program given a raw file path

    Parameters path (str) – path to python file to be imported

    Raises check50.Failure – if path doesn’t exist, or if the Python file at path throws an exception when imported.

## 2.5 check50.internal

Additional check50 internals exposed to extension writers in addition to the standard API

check50.internal.CONFIG_LOADER = <lib50.config.Loader object>
lib50 config loader

exception check50.internal.Error
Exception for internal check50 errors.

class check50.internal.Register
Class with which functions can be registered to run before / after checks. check50.internal.register should be the sole instance of this class.

    after_check (func)
    Run func once at the end of the check, then discard func.

      Parameters func – callback to run after check

      Raises check50.internal.Error – if called when no check is being run

    after_every (func)
    Run func at the end of every check.

      Parameters func – callback to be run after every check

      Raises check50.internal.Error – if called when a check is being run

    before_every (func)
    Run func at the start of every check.

      Parameters func – callback to be run before every check

      Raises check50.internal.Error – if called when a check is being run

check50.internal.check_dir = None
Directory containing the check and its associated files

check50.internal.check_running = False
Boolean that indicates if a check is currently running
check50.internal.compile_checks(checks, prompt=False, out_file='__init__.py')

Compile YAML checks to a Python file

**Parameters**

- `checks` – YAML checks read from config
- `prompt` (bool) – prompt user if `out_file` already exists
- `out_file` (str) – file to write compiled checks

**Returns** out_file

**Return type** str

check50.internal.import_file(name, path)

Import a file given a raw file path.

**Parameters**

- `name` (str) – Name of module to be imported
- `path` (str / Path) – Path to Python file

check50.internal.load_config(check_dir)

Load configuration file from `check_dir / '.cs50.yaml'`, applying defaults to unspecified values.

**Parameters** check_dir (str / Path) – directory from which to load config file

**Return type** dict

check50.internal.register = <check50.internal.Register object>

Sole instance of the check50.internal.Register class

check50.internal.run_dir = None

Temporary directory in which check is being run
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:

```bash
cHECK50 cs50/problems/2018/x/caesar -o json
```

Produces the following:

```json
{
   "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 -Ics50...",
            "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}"
      }
   ]
}
```
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** (list of string) contains the log accrued 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 student-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** *(list)* contains the stack trace of the thrown exception.

- **data** *(object)* contains any additional data the exception may carry in its *payload* attribute.
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:

```
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:
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
        exit: 0 # expect program to exit with exitcode 0
```

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:

```
check50:
  checks:
    multiline hello world:
      - run: python3 multi_hello.py
        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

```python
import check50 # import the check50 module

@check50.check() # tag the function below as check50 check
def exists(): # the name of the check
    # this is what you will see when running check50
    check50.exists("hello.py") # the actual check

@check50.check(exists) # only run this check if the exists check has passed
def prints_hello():
    # prints "hello, world\n"
    check50.run("python3 hello.py").stdout("[Hh]ello, world!").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

```python
import check50

@check50.check()
def prints_hello():
    # prints "hello, world\n"
    from re import match
    expected = ":[Hh]ello, world!\n"
    actual = check50.run("python3 hello.py").stdout()
    if not match(expected, actual):
        help = None
        if match(expected[:-1], actual):
            help = r"did you forget a newline ('\n') at the end of your printf string? \n"
        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.
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.

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:

import check50

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

Check stdout for an exact string:

@check50.check(exists)
def prints_hello_world():
    (continues on next page)
Check stdout for a rough match:

Listing 15: __init__.py

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

Put something in stdin, expect it in stdout:

Listing 16: __init__.py

```python
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

```python
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

```python
import check50

@check50.check()
def at least_one_match():
```

4.7. Python check examples
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`

```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`

```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`

```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 globed, 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**
```yaml
check50:
  files:
    - !exclude "*.pyc"
```
Excludes all files ending with `.pyc`.

**Listing 23: .cs50.yaml**
```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**
```yaml
check50:
  files:
    - !exclude "*"
    - !include "source/"
```
Exclude all files, but include all files in the source directory.

**Listing 25: .cs50.yaml**
```yaml
check50:
  files:
    - !exclude "build/"
    - !exclude "docs/"
```
Include everything, but exclude everything in the build and docs directories.

**Listing 26: .cs50.yaml**
```yaml
check50:
  files:
    - !exclude "*"
    - !include "source/"
    - !exclude "*.pyc"
```
Exclude everything, include everything from the source directory, but exclude all files ending with `.pyc`. 

---

4.8. Configuring check50
Include everything, but any files ending on `.pyc` within the source directory. The `**` here pattern matches any directory.

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

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 `.py`-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.

Has check50 install both `pyyaml` and `flask` via `pip`.
Listing 31: `.cs50.yaml`

```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
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**:

  ```bash
  wget https://raw.githubusercontent.com/cs50/check50/examples/solutions/hello_js/hello.js
cHECK50 cs50/check50/examples/js/hello
  ```

- **line.js**:

  ```bash
  wget https://raw.githubusercontent.com/cs50/check50/examples/solutions/line_js/line.js
cHECK50 cs50/check50/examples/js/line
  ```

- **addition.js**:

  ```bash
  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.
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.
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
```
• **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.
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.
cli50 is a command-line tool via which you can (easily!) mount a directory within a container running cs50/cli, a Docker image based on Ubuntu 18.04, a popular distribution of Linux. We use cli50 to work on CS50’s own apps in a standard, contained environment, without having to install apps’ dependencies on our own Macs and PCs. It provides a command-line environment similar, but not quite identical to, a terminal window on CS50 IDE.

10.1 Installation

1. Install Docker, if you haven’t already.
2. Install Python 3.6 or later, if you haven’t already.
3. Install pip, if you haven’t already.
4. Install cli50 itself:

   ```shell
   pip3 install cli50
   ```

10.1.1 Upgrading

   ```shell
   pip install --upgrade cli50
   ```

10.2 Usage

   ```shell
   ```

   positional arguments:
DIRECTORY directory to mount, else $PWD

optional arguments:
-h, --help show this help message and exit
-d DOTFILE, --dotfile DOTFILE dotfile in your $HOME to mount read-only in container's $HOME
-f, --fast skip autoupdate
-j, --jekyll serve Jekyll site
-l [CONTAINER], --login [CONTAINER] log into CONTAINER
-S, --stop stop any containers
-t TAG, --tag TAG start cs50/cli:TAG, else cs50/cli:latest
-u, --update update only
-V, --version show program's version number and exit

10.3 Examples

10.3.1 Mount current working directory

cli50

Your current working directory will be mounted in /mnt within the container.

10.3.2 Mount any directory

Assuming path/to/directory is that directory's absolute path, you can mount it within the container as follows:

cli50 path/to/directory

The directory will be mounted in /mnt within the container.

10.3.3 Skip autoupdate

By default, cli50 autoupdates (i.e., pulls) cs50/cli, the Docker image on which it's based, which can be time-consuming on slow internet connections. You can skip autoupdate as follows:

cli50 -f

or

cli50 --fast

10.3.4 Update only

You can update (i.e., pull) cs50/cli, the Docker image on which cli50 is based, without actually starting a container as follows:
10.3.5 Log into running container

If a container (based on any Docker image, cs50/cli or otherwise) is already running, you can spawn a login shell within it as follows:

`cli50 -l`

If multiple containers are running, you’ll be asked, yes or no, in reverse-chronological order, into which container you’d like to log in.

10.3.6 Mount ~/.* in container’s $HOME

You can mount, read-only, a dotfile that’s in your $HOME (e.g., ~/.vimrc) inside of a container’s $HOME as follows:

`cli50 -d .vimrc`

Directories (e.g., ~/.vim) are supported as well:

`cli50 -d .vim`

10.3.7 Serve Jekyll site

When developing a Jekyll site, you can serve (and continually rebuild) it within a container as follows:

`cli50 -j`

or

`cli50 --jekyll`

The site will be served at http://0.0.0.0:8080/ within the container, but port 8080 will be published on pseudorandom port on the host (i.e., your computer). If cli50 outputs, e.g.,

```
0.0.0.0:#####->8080/tcp, ...
```

on startup, you can access the site at http://0.0.0.0:#####/ on your computer.

10.4 Source Code

https://github.com/cs50/cli50
11.1 API docs

11.1.1 compare50

**class** `compare50.Comparator`
Abstract base class for `compare50` comparators which specify how submissions should be scored and compared.

**compare** *(scores, ignored_files)*
Given a list of scores and a list of distro files, perform an in-depth comparison of each submission pair and return a corresponding list of `compare50.Comparisons`

**score** *(submissions, archive_submissions, ignored_files)*
Given a list of submissions, a list of archive submissions, and a set of distro files, return a list of `compare50.Scores` for each submission pair.

**class** `compare50.Comparison`(sub_a, sub_b, span_matches=NOTHING, ignored_spans=NOTHING)

**Variables**
- `sub_a` – the first submission
- `sub_b` – the second submission
- `span_matches` – a list of pairs of matching `compare50.Spans`, wherein the first element of each pair is from `sub_a` and the second is from `sub_b`.
- `ignored_spans` – a list of `compare50.Spans` which were ignored (e.g. because they matched distro files)

Represents an in-depth comparison of two submissions.

**exception** `compare50.Error`
Base class for `compare50` errors.

**class** `compare50.File`(name, submission)
Variables

- **name** – file name (path relative to the submission path)
- **submission** – submission containing this file
- **id** – integer that uniquely identifies this file (files with the same path will always have the same id)

Represents a single file from a submission.

```
classmethod get(id)
Find File with given id
```

```
lexer()
Determine which Pygments lexer should be used.
```

```
path
The full path of the file
```

```
read(size=-1)
Open file, read size bytes from it, then close it.
```

```
tokens()
Returns the preprocessed tokens of the file.
```

```
unprocessed_tokens()
Get the raw tokens of the file.
```

```
class compare50.Pass
Abstract base class for compare50 passes, which are essentially ways for compare50 to compare submissions. Subclasses must define a list of preprocessors (functions from tokens to tokens which will be run on every file compare50 recieves) as well as a comparator (used to score and compare the preprocessed submissions).
```

```
class compare50.Score(sub_a, sub_b, score=0)
```

```
Variables

- **sub_a** – the first submission
- **sub_b** – the second submission
- **score** – a number indicating the similarity between sub_a and sub_b (higher meaning more similar)

A score representing the similarity of two submissions.

```
class compare50.Span(file, start, end)
```

```
Variables

- **file** – the ID of the File containing the span
- **start** – the character index of the first character in the span
- **end** – the character index one past the end of the span

Represents a range of characters in a particular file.

```
class compare50.Submission(path, files, preprocessor=<function Submission.<lambda>>, is_archive=False)
```

```
Variables

- **path** – the file path of the submission
- **files** – list of compare50.File objects contained in the submission
```
• **preprocessor** – A function from tokens to tokens that will be run on each file in the submission

• **id** – integer that uniquely identifies this submission (submissions with the same path will always have the same id).

Represents a single submission. Submissions may either be single files or directories containing many files.

```python
classmethod get(id)
Retrieve submission corresponding to specified id
```

```python
class compare50.Token(start, end, type, val)
Variables
• **start** – the character index of the beginning of the token
• **end** – the character index one past the end of the token
• **type** – the Pygments token type
• **val** – the string contents of the token
```

A result of the lexical analysis of a file. Preprocessors operate on Token streams.

```python
class compare50.compare(scores, ignored_files, pass_)
Parameters
• **scores** ([compare50.Score]) – Scored submission pairs to be compared more granu-
larly
• **ignored_files** ([compare50.File]) – files containing distro code
• **pass** (compare50.Pass) – pass whose comparator should be use to compare the sub-
missions

Returns Compare50Results corresponding to each of the given scores

Return type [compare50.Compare50Result]
```

Performs an in-depth comparison of each submission pair and returns a corresponding list of compare50. compare50Results.

```python
class compare50.expand(span_matches, tokens_a, tokens_b)
Parameters
• **span_matches** ([([compare50.Span, compare50.Span]]) – span pairs to be ex-
panded wherein the first element of every pair is from the same file and the second element of every pair is from the same file
• **tokens_a** ([compare50.Token]) – the tokens of the file corresponding to the first ele-
ment of each span_match
• **tokens_b** ([compare50.Token]) –

param tokens_a the tokens of the file corresponding to the first element of each
span_match

Returns A new list of maximally expanded span pairs

Return type [(compare50.Span, compare50.Span)]
```

Expand all span matches. This is useful when e.g. two spans in two different files are identical, but there are tokens before/after these spans that are also identical between the files. This function expands each of these spans to include these additional tokens.
compare50.missing_spans (file, original_tokens=None, processed_tokens=None)

Parameters

• file (compare50.File) – file to be examined
• original_tokens – the unprocessed tokens of file. May be optionally specified if file has been tokenized elsewhere to avoid tokenizing it again.
• processed_tokens – the result of preprocessing the tokens of file. May optionally be specified if file has been preprocessed elsewhere to avoid doing so again.

Returns The spans of file that were stripped by the preprocessor.

Return type [compare50.Span]

Determine which parts of file were stripped out by the preprocessor.

compare50.rank (submissions, archive_submissions, ignored_files, pass_, n=50)

Parameters

• submissions ([compare50.Submission]) – submissions to be ranked
• archive_submissions ([compare50.Submission]) – archive submissions to be ranked
• ignored_files ([compare50.File]) – files containing distro code
• pass (compare50.Pass) – pass whose comparator should be use to rank the submissions
• n (int) – number of submission pairs to return

Returns the top n submission pairs

Return type [compare50.Score]

Rank submissions, return the top n most similar pairs

11.1.2 compare50.passes

class compare50.passes.structure

Compares code structure by removing whitespace and comments; normalizing variable names, string literals, and numeric literals; and then running the winnowing algorithm.

class compare50.passes.exact

Removes all whitespace, then uses the winnowing algorithm to compare submissions.

class compare50.passes.misspellings

Compares comments for identically misspelled English words.

11.1.3 compare50.preprocessors

compare50.preprocessors.by_character (tokens)

Make a token for each character.

compare50.preprocessors.comments (tokens)

Remove all tokens that aren’t comments.

compare50.preprocessors.extract_identifiers (tokens)

Remove all tokens that don’t represent identifiers.
compare50.preprocessors.normalize_builtin_types(tokens)
 Normalize builtin type names

compare50.preprocessors.normalize_case(tokens)
 Make all tokens lower case.

compare50.preprocessors.normalize_identifiers(tokens)
 Replace all identifiers with v

compare50.preprocessors.normalize_numeric_literals(tokens)
 Replace numeric literals with their types.

compare50.preprocessors.normalize_string_literals(tokens)
 Replace string literals with empty strings.

compare50.preprocessors.split_on_whitespace(tokens)
 Split values of tokens on whitespace into new tokens

compare50.preprocessors.strip_comments(tokens)
 Remove all comments from tokens.

compare50.preprocessors.strip_whitespace(tokens)
 Remove all whitespace from tokens.

compare50.preprocessors.text_printer(tokens)
 Print token values. Useful for debugging.

compare50.preprocessors.token_printer(tokens)
 Print each token. Useful for debugging.

compare50.preprocessors.words(tokens)
 Split tokens into tokens containing just one word.

11.1.4 compare50.comparators

class compare50.comparators.Misspellings(dictionary)

call compare(scores, ignored_files)
 Given a list of scores and a list of distro files, perform an in-depth comparison of each submission pair and
 return a corresponding list of compare50.Comparisons

score(submissions, archive_submissions, ignored_files)
 Number of identically misspelled words.

class compare50.comparators.Winnowing(k, t)
 Comparator utilizing the (robust) Winnowing algorithm as described https://theory.stanford.edu/~aiken/
 publications/papers/sigmod03.pdf

Parameters
 t (int) – the guarantee threshold; any matching sequence of tokens of length at least
 t is guaranteed to be matched

Parameters
 k the noise threshold; any matching sequence of tokens shorter than this will be ignored

compare(scores, ignored_files)
 Given a list of scores and a list of distro files, perform an in-depth comparison of each submission pair and
 return a corresponding list of compare50.Comparisons

score(submissions, archive_submissions, ignored_files)
 Number of matching k-grams.

TODO
11.2 Installation

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

To install compare50 under Linux / OS X:

```
pip install compare50
```

Under Windows, please . Then install compare50 within the subsystem.

11.3 Usage

Usage:

```
usage: compare50 [-h] [-a ARCHIVE [ARCHIVE ...]] [-d DISTRO [DISTRO ...]]
                [-p PASSES [PASSES ...]] [-i INCLUDE [INCLUDE ...]]
                [-x EXCLUDE [EXCLUDE ...]] [--list] [-o OUTPUT] [-v]
                [-n MATCHES] [--profile] [--debug]
                submissions [submissions ...]
```

positional arguments:
  submissions Paths to submissions to compare

optional arguments:
  -h, --help show this help message and exit
  -a ARCHIVE [ARCHIVE ...], --archive ARCHIVE [ARCHIVE ...] Paths to archive submissions. Archive submissions are not compared against other archive submissions, only against regular submissions.
  -d DISTRO [DISTRO ...], --distro DISTRO [DISTRO ...] Paths to distribution files. Contents of these files are stripped from submissions.
  -p PASSES [PASSES ...], --passes PASSES [PASSES ...] Specify which passes to use. compare50 ranks only by the first pass, but will render views for every pass.
  -i INCLUDE [INCLUDE ...], --include INCLUDE [INCLUDE ...] Globbing patterns to include from every submission. Includes everything (*) by default. Make sure to quote your patterns to escape any shell globbing!
  -x EXCLUDE [EXCLUDE ...], --exclude EXCLUDE [EXCLUDE ...] Globbing patterns to exclude from every submission. Nothing is excluded by default. Make sure to quote your patterns to escape any shell globbing!
  --list List all available passes and exit.
  -o OUTPUT, --output OUTPUT location of compare50's output
  -v, --verbose display the full tracebacks of any errors
  -n MATCHES number of matches to output
  --profile profile compare50 (development only, requires lineProfiler, implies debug)
  --debug don't run anything in parallel, disable progress bar

TODO
render50 is a command-line tool with which you can generate syntax-highlighted PDFs of source code. In years past, CS50 generated PDFs of students’ submissions so that teaching fellows (TFs) could annotate the PDFs with typed feedback. These days, CS50 generates PDFs of lectures’ source code so that students can annotate them during lectures and so that David has a printout of each lecture’s source code in front of him during lectures!

PDFs can be annotated (for free) with:

- Adobe Reader DC on macOS and Windows
- Evince Document Viewer on Ubuntu Linux
- Preview on macOS

### 12.1 Installation

1. Install *Docker*, if you haven’t already.
2. Install *Python 3.6* or later, if you haven’t already.
3. Install *pip*, if you haven’t already.
4. Install *render50* itself:
   ```plaintext
   pip3 install render50
   ```
5. Install WeasyPrint’s dependencies.

### 12.1.1 Upgrading

```plaintext
pip install --upgrade render50
```
12.2 Usage

Usage

```
             [-x EXCLUDE] [-y] [-V]
              input [input ...]
```

A command-line tool that renders source code as a PDF.

Positional arguments:

- input: file or URL to render

Optional arguments:

- **-h, --help**: show this help message and exit
- **-b, --browser**: render as a browser would
- **-C, --no-color**: disable syntax highlighting
- **-i INCLUDE, --include INCLUDE**: pattern to include
- **-o OUTPUT, --output OUTPUT**: file to output
- **-r, --recursive**: recurse into directories
- **-s SIZE, --size SIZE**: size of page, per https://developer.mozilla.org/en-US/docs/Web/CSS/@page/size
- **-x EXCLUDE, --exclude EXCLUDE**: pattern to exclude
- **-y, --side-by-side**: render inputs side by side
- **-V, --version**: show program's version number and exit

12.3 Examples

12.3.1 Render a single file

To generate a PDF (e.g., `foo.pdf`) from a source file (e.g., `bar.c`), execute:

```
render50 -o foo.pdf bar.c
```

12.3.2 Render multiple files

To generate a PDF (e.g., `foo.pdf`) from multiple source files (e.g., `bar.c` and `baz.c`), execute:

```
render50 -o foo.pdf bar.c baz.c
```

The source files will be rendered in the order in which they’re specified on the command line.

12.3.3 Include files

To generate a PDF (e.g., `foo.pdf`) from multiple source files, including only those that match some pattern (e.g., `*.c`), execute:

```
render50 -i "*.c" -o foo.pdf *
```

Take care to quote (or escape with \) any patterns with wildcards, lest your shell glob the pattern before `render50` can.
12.3.4 Exclude files

To generate a PDF (e.g., foo.pdf) from multiple source files, excluding those that match some pattern (e.g., *.h), execute:

```bash
render50 -x "*.h" -o foo.pdf *
```

As before, take care to quote (or escape with \) any patterns with wildcards, lest your shell glob the pattern before render50 can.

12.3.5 Render two files side by side

To generate a PDF (e.g., foo.pdf) with two source files (e.g., bar.c and baz.c) side by side, execute:

```bash
render50 -o foo.pdf -y bar.c baz.c
```

12.3.6 Render files recursively

To recurse into directories, invoke render50 with -r.

12.3.7 Disable syntax highlighting

To disable syntax highlighting, invoke render50 with -C.

12.3.8 Rendering URLs

To generate a PDF (e.g., foo.pdf) with a source file at a URL (e.g., https://github.com/cs50/render50/blob/master/render50), execute:

```bash
render50 -o foo.pdf https://github.com/cs50/render50/blob/master/render50
```

Note that URLs on github.com are handled specially: URLs of the form https://github.com/*/*/*/blob/* are resolved to https://github.com/*/*/*/*raw/ so that the file is downloaded from raw.githubusercontent.com. And URLs of the form https://gist.github.com/*/*/*/*/raw (and https://gist.github.com/*/*/*/raw/*) are resolved to https://gist.github.com/*/*/*/raw/* so that the file is downloaded from gist.githubusercontent.com.

12.3.9 Overriding page size

By default, render50 outputs letter-sized (8.5" × 11") pages in landscape orientation. To override that default, invoke render50 with --size SIZE, where SIZE is as prescribed by CSS. Supported values for SIZE thus include:

- letter landscape
- letter portrait
- A4 landscape
- A4 portrait
12.4 Source Code

https://github.com/cs50/render50
style50 is a command-line tool with which you can check your code for consistency with CS50’s style guide (for C). If your code isn’t styled consistently, style50 will summarize the changes you should make to your code, as by highlighting in green characters you should add and highlighting in red characters you should delete.

For instance, consider the code below, wherein the call to printf isn’t properly indented.

```c
#include <stdio.h>
int main(void)
{
    printf("hello, world\n");
}
```

Given that code as input, style50 will output

wherein highlighted are four spaces that should be added for style’s sake.

On the other hand, consider the code below, wherein the curly braces are unnecessarily indented.

```c
#include <stdio.h>
int main(void)
{
    printf("hello, world\n");
}
```

Given that code as input, style50 will output

wherein highlighted are four spaces that should be deleted for style’s sake.

### 13.1 Usage

To check your code’s style, execute
where `file` is the (path to some) file whose style you’d like to check.

### 13.1.1 Modes

By default, `style50` operates in `character` mode, but you can specify other modes with `-o` or `--output`. Consider the (poorly styled) file below, `hello.c`, for a look at these modes.

```c
#include <stdio.h>
int main(void)
{
    printf("hello, world\n");
}
```

##### character

In `character` mode, `style50` compares its input against CS50’s style guide character by character. Were you to run

```bash
style50 --output character hello.c
```

or just

```bash
style50 -o character hello.c
```

or even just

```bash
style50 hello.c
```

you would see the below.

##### split

In `split` mode, `style50` displays its input and output side by side. Were you to run

```bash
style50 --output split hello.c
```

or just

```bash
style50 -o split hello.c
```

you would see the below.

##### unified

In `unified` mode, `style50` displays its output line by line, akin to `git-diff`. Were you to run

```bash
style50 --output unified hello.c
```

or just
you would see the below.

### 13.2 Installation

**style50** is already installed for you in **CS50 IDE**, so no need to install it yourself; simply use it as directed!

If you’d like to install **style50** on your own Mac or PC, so that you can check your code’s style without using CS50 IDE, you’ll need a command-line environment:

- If running Linux or the like, you already have one! Open a terminal window in your usual way.
- If running Mac OS, you already have one! Open **Applications > Utilities > Terminal**.
- If running Windows, you’ll need to install the **Windows Subsystem for Linux**, which is only supported on Windows 10. Once installed, run **bash**.

To install **style50** within that command-line environment:

1. **Install Python** 2.7 or higher, if not already installed.

2. **Install pip**, as via

   ```
   sudo easy_install pip
   ```

   if not already installed.

3. **Execute**

   ```
   sudo pip3 install style50
   ```

   to install **style50** itself.

4. **Install Artistic Style 3.0**. If running a Debian-based operating system (e.g., Ubuntu Linux), simply run

   ```
   add-apt-repository ppa:cs50/ppa
   apt-get update
   apt-get install astyle
   ```

   to install CS50’s own compiled version of **astyle**.

### 13.3 Upgrading

Execute

```
sudo pip install --upgrade style50
``` to upgrade **style50**, once installed.

### 13.4 Source Code

https://github.com/cs50/style50
submit50 is a command-line tool with which you can submit work (e.g., problem sets) to a course (e.g., CS50). It's based on git, a "distributed version control system" that allows you to save different versions of files without having to give each version a unique filename (as you might be wont to do on your own Mac or PC!). Via submit50 and, in turn, git can you thus submit work multiple times (i.e., multiple versions thereof).

When you run submit50, your files are “pushed” (i.e., uploaded) to CS50's “organization” (also named “submit50”) on GitHub, a popular service via which developers (like you!) can share code. Your files are stored in a “repository” (a folder, essentially) to which only you and some of CS50’s staff have access (and anyone else to whom you grant access). Your work can thus be reviewed and scored in one central place, whether you wrote it in CS50 IDE or elsewhere!

### 14.1 Installation

1. Install Python 3.6 or later, if you haven’t already.
2. Install pip, if you haven’t already.
3. Install submit50 itself:

   ```bash
   pip3 install submit50
   ```

### 14.1.1 Upgrading

```bash
pip install --upgrade submit50
```
14.2 Usage


positional arguments:
  slug  prescribed identifier of work to submit

optional arguments:
  -h, --help    show this help message and exit
  --logout  logout of submit50
  -v, --verbose show commands being executed
  -V, --version show program's version number and exit

14.3 Examples

14.3.1 Submitting with submit50

To submit work with submit50, cd to the work's directory and execute

```
submit50 slug
```

where `slug` is the unique identifier for the work you're submitting, as prescribed by the course (as in a problem's specification). Although the `slug` might resemble the path to a directory, it's simply a unique identifier, independent of your own work's location. If you've not recently run `submit50` (within the past week), you might be prompted to log in with your GitHub username and password. (Per the source code for `submit50`, your username and password are sent only to GitHub, not to CS50's own servers.) You will then be prompted to confirm whether you indeed want to submit one or more files from your current directory, unless you're missing one or more required files, in which case `submit50` will instead exit without submitting anything.

Via SSH

By default, `submit50` pushes your work to GitHub via HTTPS, which requires your GitHub username and password, which is why `submit50` prompts you for both at least once per week. If you'd prefer not to provide `submit50` with your GitHub username and password at all, you can instead push your work to GitHub via SSH. Configure your workspace on CS50 IDE (or your own computer) as follows.

1. Generate an SSH key and add it to `ssh-agent`.

2. Add the SSH key to your GitHub account.

3. Execute

```
ssh -T -p443 git@ssh.github.com
```

To add `ssh.github.com` to the list of known hosts for `ssh`, answering “yes” if prompted whether you’re sure you want to continue connecting. If all goes well, you should see the message

```
Hi <USERNAME>! You've successfully authenticated, but GitHub does not provide shell access.
Connection to github.com closed.
```
(where <USERNAME> is your GitHub username).
Thereafter, you should be able to run submit50 without ever being prompted for your GitHub username or password.

### 14.3.2 Submitting without submit50

If comfortable with git, you can submit work without submit50. Simply push your work to the expected branch (i.e., the work’s prescribed slug) of https://github.com/submit50/jharvard (or git@github.com:submit50/jharvard.git), where jharvard is your own GitHub username. To get started, either clone that repository or add it to an existing repository as a remote.

On each such branch, take care to create a .gitignore file based on https://github.com/cs50/checks/raw/master/slug/submit50/exclude, where slug is as before, so that you don’t submit files that submit50 would otherwise ignore.

### 14.4 Implementation Details

To see how submit50 uses git underneath the hood, execute

```
submit50 -v slug
```

or

```
submit50 --verbose slug
```

where slug is the unique identifier for the work you’re submitting.

### 14.5 FAQs

#### 14.5.1 Do I need to provide submit50 with my GitHub username and password?

Nope, you can instead authenticate via SSH.

#### 14.5.2 If I use git locally, will submit50 affect my local repository?

Nope, submit50 uses its own GIT_DIR (in /tmp). It will ignore any .git directory that you might have locally.

#### 14.5.3 How does submit50 remember my GitHub password?

submit50 remembers your username and password in RAM using git-credential-cache. Your password is never stored on disk or transmitted elsewhere.

### 14.6 Source Code

https://github.com/cs50/submit50
cs50/baseimage is a Docker image on Docker Hub on which cs50/cli and cs50/check are based.
cs50/check is the Docker image on Docker Hub used by submit.cs50.io to run check50.
cs50/cli

cs50/cli is the Docker image on Docker Hub used by cli50.
cs50/ide

cs50/ide is the Docker image on Docker Hub used by CS50 IDE.
cs50/sandbox is a Docker image on Docker Hub with which you can create a container nearly identical to those used by CS50 Sandbox.
cs50/server is a Docker image on Docker Hub with which you can (easily!) serve websites with, optionally, back ends implemented in JavaScript, PHP, Python, or Ruby. (We use it to serve CS50’s own apps on AWS Elastic Beanstalk!) Essentially, it’s a lightly customized installation of Passenger, an app server, to which we’ve added support for PHP (for some of CS50’s older web apps). It also facilitates configuration of Nginx, the web server used by Passenger in Standalone mode, via two files, httpd.conf and server.conf. The image itself is based on cs50/cli, which, in turn, is based on cs50/baseimage, which, in turn, is based on Ubuntu 18.04, a popular distribution of Linux.

### 20.1 Usage

Assuming you already have Docker installed, base your own Dockerfile on cs50/server as follows, exposing TCP port 8080, the server’s default:

```
FROM cs50/server
EXPOSE 8080
```

Then ensure your app is structured as follows.

- If your app’s back end is implemented in Meteor
  - in bundled/packaged mode, ensure you have a file called app.js (your app’s entry point file) in the same directory as your Dockerfile.
  - in non-bundled/packaged mode, ensure you have a file called .meteor in the same directory as your Dockerfile.

- If your app’s back end is implemented in Node.js, ensure you have a file called app.js (your app’s entry point file) in the same directory as your Dockerfile.

- If your website’s back end is implemented in PHP, ensure that you have a directory called public in the same directory as your Dockerfile, inside of which are any PHP files meant to be served publicly.

- If your website’s back end is implemented in Python, ensure you have a (WSGI) file called passenger_wsgi.py, formatted as prescribed, in the same directory as your Dockerfile.
• If your website’s back end is implemented in Ruby (or Ruby on Rails), ensure you have a file called `config.ru`, formatted as prescribed, in the same directory as your Dockerfile.

• If your website does not have a back end, only a front end implemented in HTML (presumably with CSS and/or JavaScript), ensure that you have a directory called `public` in the same directory as your Dockerfile, inside of which are any HTML (and CSS and/or JavaScript) files meant to be served publicly.

20.2 Configuration

20.2.1 APPLICATION_ENV

If you set an environment called `APPLICATION_ENV` to a value of `dev`, as via a `docker-compose.yml` file, `cs50/server` (and, in turn, Passenger) will restart your application’s back end after every HTTP request (by creating a temporary file called `tmp/always_restart.txt`), thereby ensuring that any changes you make to files are noticed (and not cached).

20.2.2 Entry Point

By default, `cs50/server` looks for a file called `app.js`, `config.ru`, `.meteor`, or `passenger_wsgi.py` per its `Usage`. To configure `cs50/server` to use some other file as an app’s entry point, adjust your Dockerfile as follows, where `app_type` is as prescribed and `startup_file` is the relative path of the file to use.

```
FROM cs50/server
EXPOSE ####
...
CMD passenger start --app-type app_type --startup-file startup_file
```

20.2.3 Nginx

You can customize `cs50/server`’s installation of Nginx by adding directives as follows.

- To add directives to Nginx’s `http` context, put them in a file called `http.conf` in the same directory as your Dockerfile. Do not surround them with `http { and }`.

- To add directives to Nginx’s `server` context, put them in a file called `server.conf` in the same directory as your Dockerfile. Do not surround them with `server { and }`.

**rewrite**

To redirect, say, `/surprise` (and `/surprise/`) to `https://youtu.be/dQw4w9WgXcQ`, create a file called `server.conf` in the same directory as your Dockerfile, the contents of which are as follows.

```
rewrite ^/surprise/?$ https://youtu.be/dQw4w9WgXcQ redirect;
```

To redirect one domain to another (e.g., `www.cs50.harvard.edu` to `cs50.harvard.edu`), create a file called `server.conf` in the same directory as your Dockerfile, the contents of which are as follows.

```
if ($http_host = www.cs50.harvard.edu) {
    rewrite (.*) https://cs50.harvard.edu$1;
}
```
try_files

To route all requests (that aren’t for actual files or directories) to public/index.php, create a file called server.conf in the same directory as your Dockerfile, the contents of which are as follows.

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

To route all requests (that aren’t for actual files or directories) to public/index.html (as you might for a JavaScript-based single-page app), create a file called server.conf in the same directory as your Dockerfile, the contents of which are as follows.

```
location / {
    try_files $uri $uri/ /index.html;
}
```

20.2.4 Port

By default, cs50/server uses TCP port 8080. To configure cs50/server to use some other port, adjust your Dockerfile as follows, where #### is your choice of ports:

```
FROM cs50/server
EXPOSE ####
...
CMD passenger start --port ####
```

20.2.5 Static Files

By default, cs50/server assumes that your app’s static files are in public, which is assumed to be in the same directory as your Dockerfile. To configure cs50/server to look in some other directory, configure your Dockerfile as follows, where path/to/directory is the relative path to that directory:

```
FROM cs50/server
...
CMD passenger start --static-files-dir path/to/directory
```

20.3 Notes

20.3.1 Caching

By default, HTTP responses from apps served by cs50/server are not cached by browsers (or proxies) because the image adds

```
Cache-Control: no-cache, no-store, must-revalidate
Expires: 0
Pragma: no-cache
```

to those responses.

To allow responses to be cached, create a file called server.conf in the app’s root containing the below, which will remove those headers:

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```
more_clear_headers 'Cache-Control' 'Expires' 'Pragma';

20.3.2 HTTPS

If `cs50/server` detects that it’s running behind a load balancer, whereby `X-Forwarded-Proto` (an HTTP header) is set, and the value of that header is `http` (the implication of which is that a client’s request used HTTP instead of HTTPS), `cs50/server` will redirect the request to use HTTPS.

20.3.3 Inline Frames

By default, apps based on `cs50/server` cannot be iframed in other sites, as the image adds

```bash
Content-Security-Policy: frame-ancestors 'self'
```

to HTTP responses. To allow an app to be iframed by another site, create a file called `server.conf` in the app’s root containing the below, which will remove that header:

```bash
more_clear_headers 'Content-Security-Policy';
```
CS50x FAQs

For answers to other questions, do just strike up a discussion with classmates!

21.1 Does CS50x have a test or quiz?

Nope! CS50 at Harvard University and Yale University does have a test and quiz, but CS50x does not.

21.2 Does CS50x have office hours?

No, afraid CS50x is too large, but you can still start a discussion with classmates!

21.3 Does CS50x have sections?

No, afraid CS50x is too large, but you can still start a discussion with classmates!

21.4 What’s the difference between “less comfortable” and “more comfortable” problems? Do I have to do both?

The “less comfortable” are what you might consider the “standard” version of the problem, designed for students who have little or no prior experience. The “more comfortable” are the “challenge” version, designed for students who consider themselves more comfortable due to prior study/experience before this class. As such, they may require more concepts than have been covered in the course so far.

You don’t get any extra points for doing the “more comfortable” problems. If you submit both, the one with the highest grade will count towards your overall pset grade.
21.5 Do I have to choose Verification before starting the course?

Nope! If you would like an edX Verified Certificate, you can pay for verification at any point, even after completing all of the coursework.

21.6 If I paid for a verified certificate before 2019, do I need to pay again?

No, you should be able to re-verify yourself via edX’s dashboard.

21.7 If I started CS50x before 2019, can I resume?

Yes, any scores you received on past years’ problem sets have been imported into CS50x 2019. However, moving forward, you must submit CS50x 2019’s problem sets.

21.8 Is my child too young to take CS50x?

CS50x is perhaps best suited for ages 12 and up. Younger students might need a hand from a parent.

21.9 How do I know if I have completed all requirements for a Verified Certificate?

cs50.me/cs50x shows the scores you have received for each submitted problem. Once you have completed all requirements, you will see a green banner atop that page, confirming your completion.

21.10 When are Verified Certificates generated?

At the end of each month, CS50’s staff provides edX with a list of students who have satisfied CS50x’s requirements (by receiving scores of at least 70% on every required problem submitted as part of the course’s nine problem sets as well as on the final project). Within a few days, edX then generates students’ Verified Certificates, at which point they should appear automatically on edX’s dashboard. Note that edX will not notify you, so be sure to check your dashboard. If it has been more than a month since you met all requirements and you have still not received your certificate, email certificates@cs50.harvard.edu with your details.

21.11 When are deadlines?

CS50x does not have deadlines for problem sets or for the final project. You are welcome to work on and submit at your own pace.
21.12  I registered for edX using my Facebook, Google, or Microsoft account, and can’t log in to submit Problem Set 0. What should I do?

Unfortunately, edX’s open ID (which we use for authentication) doesn’t support logging in with Facebook, Google, or Microsoft accounts yet. To get around this, for now, you should try resetting your password (even though you may have never set any before) from your Account Settings page on edX. Then try the pset0 form again, logging in with the email associated with your Facebook, Google, or Microsoft account you use to log into edX, and the password you just set.

21.13  My Style grade is not what I expected. Why might that be?

Your style grade will be based on the style50 results for every source code file in your submission. When you run submit50, be sure that your folder includes only those files that are necessary for the problem set. If you have any extra test files in there, those will be counted towards your grade. Remove the extra files from the folder and resubmit to get an updated Style score.

21.14  Why does edX think my grade 0%?

CS50x doesn’t use edX’s built-in gradebook, so all students’ scores appear (in edX’s gradebook) as 0%. Not to worry, see cs50.me/cs50x for your actual scores.
CS50 IDE and *cli50* (as well as CS50’s web apps) are implemented with containers, “a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.”

To run containers on your own Mac or PC, you just need to install a container platform like Docker Community Edition (CE), which is freely available for

- Linux
- macOS
- Windows

and other platforms.
Heroku is a “platform as a service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud.” Heroku even offers a free plan.

Here’s how to deploy your implementation of, say, C$50 Finance to Heroku from GitHub.

1. Watch Brian’s seminar to learn about git and GitHub, if not already familiar.
2. Sign up for free, private GitHub repositories at https://education.github.com/discount_requests/new, if you haven’t already.
3. Create a new private repository at https://github.com/new (called, e.g., finance).
4. Take note of the HTTPS URL of the repository (e.g., https://github.com/username/finance.
5. Change to your implementation’s directory on CS50 IDE.
   ```
cd ~/pset8/finance/
```
6. Create a git repo therein.
   ```
git init
```
7. Add the GitHub repository as a “remote,” where username is your own GitHub username.
   ```
git remote add origin https://github.com/username/finance.git
```
8. Push your code to GitHub.
   ```
git commit -am "first commit"
git push -u origin master
```
If you visit https://github.com/username/finance, where username is your own GitHub username, you should see your code in the repository.

9. Sign up for a free Heroku account at https://signup.heroku.com/, if you don’t have one already.

11. Configure your app at https://dashboard.heroku.com/apps/app-name/deploy/github, where app-name is your Heroku app's name.
   - Add this app to a pipeline: No need to configure; leave as is.
   - Deployment method: Select GitHub, then click Connect to GitHub. If prompted to log into GitHub, click Authorize heroku.
   - App connected to GitHub: Search for your app’s repository (e.g., username/finance, where username is your own GitHub username), then click Connect.
   - Automatic deploys: Click Enable Automatic Deploys.

12. Search for and provision Heroku Postgres at https://dashboard.heroku.com/apps/app-name/resources, where app-name is your Heroku app’s name; select a Plan name of Hobby Dev — Free.

13. At https://dashboard.heroku.com/apps/app-name/resources, where app-name is your Heroku app’s name, click Heroku Postgres :: Database. In the tab that opens, click Settings, then click View Credentials… Highlight and copy the URI that appears.

14. In CS50 IDE, open application.py in ~/pset8/finance/ and find:

   ```python
   db = SQL("sqlite:///finance.db")
   ```

   Replace sqlite:///finance.db with that URI (so that the CS50 Library will connect to your Postgres database instead of your SQLite database).

15. In CS50 IDE, execute the below to import finance.db into your Postgres database, where URI is that same URI. Be sure to append ?sslmode=require to the URI.

   ```bash
   pgloader finance.db URI?sslmode=require
   ```

   Thereafter, if you’d like to browse or edit your Postgres database, you can use Adminer (a tool like phpLiteAdmin for Postgres databases), at adminer.cs50.net. Log in using your database’s credentials: at https://dashboard.heroku.com/apps/app-name/resources, where app-name is your Heroku app’s name, click Heroku Postgres :: Database. In the tab that opens, click Settings, then click View Credentials…

16. Create a new file in CS50 IDE called Procfile in ~/pset8/finance/ whose contents are:

   ```text
   web: gunicorn application:app
   ```

   That file will tell Heroku to look in a file called application.py for a variable called app and serve it with Gunicorn, a production-quality web server. (Flask’s built-in web server is “good enough for testing but probably not what you want to use in production.”)

17. Add that file to your repository and push it to GitHub.

   ```bash
   git add -am "added Procfile"
   git push
   ```

   If you visit https://app-name.herokuapp.com/, where app-name is your Heroku app’s name, you should see your app! If you instead see some error, visit https://dashboard.heroku.com/apps/app-name/logs, where app-name is your app’s name, to diagnose! Each time you add (new or changed) files to your repository and push to GitHub hereafter, your app will be re-deployed to Heroku.
Most of CS50’s command-line tools are implemented in Python and can be installed on your own Mac or PC via `pip`, a command-line tool via which you can install Python packages from PyPI, “the default package index for the Python community.”

If you already have `python` installed, odds are you also have `pip`. If not, you can install it for

- Linux
- macOS
- Windows

and other platforms.
Most of CS50’s command-line tools require that you have Python 3.6 or higher installed, since CS50 uses f-strings and assumes that keyword arguments are ordered. (If you already have Python installed on your Mac or PC but didn’t install it yourself, odds are it’s an older version.) You can install the latest version of Python for

- Linux, via `pyenv`,
- macOS, via installer or `pyenv`,
- Windows,

and other platforms.
TODO

If you’d like to install `render50` on your own Mac or PC, so that you can render PDFs without using CS50 IDE, you’ll need a command-line environment:

- If running Linux or the like, you already have one! Open a terminal window in your usual way.
- If running Mac OS, you already have one! Open Applications > Utilities > Terminal.
- If running Windows, you’ll need to install the Windows Subsystem for Linux, which is only supported on Windows 10. Once installed, run `bash`. 
Teacher Sites

This is where all the resources for your CS50 classroom will live for your classroom. With this model, you have complete control over what content is displayed as well as how it is displayed.

27.1 Getting Started

1. Sign up or login to your GitHub account and visit https://github.com/cs50/ap
2. Click the Fork button in the top right corner of the page. This will create your own copy of our teacher site, which will live at https://github.com/USERNAME/ap, where USERNAME is your own GitHub username.
3. Go to https://github.com/USERNAME/ap/settings, where USERNAME is your own username. Scroll down to the GitHub Pages section. Under source, select “Master Branch” and click save. After you make a change (or a push), your course site should be live at https://USERNAME.github.io/ap

27.2 Structure and Organization of Repository

Most files you will be interacting with, particularly if you are less comfortable, are markdown (.md) files. Markdown is a lightweight markup language that allows you to format text that will later be converted to HTML when your site is built.

27.2.1 _includes/

Your _includes folder is where you will keep files that will be used on more than one page. For example, nav.md is included here because we want the navigation bar to appear on every page.

- header.md: This is where you will set the title and subtitle for your own site.
- nav.md: This is where you will add menu items to your sidebar.
27.2.2 assets

This folder contains static files, such as PDFs, images, etc.

- pdfs/: This folder contains all the PDFs used in the CS50 AP curriculum. You do not need to edit any of these files. If interested, all of the reference sheets live here.
- css/: This folder is where you can add custom css to your site.

27.2.3 periods/

If you have multiple CS50 classes running at different paces, this is where each individual period’s content goes. One method for displaying content is to include all of the units and just comment out what you do not want your students to view like in 1.md or 5.md. Be sure to link these in _includes/nav.md so that they appear in your nav bar. HTML comments, <!-- like this -->, work in .md just make sure you have a blank line between the end of your displayed text and the start of your comment. Otherwise Markdown will print out the plaintext <!-- ... --> on your website.

27.2.4 curriculum/

The curriculum directory contains a directory of each chapter. Within each chapter directory are a md files where are the resources are listed and a notes folder where the images and text for the notes pages live.

27.2.5 Gemfile and _config.yml

These are files that you do not need to edit. Our teacher sites use a tool called Jekyll, which uses _config.yml to configure the site. Gemfile lists the Gems a Ruby project needs to run. For Jekyll specifically, they are used to for Jekyll plugins.
27.2.6 index.md

This is the markdown file for the homepage of your site, https://USERNAME.github.io/ap. You can use this page for most anything including but limited to: course-wide announcements, Twitter feeds, embedded Google Calendars, links to school specific resources etc.

27.2.7 tools.md

This page has a pdf of the workflow for how students typically use CS50 tools. This page also includes usage of CS50-specific command-line tools. If you have created your own tools or would like to share tools you have found to be helpful for your students you can feel free to post them here.

27.3 Content Updates

You are more than welcome to change the files in this repository in whatever way you see fit. We’ll update our every year with the new resources but just be aware that while you can use older versions of the course, some websites and tools may be updated that are not necessarily reflected in older resources.

If you are among those less comfy and you will want to download your current version of your fork to your machine. You can do this by visiting https://github.com/USERNAME/ap, where USERNAME is your own GitHub username and clicking on the green clone or download and then select download ZIP. This will allow you to keep any changes you’ve made on your previous version of the site. Then visit https://github.com/USERNAME/ap/settings, where USERNAME is your own GitHub username, and scroll down to the bottom and select Delete this repository. Only do this after you are sure you’ve downloaded a copy of it locally, lest you lose any customizations you’ve made.

Then go ahead and visit github.com/cs50/ap and fork the repo per the instructions in the Getting Started Section.

If you are among those more comfy and you already have a fork of the repo you are certainly welcome to sync your fork with ours.

27.4 Custom Domains

If you would like to serve your site from a different domain, you can visit https://github.com/USERNAME/ap/settings, where USERNAME is your own GitHub username. Scroll down to the GitHub Pages section and specify your custom domain, i.e ap.cs50.school. This will create a file in your ap repository called CNAME that specifies where this site will be served.

27.5 Editing Files

27.5.1 Less Comfy

If you are among those less comfy, you can do most everything from GitHub’s web UI. For the most part, you will be creating Markdown files, but keep in mind that you can use raw HTML directly into your .md files. For example if you wanted to include an iframe for a youtube video. You could literally type <iframe src="https://www.youtube.com/embed/..."></iframe> and it will create the iframe on your page.

To create a new file, simply click create new file and you can specify the file’s path and type the contents of that file directly into the text editor provided. You could also type the contents in some other text editor and upload the file directly with the upload files button.
You can also edit existing files. Once you’ve selected the file you’d like to edit, click on the pencil icon at the top right of the page. You’ll be able to make changes in the text editor provided.

Once you’re satisfied with your changes, you’ll want to commit them. Commit is GitHub lingo for saving a file. The top text box is the message for your commit. It could be something like “adds link to period4.md.” These messages are solely for you so don’t worry about being super descriptive. If you’d like to add further text here you can use the second box for a longer explanation but it is optional. Make sure that you have selected “Commit directly to the master branch.” Then click “Commit changes.” Within a few minutes or so, your site should reflect these changes.

27.5.2 More Comfy

If you’d like to develop your site and run it locally, before committing your changes, you’ll need to do a little setup first.

First you’ll need to check if you have ruby as with `ruby -v` in your terminal window. If you do not have ruby, you can find installation instructions here.

After you’ve successfully installed Ruby, you’ll need to install Jekyll, the static site generator we use for our teacher sites and Bundler, which provides a consistent environment for Ruby projects by tracking and installing the exact gems and versions that are needed. You can install both of these with `gem install jekyll bundler`. Once everything is installed and you’ve cloned your GitHub repo to your machine, you’ll be able to build and view your site locally.

First you’ll need to run `bundle install`.

This will install all the dependencies specified in your Gemfile. You will only need to do this once, unless you modify your Gemfile.

Then to build and serve up your site you’ll run `bundle exec jekyll serve`.

You can view your site at `http://localhost:4000`. Your page will reflect changes you’ve made to files in real time after saving the file and refreshing the page (it may take a minute or so). From there, you `git add`, `git commit`, and `git push` as normal.
27.6 Additional Documentation
CHAPTER 28

CS50 Theme

TODO
29.1 Installation

29.1.1 Ubuntu

```bash
$ curl -s https://packagecloud.io/install/repositories/cs50/repo/script.deb.sh | sudo bash
$ sudo apt install libcs50
```

29.1.2 Fedora

```bash
$ curl -s https://packagecloud.io/install/repositories/cs50/repo/script.rpm.sh | sudo bash
$ dnf install libcs50
```

29.1.3 From Source (Linux and Mac)

1. Download the latest release from [https://github.com/cs50/libcs50/releases](https://github.com/cs50/libcs50/releases)
2. Extract `libcs50-*.*`
3. `cd libcs50-*`
4. `sudo make install`

29.1.4 Environment Variables

For parity with CS50 CLI, CS50 Sandbox, CS50 Lab, and CS50 IDE, you may want to set these environment variables:
29.1.5 Troubleshooting

If when compiling your program, you see:

```
/usr/bin/ld: cannot find -lcs50: Add export LIBRARY_PATH=/usr/local/lib to your .bashrc.
fatal error: 'cs50.h' file not found: Add export C_INCLUDE_PATH=/usr/local/include to your .bashrc.
cannot open shared object file: No such file or directory: Add export LD_LIBRARY_PATH=/usr/local/lib to your .bashrc.
```

29.2 Usage

Note: To use these functions, make sure to include #include <cs50.h> atop your file and compile with the -lcs50 flag.

**string**

Type representing a C string. Aliased to char *.

Example usage:

```
string s = "hello, world!";
```

char **get_char** (const char *format, ...)

Parameters

- *format* – the printf() -like format string used to display the prompt
- *...* – values to be substituted into the format string a la printf()

Returns the char equivalent to the line read from stdin, or CHAR_MAX on error

Prompts user for a line of text from standard input and returns the equivalent char; if text does not represent a single char, user is reprompted.

Example usage:

```
#include <stdio.h>
#include <cs50.h>

int main(void)
{
    // attempt to read character from stdin
    char c = get_char("Enter char: ");

    // ensure character was read successfully
```
double get_double (const char *format, ...)

Parameters

- format – the printf()-like format string used to display the prompt
- ... – values to be substituted into the format string a la printf()

Returns the double equivalent to the line read from stdin in [DBL_MIN, DBL_MAX), as precisely as possible, or DBL_MAX on error.

Prompts user for a line of text from standard input and returns the equivalent double; if text does not represent a double or would cause overflow or underflow, user is reprompted.

Example usage:

double divide_doubles (void)
{
  // read double from stdin
  double d = get_double("Enter a double: ");

  // make sure we read one successfully
  if (d == DBL_MAX)
  {
    return DBL_MAX;
  }

  double e = get_double("What do you want to divide %lf by? ", d);

  // make sure we don't divide by zero
  if (e == DBL_MAX || e == 0.0)
  {
    return DBL_MAX;
  }

  return i / j;
}

int get_int (const char *format, ...)

Parameters

- format – the printf()-like format string used to display the prompt
- ... – values to be substituted into the format string a la printf()
Returns the int equivalent to the line read from stdin in [INT_MIN, INT_MAX] or INT_MAX on error.

Prompts user for a line of text from standard input and returns the equivalent int; if text does not represent an int or would cause overflow, user is reprompted.

Example usage:

```c
#include <cs50.h>
...

// Returns the sum of two ints read from stdin, or INT_MAX if there was an error.
int add_ints(void)
{
    // read int from stdin
    int i = get_int("Enter an int: ");

    // make sure we read one successfully
    if (i == INT_MAX)
    {
        return INT_MAX;
    }

    int j = get_int("What do you want to add %d to? ", i);

    if (j == INT_MAX)
    {
        return INT_MAX;
    }

    return i + j;
}
```

float get_float (const char *format, ...)

Parameters

- **format** – the printf()-like format string used to display the prompt
- **...** – values to be substituted into the format string a la printf()

Returns the float equivalent to the line read from stdin in [FLT_MIN, FLT_MAX), as precisely as possible, or FLT_MAX on error.

Prompts user for a line of text from standard input and returns the equivalent float; if text does not represent a float or would cause overflow or underflow, user is reprompted.

Example usage:

```c
// Returns the product of two floats, or FLT_MAX on error.
float multiply_floats(void)
{
    // read float from stdin
    float f = get_float("Enter a float: ");

    // make sure we read one successfully
    if (f == FLT_MAX)
    {
        return FLT_MAX;
    }
```

(continues on next page)
long \textbf{get\_long}(\text{const char *format, ...})

\textbf{Parameters}

\begin{itemize}
  \item \texttt{format} – the \texttt{printf()}-like format string used to display the prompt
  \item \ldots – values to be substituted into the format string a la \texttt{printf()}
\end{itemize}

\textbf{Returns} the \texttt{long} equivalent to the line read from stdin in \texttt{[LONG\_MIN, LONG\_MAX]} or \texttt{LONG\_MAX} on error

Prompts user for a line of text from standard input and returns the equivalent \texttt{long}; if text does not represent an \texttt{int} or would cause overflow, user is reprompted.

Example usage:

```
#include <cs50.h>
...

// Returns the difference of two longs read from stdin, or LONG\_MAX if there was an error.
long subtract\_longs(\text{void})
{
  // read long from stdin
  long i = get\_long("Enter a long: ");

  // make sure we read one successfully
  if (i == LONG\_MAX)
  {
    return LONG\_MAX;
  }

  long j = get\_long("What do you want to subtract from %ld? ", i);

  if (j == LONG\_MAX)
  {
    return LONG\_MAX;
  }

  return i - j;
}
```

char *\textbf{get\_string}(\text{const char *format, ...})

\textbf{Parameters}

\begin{itemize}
  \item \texttt{format} the \texttt{printf()}-like format string used to display the prompt
  \item \ldots – values to be substituted into the format string a la \texttt{printf()}
\end{itemize}
returns the read line as a string sans line endings, or NULL on EOF.

Prompts user for a line of text from standard input and returns it as a string (char *), sans trailing line ending. Supports CR (\r), LF (\n), and CRLF (\r\n) as line endings. Stores string on heap, but library’s destructor frees memory on program’s exit.

Example usage:

```c
int main(void)
{
    string s = get_string("Enter string: ");

    // ensure string was read
    if (s == NULL)
    {
        return 1;
    }

    string next = get_string("You just entered %s. Enter a new string: ", s);

    if (next == NULL)
    {
        return 1;
    }

    printf("Your last string was %s\n", next);
}
```
CHAPTER 30

CS50 Library for C++

TODO
CHAPTER 31

CS50 Library for Java

TODO
There’s no one, right way to stylize code. But there are definitely a lot of wrong (or, at least, bad ways). Even so, CS50 does ask that you adhere to the conventions below so that we can reliably analyze your code’s style. Similarly do companies typically adopt their own, company-wide conventions for style.

### 33.1 Line Length

By convention the maximum length of a line of code is 80 characters long in C, with that being historically grounded in standard-sized monitors on older computer terminals, which could display 24 lines vertically and 80 characters horizontally. Though modern technology has obsoleted the need to keep lines capped at 80 characters, it is still a guideline that should be considered a “soft stop,” and a line of 100 characters should really be the longest you write in C, else readers will generally need to scroll. If you need more than 100 characters, it may be time to rethink either your variable names or your overall design!

```c
// These next lines of code first prompt the user to give two integer values and then multiplies those two integer values together so they can be used later in the program
int first_collected_integer_value_from_user = get_int("Integer please: ");
int second_collected_integer_value_from_user = get_int("Another integer please: ");
int product_of_the_two_integer_values_from_user = first_collected_integer_value_from_user * second_collected_integer_value_from_user;
```

In other languages, particularly JavaScript, it is significantly more difficult to constrain lines to a maximum length; there, your goal should instead be to break up lines (as via \n) in locations that maximize readability and clarity.

### 33.2 Comments

Comments make code more readable, not only for others (e.g., your TF) but also for you, especially when hours, days, weeks, months, or years pass between writing and reading your own code. Commenting too little is bad. Commenting too much is bad. Where’s the sweet spot? Commenting every few lines of code (i.e., interesting blocks) is a decent guideline. Try to write comments that address one or both of these questions:
1. What does this block do?
2. Why did I implement this block in this way?

Within functions, use “inline comments” and keep them short (e.g., one line), else it becomes difficult to distinguish comments from code, even with syntax highlighting. Place the comment above the line(s) to which it applies. No need to write in full sentences, but do capitalize the comment’s first word (unless it’s the name of a function, variable, or the like), and do leave one space between the // and your comment’s first character, as in:

```
// Convert Fahrenheit to Celsius
float c = 5.0 / 9.0 * (f - 32.0);
```

In other words, don’t do this:

```
//Convert Fahrenheit to Celsius
float c = 5.0 / 9.0 * (f - 32.0);
```

Or this:

```
// convert Fahrenheit to Celsius
float c = 5.0 / 9.0 * (f - 32.0);
```

Or this:

```
float c = 5.0 / 9.0 * (f - 32.0); // Convert Fahrenheit to Celsius
```

Atop your .c and .h files should be a comment that summarize what your program (or that particular file) does, as in:

```
// Says hello to the world
```

Atop each of your functions (except, perhaps, main), meanwhile, should be a comment that summarizes what your function is doing, as in:

```
// Returns the square of n
int square(int n)
{
    return n * n;
}
```

### 33.3 Conditions

Conditions should be styled as follows:

```
if (x > 0)
{
    printf("x is positive\n");
}
else if (x < 0)
{
    printf("x is negative\n");
}
else
{
    printf("x is zero\n");
}
```
Notice how:

- the curly braces line up nicely, each on its own line, making perfectly clear what’s inside the branch;
- there’s a single space after each if;
- each call to printf is indented with 4 spaces;
- there are single spaces around the > and around the <; and
- there isn’t any space immediately after each ( or immediately before each ).

To save space, some programmers like to keep the first curly brace on the same line as the condition itself, but we don’t recommend, as it’s harder to read, so don’t do this:

```c
if (x < 0) {
    printf("x is negative\n");
} else if (x < 0) {
    printf("x is negative\n");
}
```

And definitely don’t do this:

```c
if (x < 0)
{
    printf("x is negative\n");
}
else
{
    printf("x is negative\n");
}
```

33.4 Switches

Declare a switch as follows:

```c
switch (n) {
    case -1:
        printf("n is -1\n");
        break;
    case 1:
        printf("n is 1\n");
        break;
    default:
        printf("n is neither -1 nor 1\n");
        break;
}
```

Notice how:

- each curly brace is on its own line;
- there’s a single space after switch;
- there isn’t any space immediately after each ( or immediately before each );
- the switch’s cases are indented with 4 spaces;
the cases’ bodies are indented further with 4 spaces; and
  • each case (including default) ends with a break.

## 33.5 Functions

In accordance with C99, be sure to declare `main` with:

```c
int main(void)
{
  
}
```

or, if using the CS50 Library, with:

```c
#include <cs50.h>

int main(int argc, string argv[])
{
  
}
```

or with:

```c
int main(int argc, char *argv[])
{
  
}
```

or even with:

```c
int main(int argc, char **argv)
{
  
}
```

Do not declare `main` with:

```c
int main()
{
  
}
```

or with:

```c
void main()
{
  
}
```

or with:

```c
main()
{
  
}
```
As for your own functions, be sure to define them similarly, with each curly brace on its own line and with the return type on the same line as the function’s name, just as we’ve done with main.

### 33.6 Indentation

Indent your code four spaces at a time to make clear which blocks of code are inside of others. If you use your keyboard’s Tab key to do so, be sure that your text editor’s configured to convert tabs (\t) to four spaces, else your code may not print or display properly on someone else’s computer, since \t renders differently in different editors. (If using CS50 IDE, it’s fine to use Tab for indentation, rather than hitting your keyboard’s space bar repeatedly, since we’ve preconfigured it to convert \t to four spaces.)

Here’s some nicely indented code:

```c
// Print command-line arguments one per line
printf("\n");
for (int i = 0; i < argc; i++)
{
    for (int j = 0, n = strlen(argv[i]); j < n; j++)
    {
        printf("%c\n", argv[i][j]);
    }
    printf("\n");
}
```

### 33.7 Loops

#### 33.7.1 for

Whenever you need temporary variables for iteration, use i, then j, then k, unless more specific names would make your code more readable:

```c
for (int i = 0; i < LIMIT; i++)
{
    for (int j = 0; j < LIMIT; j++)
    {
        for (int k = 0; k < LIMIT; k++)
        {
            // Do something
        }
    }
}
```

If you need more than three variables for iteration, it might be time to rethink your design!

#### 33.7.2 while

Declare while loops as follows:

```c
while (condition)
{
    // Do something
}
```
Notice how:

- each curly brace is on its own line;
- there’s a single space after `while`;
- there isn’t any space immediately after the `( or immediately before the `)`; and
- the loop’s body (a comment in this case) is indented with 4 spaces.

### 33.7.3 do ... while

Declare `do ... while` loops as follows:

```c
do
{
    // Do something
}
while (condition);
```

Notice how:

- each curly brace is on its own line;
- there’s a single space after `while`;
- there isn’t any space immediately after the `( or immediately before the `)`; and
- the loop’s body (a comment in this case) is indented with 4 spaces.

### 33.8 Pointers

When declaring a pointer, write the `*` next to the variable, as in:

```c
int *p;
```

Don’t write it next to the type, as in:

```c
int* p;
```

### 33.9 Variables

Because CS50 uses C99, do not define all of your variables at the very top of your functions but, rather, when and where you actually need them. Moreover, scope your variables as tightly as possible. For instance, if `i` is only needed for the sake of a loop, declare `i` within the loop itself:

```c
for (int i = 0; i < LIMIT; i++)
{
    printf("%i\n", i);
}
```

Though it’s fine to use variables like `i`, `j`, and `k` for iteration, most of your variables should be more specifically named. If you’re summing some values, for instance, call your variable `sum`. If your variable’s name warrants two words (e.g., `is_ready`), put an underscore between them, a convention popular in C though less so in other languages.

If declaring multiple variables of the same type at once, it’s fine to declare them together, as in:
Just don’t initialize some but not others, as in:

```c
int quarters, dimes = 0, nickels = 0 , pennies;
```

Also take care to declare pointers separately from non-pointers, as in:

```c
int *p;
int n;
```

Don’t declare pointers on the same line as non-pointers, as in:

```c
int *p, n;
```

### 33.10 Structures

Declare a `struct` as a type as follows, with each curly brace on its own line and members indented therein, with the type’s name also on its own line:

```c
typedef struct
{
    string name;
    string dorm;
}
student;
```

If the `struct` contains as a member a pointer to another such `struct`, declare the `struct` as having a name identical to the type, without using underscores:

```c
typedef struct node
{
    int n;
    struct node *next;
}
node;
```
CS50 Forms is a web app at forms.cs50.io that allows you to:

- add authentication to Google Forms, requiring that respondents authenticate via edX, HarvardKey, or Yale NetID, thereby allowing you to post a form’s URL online while still restricting access
- pre-fill Google Forms with a respondent’s name and email address as well as a respondent’s edX username, HUID, or NetID, thereby decreasing the probability of typographical errors

Upon authenticating, a respondent will be redirected to a “pre-filled link” like

```
https://docs.google.com/forms/d/e/1FAIpQLSearFUWMKeKJo5225PoeCeJXHktJecyStal_˓
    sn6nYEb0rOEgYw/viewform?entry.1142694446=username%40example.com&entry.˓
    164044587=Full+Name
```

wherein username%40example.com (i.e., username@example.com) represents the respondent’s email address and Full Name represents the respondent’s full name, values that Google will use to pre-fill the form.

### 34.1 Notes

- Respondents can still override (i.e., manually change) any pre-filled value. In our experience, respondents tend not to, unless explicitly asked to, e.g., adjust their full name to reflect a preferred nickname.
- After authenticating, a respondent could theoretically share a form’s URL, pre-filled or otherwise, with anyone on the internet (as via email or any other mechanism), thereby allowing others to access the form without authentication. In our experience, respondents tend not to, if only because there isn’t much demand for unauthenticated access to the forms in question, which are usually assignments! CS50 Forms simply raises a bar thereto.
35.1 CS50 IDE FAQs

35.1.1 How can I open a file in the editor from a terminal tab?

If you want to open a file named foo, run `open path/to/foo`.

35.1.2 I closed the terminal under my editor by mistake. How do I get it back?

If the Console pane at the bottom is visible, you should be able to open a new terminal tab by clicking the plus button atop that pane, and choosing `New Terminal`.
TIP: If you no longer see the Console pane, you can bring it back via View > Console or by hitting F6. You can also open a terminal in any other pane.

### 35.1.3 Can I use a different editor?

Sure, if you are more familiar, you can run editors in the terminal such as `emacs`, `nano`, or `vim`.

### 35.1.4 What if I already have a AWS account?

The CS50 IDE isn’t available yet to all AWS accounts.

### 35.1.5 I’m getting an error that says "Looks like check50 (or submit50) isn’t enabled for your account yet. Log into https://cs50.me/ in a browser, click Authorize Application, and re-run check50 (or submit50) here! even after following these instructions. What should I do?"

Go to https://github.com/check50/<your-username>/invitations and https://github.com/submit50/<your-username>/invitations, after replacing `<your-username>` with your actual GitHub username, and click Accept invitation, then try check50 and/or submit50 again. Let sysadmins@cs50.harvard.edu know if you need further assistance!

If curious why this happens, earlier the Authorize Application step was sufficient for adding your GitHub user as a collaborator with write access on your check50 and submit50 repositories that we automatically create for you, but GitHub updated their API recently to require users to accept invitations for repositories they’re added to, so the Authorize Application step isn’t sufficient any more, and you have to accept these invitations.
35.1.6 How to restore files that I have accidentally deleted?

Please follow the instructions at How do I recover a deleted file?. Let sysadmins@cs50.harvard.edu know if you need further assistance!

35.2 CS50 IDE Offline

Normally, CS50 IDE requires a constant connection to the internet. This may not fit users with slow or unstable internet connections, or those who want to continue using the IDE offline for any number of reasons. The following instructions walk you through how to download and install the offline version of CS50 IDE. Please note that the new offline IDE may be quite a large download (several GB).

35.2.1 Installing

CS50 IDE Offline is a containerized app. We need to install a platform called Docker to run it.

Linux

Follow the instructions for your Linux distribution to install Docker Engine.

1. Open up a terminal window and run the following command to create a new CS50 IDE instance:

   ```
   docker run --privileged -e "IP=127.0.0.1" -e "PORT=8080" --name ide50 -d -p 5050:5050 -p 8080-8082:8080-8082 cs50/ide
   ```

2. Visit http://localhost:5050/ in your favorite browser to access CS50 IDE.

Mac

NOTE: If these system requirements are not met, skip to Docker Toolbox.

1. Install Docker for Mac.

2. Open up a terminal window and run the following command to create a new CS50 IDE instance:

   ```
   docker run --privileged -e "IP=127.0.0.1" -e "PORT=8080" --name ide50 -d -p 5050:5050 -p 8080-8082:8080-8082 cs50/ide
   ```

3. Visit http://localhost:5050/ in your favorite browser to access CS50 IDE.

Windows

NOTE: If these system requirements are not met, skip to Docker Toolbox.

1. Install Docker for Windows.

2. Open up a command prompt window and run the following command to create a new CS50 IDE instance:

   ```
   docker run --privileged -e "IP=127.0.0.1" -e "PORT=8080" --name ide50 -d -p 5050:5050 -p 8080-8082:8080-8082 cs50/ide
   ```

3. Visit http://localhost:5050/ in your favorite browser to access CS50 IDE.
**Docker Toolbox**

NOTE: You should only follow these instructions if you are a Windows or Mac user and the system requirements for Windows or Mac, respectively, are not met.

1. Download and install Docker Toolbox for Windows or Mac.

2. Open up the Docker QuickStart Terminal app that was installed on your computer. After a while, you should see something like the following, with a command prompt below it:

   ![Docker QuickStart Terminal](image)

   ### docker is configured to use the default machine with IP 192.168.99.101
   For help getting started, check out the docs at https://docs.docker.com

3. The IP stated in the screenshot above is the IP of the Docker machine. It may be different on your computer. You are going to use this IP to access your CS50 IDE, so you should record it.

4. Run the following command in the Docker QuickStart Terminal to create a new CS50 IDE instance:

   ```bash
   docker run --privileged -e "IP=$(docker-machine ip default)" -e "PORT=8080" --
   -> name ide50 -d -p 5050:5050 -p 8080-8082:8080-8082 cs50/ide
   ```

5. Visit <http://:5050/>, in your favorite browser, to access CS50 IDE, after replacing <machine-ip> with the actual IP for the Docker machine.

TIP: If you ever forgot or lost the IP of the Docker machine, you should be able to get it back by running docker-machine ip default in the Docker QuickStart Terminal.

### 35.2.2 Downloading the Docker Image

When you execute the `docker run` step above, the latest IDE base image should be automatically downloaded for you. If you have a very slow or unstable internet connection, the download might fail. The following steps describe how you can download the Docker image separately and load it into your Docker setup:

1. Download the image (or torrent-download if you prefer).

2. Open up a terminal window or Docker QuickStart Terminal and run

   ```bash
   docker load < path/to/ide.tar
   ```

3. Try running the `docker run` step from the set of instructions that you followed above again.

### 35.2.3 Managing the IDE
Starting the IDE

If you can’t access your CS50 IDE (e.g., after restarting your computer), it’s possibly because it’s stopped. To start it back, open up a terminal window (if using Mac), a command prompt (if using Windows), or Docker QuickStart Terminal (if using Docker Toolbox), and run the following command:

```
docker start ide50
```

Restarting

Normally, you shouldn’t need to restart the IDE, but if you ever wanted to open up a terminal window (if using Mac), a command prompt (if using Windows) or Docker QuickStart Terminal (if using Docker Toolbox), and run the following command:

```
docker restart ide50
```

Stopping

To stop CS50 IDE, open up a terminal window (if using Mac), a command prompt (if using Windows) or Docker QuickStart Terminal (if using Docker Toolbox), and run the following command:

```
docker stop ide50
```

Removing

To remove your CS50 IDE instance, open up a terminal window (if using Mac), a command prompt (if using Windows) or Docker QuickStart Terminal (if using Docker Toolbox), and run the following command:

```
docker rm ide50
```

35.2.4 Where to Go Next?

Read up on how to use CS50 IDE itself!

35.3 CS50 IDE

35.3.1 Introduction

CS50 IDE is a cloud-based Integrated Development Environment powered by AWS Cloud9 that features a cloud-based Ubuntu environment, a browser-based editor that supports syntax highlighting and word completion, a GUI-based GDB debugging, themes, customizable layouts, keyboard shortcuts, and many more features. Since it’s cloud-based, you can continue working on your problem sets even if you use a different computer!

35.3.2 Getting Started

1. If you don’t already have one, create a GitHub account [here](#).
2. Visit ide.cs50.io.
3. Click Sign in with GitHub then login into your GitHub account and authorize the CS50 IDE GitHub app if prompted.

4. Once you log in, you will automatically be forwarded to CS50 IDE! Hereafter, you may simply return to ide.cs50.io to log in and return to CS50 IDE, where all your files and settings are preserved.

### 35.3.3 Working with Files

#### Creating Files

There are multiple ways to create a new file in CS50 IDE:

- Click **File > New File**.
- Click on the little button atop any of the open panes and choose **New File** to open a blank file in that particular pane.
- From the file browser on the left, right-click or control-click on a directory and choose **New File** from the menu to create a blank file inside that directory, then double-click that file to open it.
- Press Alt + N (on a PC) or + N (on a Mac).

#### Saving Files

When a file is open in a tab and you have some unsaved changes, CS50 IDE will show a red dot atop that tab, until you save your changes. Probably the easiest way to save a file is to press Ctrl + S (on a PC) or + S (on a Mac), but you can also achieve the same by clicking **File > Save** (or **File > Save As...** if you want to save that as a new file), while you’re working on that file.

![Code Editor](https://via.placeholder.com/150)

#### Downloading Files

To download a file from your workspace to your local computer, simply navigate to the location of that file, in the file browser on the left, right-click on that file’s name, and choose **Download**.

To download all files in your home folder (i.e., ~/), click **File > Download Project**.
Uploading Files

To upload a file from your local computer to your workspace:

1. Select a directory where you want your files to get uploaded into, by clicking on that directory in the file browser on the left. By default, this is going to be your ~/ directory.
2. Click File > Upload Local Files..., then choose either Select files or Select folder, depending on what you want to upload.

File Revision History

While working on a file, you can easily undo changes by clicking Edit > Undo or by pressing Ctrl + Z (on a PC) or + Z on your keyboard. Similarly, you can redo changes by clicking Edit > Redo or by pressing Ctrl + Shift + Z.

The CS50 IDE also keeps track of file revisions, in case you want to toggle between file revisions, without having to undo or redo many times. You can show the whole file revision history by clicking File > Show File Revision History, which will show a timeline similar to the following, on which you can click to jump to a particular version.

35.3.4 Working with Terminals

Terminals allow you to interact with the underlying Ubuntu environment of CS50 IDE, using textual commands, to do all sorts of things, such as creating, copying, or moving files, compiling and running your programs, and more.

Opening New Terminals

When CS50 IDE first starts, there should be a terminal tab open at the bottom by default. You can also open a new terminal tab in that or any other pane of your choice by clicking the button atop that pane, and choosing New Terminal. Alternatively, you may just hit Alt + T (on a PC) or Option + T (on a Mac).

By default, the current working directory (CWD) in a new terminal is your ~/ directory. You can always navigate to your desired directory using cd path/to/directory. To open a terminal in a different directory, navigate to that directory in your file browser, right-click (on a PC) or Ctrl-click (on a Mac) on the directory’s name, and choose Open Terminal Here.

Copying and Pasting

You will probably need to copy and paste commands into terminal tabs to run them. By default, copying and pasting via menus will work inside CS50 IDE, but you may have to grant the IDE permission to see the contents of your clipboard first. You can either choose to grant the IDE that permission or use your keyboard to copy and paste instead by pressing Ctrl + C and Ctrl + V> (on a PC) or + C and + V (on a Mac).

Command History

You will be often using the same commands over and over. Whether you don’t remember a particular command, or too lazy to type it again, you can leverage the command history that is kept by your terminals. You can scroll up and down through the list of commands by pressing your keyboard’s up or down arrow.
Additionally, you can search for a particular command by pressing Ctrl + R (on a PC) or + R (on a Mac), then pressing the same key combination again to scroll through the matches, and finally Tab to select a particular match to modify it before running or Enter if you want to run it directly.

**Clearing Terminals**

From time to time you will need to clear your terminal so that it’s easier to see what you’re doing. There are two main ways to do that

1. Press Ctrl + L (on a PC) or + L (on a Mac). This doesn’t actually clear the terminal, but rather just scrolls down, so you can always scroll back up and see what got cleared if you want to.

2. Press Ctrl + K (on a PC) or + K (on a Mac). This actually clears the terminal; you won’t be able to scroll back up and see what got cleared.

**Troubleshooting**

If you want to force a program to quit, for example because it’s stuck in an infinite loop, press Ctrl + C (on a PC) or + C (on a Mac). It may take several seconds for the program to respond.

As a last resort, in case the program won’t stop, you might need to forcibly kill it. Perhaps the easiest way to do that is to just close the terminal tab, clicking *Close* when prompted, and opening a new one.

### 35.3.5 Layouts and Themes

**Layouts**

The CS50 IDE is very customizable when it comes to laying out panes and tabs. You could very easily split a pane horizontally or vertically, by right-clicking (on a PC) or Ctrl-clicking (on a Mac) somewhere next to the button atop the pane you want to split and choosing *Split Pane in Two Rows* or *Split Pane in Two Columns*. 
You could also move tabs between different panes by dragging and dropping a tab to the targeted pane or even to somewhere you want a new pane with that tab to be created.

**Themes**

By default a theme called **Cloud9 Day** is selected, but if you prefer a dark theme, you can select it via **View > Dark Mode**.
Presentation Mode

CS50 IDE also provides **Presentation Mode** in which the user interface is even more simplified and font sizes are larger. You can toggle that mode via **View > Presentation Mode**.

### 35.3.6 Sharing Your Workspace

**Adding a Member to Your Workspace**

1. Head to your IDE and click **Share** on the top-right corner.
2. Under **Invite members**, type in the GitHub username of the user you’d like to share your IDE with and click **Invite**.

   ![CS50 IDE](image)

   **CS50 IDE**
   
   integrated development environment for students and teachers
   
   **Invite members**
   
   [student50]

   **Who has access**

3. Copy your IDE’s link and share it with that user. They should now have access to your IDE.

   ![CS50 IDE](image)

   **CS50 IDE**
   
   integrated development environment for students and teachers
   
   **Invite members**
   
   [Username]

   **Who has access**

   [student50] [x]
   
   [jharvard: Owner]
Removing a Member to Your Workspace

1. Head to your IDE and click Share on the top-right corner.

2. Under **Who has access** find the GitHub username of the user you’d like to remove from your IDE.

3. Click the x button to the right of that username. They should no longer have access to your IDE.
35.3.7 Reporting Problems

If having any problems with CS50 IDE, please contact sysadmins@cs50.harvard.edu with all the necessary information about the problem, and how to replicate it, attaching screenshot(s) if need be!
CS50 is a programming environment at lab.cs50.io for scaffolded learning that enables

- teachers to create step-by-step programming lessons (labs), providing incremental feedback at each step, and
- students to progress from an empty file (or starter code) to working code, with hints and feedback along the way.

To create a lab as a teacher, all you need is a GitHub account and a (public or private) repository. To log into and work on a lab as a student, all you need is the former. Neither use case requires familiarity with git itself though if or once comfortable with git, you can create labs even more quickly via a command line!

CS50 Lab is essentially an extension of CS50 Sandbox that allows teachers to embed interactive instructions alongside a sandbox. As such, CS50 Lab is, also, essentially a lightweight version of CS50 IDE with problems’ specifications embedded in students’ actual programming environments.

URLs of labs are of the form https://lab.cs50.io/:owner/:repo/:branch/:path, where

- :owner is the lab’s owner, a GitHub user or organization, on github.com,
- :repo is that owner’s repository in which the lab’s source can be found,
- :branch is the branch on which the lab’s source can be found in that repository, and
- :path is the path to the lab’s source on that branch.

The source for a lab like https://lab.cs50.io/cs50/labs/python/loops/ can thus be found in https://github.com/cs50/labs/tree/python/loops/, wherein

- cs50 is the lab’s owner (hey, that’s us),
- labs is the lab’s repository,
- python is the lab’s branch,
- loops is the lab’s path, and heretofore unmentioned,
- tree is just a GitHub-specific trick, sandwiched between :repo and :branch, via which you can browse that branch and path.
36.1 Creation

To create a lab:

1. **Sign up** for a (free) GitHub account, if you don’t have one already.
2. **Create a repository**, if you don’t have one (that you’d like to use) already.
3. **Create a file** in that repository called `.cs50.yml`, optionally inside of one or more directories, using GitHub’s website. Or create (and push) the same using `git` itself. Configure `.cs50.yml` *per below*.
4. Optionally create another file in the same directory as `.cs50.yml` called `README.md`, configured *per below*. While technically optional, without this file your lab won’t have instructions!
5. Optionally create in or upload to that directory (or any descendent thereof) any files you’d like to install in a student’s environment (and automatically open in the text editor’s tabs).

You can then (assuming no mistakes!) visit https://lab.cs50.io/:owner/:repo/:branch/:path, where each of those placeholders is *as above*, to see your lab!

36.2 Configuration

36.2.1 `.cs50.yml`

To define a lab, it suffices to create a file called `.cs50.yml` in the root (or subdirectory) of a branch in a repository that contains, minimally, a top-level `lab50` key, the value of which is `true`:

```
lab50: true
```

**window**

It turns out *the above* is an abbreviation of (and equivalent to)

```
lab50:
    window:
    - editor
    - readme
    - terminal
```

wherein

- **editor** signifies that the lab should have an embedded text editor,
- **readme** signifies that the lab has instructions (written in `README.md`), and
- **terminal** signifies that the lab should have an embedded terminal window.

A value of `terminal` (implicit or explicit) is required.

Also available as values for `window` are

- **browser**, which signifies that the lab should have an embedded browser, and
- **x**, which signifies that the lab should have an embedded X window,

but those two values are mutually exclusive.
It’s worth noting that a lab without readme is functionally similar to CS50 Sandbox. Whereas sandboxes are intended to be temporary, labs are persistent: if a student logs into a lab and makes changes, those changes will persist indefinitely (unless the student resets the lab).

### files

To install files in students’ environments (e.g., foo.c and foo.h), add a key below lab50 called files (as a sibling of window, if explicitly present):

```yaml
lab50:
  files:
  - !include foo.c
  - !include foo.h
```

That !include is a (confusing) feature of YAML; it indeed means “include,” not “don’t include,” as a programmer might otherwise assume.

If those files exist (in the same directory as .cs50.yml), they will be copied into students’ environments and opened automatically (if recognized as text files). If those files don’t exist, they will be created as empty (and opened).

Files (e.g., bar.c and bar.h) can also be in subdirectories (of whatever directory .cs50.yml is in):

```yaml
lab50:
  files:
  - !include foo/bar.c
  - !include foo/bar.h
```

Alternatively, you can specify subdirectories:

```yaml
lab50:
  files:
  - !include foo/
```

Globbing is also supported, but asterisks have special meaning in YAML, so take care to quote any strings that have wildcards:

```yaml
lab50:
  files:
  - !include "foo/*".c
  - !include "foo/*".h
```

You can also exclude files, as with:

```yaml
lab50:
  files:
  - !exclude "*"
  - !include "foo.*"
```

The value of files is an ordered list, top to bottom, so the above means that all files are excluded by default but foo.* is then included, thereby overriding their exclusion.

**36.2.2 README.md**

A lab’s instructions should be written in README.md (which must be in the same directory as .cs50.yml), using GitHub-flavored Markdown. Via CS50-specific “tags” you can add interactive features to those instructions. If present, each should appear on a line of its own but might very work in other contexts too (e.g., in ordered or unordered lists).
Your Markdown can also contain, if need, raw HTML, but not these tags. Your Markdown can also contain emoji.

**next**

To paginate a lab’s instructions, inserting a Next button and hiding, until clicked, everything below it, you can use this tag:

```
{% next %}
```

You can override the button’s label with a quoted string:

```
{% next "Step 2" %}
```

**spoiler**

To provide students with a spoiler, code or information they should only by clicking a Spoiler button, you can use these tags:

```
{% spoiler %}
The Answer to the Great Question...
Of Life, the Universe and Everything...
Is...
Forty-two.
{% endspoiler %}
```

You can override the button’s label with a quoted string. Accordingly, via

```
{% spoiler "Hint" %}
You're really not going to like it.
{% endspoiler %}
```

could you provide students with a hint. And via

```
{% spoiler "Solution" %}
Forty-two.
{% endspoiler %}
```

could you provide students with a solution.

**video**

To embed a YouTube video (responsively) in a lab’s instructions, you can use this tag, wherein the URL can be any URL of a video on YouTube:

```
{% video https://www.youtube.com/watch?v=oHg5SJYRHA0 %}
```

### 36.3 Acknowledgements

Special thanks to CS50’s friends at Next Tech and Google for their support of this app!
CS50 Sandbox is a web app at sandbox.cs50.io, for students and teachers especially, that allows users to

- create temporary programming environments (sandboxes) quickly and
- share copies of those sandboxes with others.

For instance, a teacher might

- begin a class by asking students to visit sandbox.cs50.io and start a sandbox for C with a terminal window and text editor, with a file called hello.c, all of which the app’s landing page facilitates,

- provide students with a pre-configured link like https://sandbox.cs50.io/?file=hello.c&window=editor&window=terminal (perhaps shortened via a URL shortener) that, when visited, would yield the same, or

- provide students with starter code, as by creating a sandbox, editing one or more files, and asking students to clone that sandbox by visiting its unique URL.

CS50 Sandbox is essentially a lightweight version of CS50 IDE, CS50’s web-based integrated development environment, but CS50 Sandbox

- does not have debug50, CS50 IDE’s graphical debugger for C and Python,
- does not offer file revision history, and
- does not support real-time collaboration or chat.

To use CS50 Sandbox, students and teachers need only have a (free) GitHub account via which to log in.

### 37.1 Configuration

CS50 Sandbox supports, via both GET and POST, these HTTP parameters:

- **file**, which, if present, is a file path to pre-create within the sandbox in /root/sandbox. If window has a value of editor (potentially among other values), that file, if not binary, will also be pre-opened in the code editor.
• window, which must have a value of terminal (so that the sandbox will have a terminal window) and may have additional values of
  – browser, in which case the sandbox will have an embedded browser, pre-configured with an address of http://localhost:8080/,
  – editor, in which case the sandbox will have a code editor, and/or
  – x, in which case the sandbox will have an embedded X window.

For GET, then, the shortest supported URL is https://sandbox.cs50.io/?window=terminal.

A URL with multiple values for window, meanwhile, might be https://sandbox.cs50.io/?window=editor&window=terminal.

Values of browser and x for window are mutually exclusive.

Unsupported parameterizations will yield an HTTP status code of 400.

When submitted via POST, each value of file can be an actual file encoded as multipart/form-data, as via a form like the below:

```html
<form action="https://sandbox.cs50.io/" enctype="multipart/form-data" method="post">
  <input multiple name="file" type="file">
  <input name="window" type="hidden" value="editor">
  <input name="window" type="hidden" value="terminal">
  <input type="submit">
</form>
```

37.2 Related

37.3 Acknowledgements

Special thanks to CS50’s friends at Next Tech and Google for their support of this app!
CS50 Vault is a web app at vault.cs50.io that allows you to add authentication to any link, requiring that users authenticate via edX, HarvardKey, or Yale NetID in order to access it, thereby allowing you to post a link online while still restricting access.

Upon authenticating, a user will be redirected to the link, unless the link can be rendered in an iframe, in which case its URL will be masked. Alternatively, if the link is to a:

- file on dropbox.com, the link can be customized to trigger an automatic download of the file
- folder on dropbox.com, the link can be customized to trigger an automatic download of a ZIP of the folder
- video on youtube.com (or youtu.be), the link can be customized to embed the video in an iframe such that it fills the user’s window and autoplays

### 38.1 Notes

- After authenticating, a user could theoretically share a link with anyone on the internet (as via email or any other mechanism), thereby allowing others to access it without authentication. However, once accessed by one user, most any digital asset can be shared with (or copied for) others. CS50 Vault simply raises a bar thereto.
- Embedding a YouTube video in an iframe obfuscates, but does not prevent discovery of, the video’s underlying URL, which is exposed via HTML. After authenticating, a user could theoretically share that link as well.
- A link will only be rendered in an iframe if it
  - is not served with an X-Frame-Options header,
  - is not served with a Content-Security-Policy header, the value of which contains frame-ancestors, and
  - is served with a Content-Type header, the value of which is application/pdf or text/html.
Python Module Index

C
check50, 9
check50.c, 13
check50.flask, 14
check50.internal, 16
check50.py, 15
compare50, 51
compare50.comparators, 55
compare50.passes, 54
compare50.preprocessors, 54
A
after_check() (check50.internal.Register method), 16
after_every() (check50.internal.Register method), 16
app (class in check50.flask), 14
append_code() (in module check50.py), 15

B
before_every() (check50.internal.Register method), 16
by_character() (in module compare50.preprocessors), 54

C
CC (in module check50.c), 13
CFLAGS (in module check50.c), 13
check() (in module check50), 12
check50 (module), 9
check50.c (module), 13
check50.flask (module), 14
check50.internal (module), 16
check50.py (module), 15
check_dir (in module check50.internal), 16
check_running (in module check50.internal), 16
comments() (in module compare50.preprocessors), 54
Comparator (class in compare50), 51
compare() (compare50.Comparator method), 51
compare() (compare50.comparators.Misspellings method), 55
compare() (compare50.comparators.Winnowing method), 55
compare() (in module compare50), 53
compare50 (module), 51
compare50.comparators (module), 55
compare50.passes (module), 54
compare50.preprocessors (module), 54
Comparison (class in compare50), 51
compile() (in module check50.c), 13
compile() (in module check50.py), 16
compile_checks() (in module check50.internal), 16
CONFIG_LOADER (in module check50.internal), 16
content() (check50.flask.app method), 14

data() (in module check50), 9

E
EOF, 13
Error, 16, 51
exact (class in compare50.passes), 54
exists() (in module check50), 9
exit() (check50.run method), 10
expand() (in module compare50), 53
extract_identifiers() (in module compare50.preprocessors), 54

F
Failure, 12
File (class in compare50), 51

G
get() (check50.flask.app method), 14
get() (compare50.File class method), 52
get() (compare50.Submission class method), 53
get_char (C function), 106
get_double (C function), 107
get_float (C function), 108
get_int (C function), 107
get_long (C function), 109
get_string (C function), 109

H
hash() (in module check50), 10

I
import_() (in module check50.py), 16
import监督检查() (in module check50), 9
import_file() (in module check50.internal), 17
include() (in module check50), 10

K
n kill() (check50.run method), 11

L
lexer() (compare50.File method), 52
load_config() (in module check50.internal), 17
log() (in module check50), 11

M
Mismatch, 12
missing_spans() (in module compare50), 53
Misspellings (class in compare50.comparators), 55
misspellings (class in compare50.passes), 54

N
normalize_builtin_types() (in module compare50.preprocessors), 54
normalize_case() (in module compare50.preprocessors), 55
normalize_identifiers() (in module compare50.preprocessors), 55
normalize_numeric_literals() (in module compare50.preprocessors), 55
normalize_string_literals() (in module compare50.preprocessors), 55

P
Pass (class in compare50), 52
path (compare50.File attribute), 52
post() (check50.flask.app method), 15

R
rank() (in module compare50), 54
raw_content() (check50.flask.app method), 15
read() (compare50.File method), 52
Register (class in check50.internal), 16
register (in module check50.internal), 17
reject() (check50.run method), 11
run (class in check50), 10
run_dir (in module check50.internal), 17

S
Score (class in compare50), 52
score() (compare50.Comparator method), 51
score() (compare50.comparators.Misspellings method), 55
score() (compare50.comparators.Winnowing method), 55
Span (class in compare50), 52

def split_on_whitespace() (in module compare50.preprocessors), 55
def status() (check50.flask.app method), 15
def stdin() (check50.run method), 11
def stdout() (check50.run method), 11

def strip_comments() (in module compare50.preprocessors), 55
def strip_whitespace() (in module compare50.preprocessors), 55

def structure (class in compare50.passes), 54
def Submission (class in compare50), 52
def text_printer() (in module compare50.preprocessors), 55
def Token (class in compare50), 53
def token_printer() (in module compare50.preprocessors), 55
def tokens() (compare50.File method), 52

def unprocessed_tokens() (compare50.File method), 52

def words() (in module compare50.preprocessors), 55