Galah Interact Documentation

Release v0.2.1

Galah Group LLC and other contibuters as specified in CONTRIBU

Sep 27, 2017

Contents

This is the main documentation for the Galah Interact project. The contents of this site are automatically generated via Sphinx based on the Python docstrings throughout the code and the reStructuredText documents in the docs/ directory of the git repository. If you find an error in the documentation, please report it in the bug tracker here (which we share with the Galah project), or even better, submit a pull request!

CHAPTER 1

What is Galah Interact

Galah Interact is a library designed to make it very easy to create Test Harnesses that grade student's assignments. It provides code to perform a number of tests that many instructors care about (such as checking that code is properly indented, or that code compiles without any warnings) along with providing utilities to make more complicated testing much easier (unit testing for example is extremely easy with this library, as you can *simply import all of the student's functions and classes into Python*).

The reason for Galah Interact's creation was to make it easier to create Test Harnesses for Galah, however, we don't have any intention on locking in this library's use (this is why we've released it under the very permissive *Licensing*). The real goal of Galah Interact is to provide a powerful framework for creating excellent test harnesses no matter what the submission system you are using for your class is.

Galah Interact can only test C++ code right now. We would really like to support other languages as well, so if you are interested in using this library at your university that teaches using a different language, please let us know so we can work with you.

CHAPTER 2

Tutorials, Examples, and Guides

If you have not used this library before, you should start your journey by going through these tutorials.

Getting Started

Step 1 - Installing Galah Interact

Galah Interact is a Python library. As such, the first thing you need to do is get it installed and get to the point where you can import the library.

Note that Galah Interact only officially supports Linux at this point as the developers don't have access to Macs. If you have a Mac and would like to help us test the library, please let us know. Windows users, if you have a great desire to use Galah Interact please let us know and we will try to add support.

If you want to avoid affecting the entire system, you can use a virtualenv with any of the methods, though unless you are already familiar with Python virtual environments it may be more trouble than its worth.

To install Galah Interact you can choose from one of the three methods below.

Method 1 - Installing through pip

Galah Interact may be installed simply using the Python package manager pip (to get pip see this page). This method of installation is the recommended method as it will make it a little easier to upgrade your installation in the future, but if you do not already have pip set up and want to go the easiest route, try Method 2 instead.

After getting pip installed, you should be able to execute `pip install galah-interact` on the command line. After doing so Galah Interact will be installed automatically.

Method 2 - Installing from Source

If you do not wish to install pip, you can install Galah Interact a little more directly, just follow the steps below.

1. Go and download an archive containing Galah Interact's source either from the list of tagged releases or from the main project page (this will grab the latest code in the repo).
2. Unpack the archive somewhere.
3. Find the file `setup.py` in the top-level directory and set its executable bit (`chmod +x setup.py`).
4. Execute `./setup.py install`.

Method 3 - Using Directly

If you don't want to deal with installing the library, you can use it directly by unpacking the archive as above, and then just copying the `interact/` directory into the same directory as your test harness.

Step 2 - Hello World

The typical first program for Computer Science students is the "Hello World" program. The first Test Harness we create, then, will be a Test Harness that grades a "Hello World" assignment. The harness is below, you can go ahead and place it in a file with a `.py` extension, set the executable bit (`chmod +x file.py`) and then run it with `./file.py --mode test`.

The comments document each line thoroughly and make up the contents of this tutorial, so please read through them carefully. I assume minimal Python knowledge, so I try to describe any advanced Python features enough that you can search for more information on them online.

```
#!/usr/bin/env python

import interact
import os.path

# The Harness class is an idempotent object you should only ever make one of
# that takes care of a lot of the boilerplate for you. You should always copy
# and paste these two lines of code to the top of every test harness you create.
harness = interact.Harness()
harness.start()

# Get a list of absolute paths to the student's files (in this case, we only
# care about their main.cpp file).
student_files = harness.student_files("main.cpp")

# The line below starting with @harness.test is how we designate that certain
# functions are test functions that return TestResult objects (more on this
# in a bit). The argument here is what the user sees as the name of the test
# that is run, the function name has no real significance and is only used to
# reference the test function when needed.
@harness.test("Proper files exist.")
def check_files():
        # Checking to make sure certain files exist is a very typical thing for
        # Test Harnesses to do so Galah Interact ships with a standard test to do
        # it for you. The * before student files is expanding the list student_files
        # such that each item in the list will be treated as a seperate argument to
        # the function (this is because check_files_exist takes in variadic
        # arguments, ex: check_files_exist("main.cpp", "foo.cpp", "bar.cpp")).
    return interact.standardtests.check_files_exist(*student_files)

# Notice the `depends = [check_files]` line here. This is how you can create
# dependencies between tests. Here we are saying that this test should only be
```

```
# run if the check_files test passed. Any number of tests can be given there
# (this is a Python list) and Galah Interact will make sure to run things in the
# correct order.
@harness.test("Program compiles correctly.", depends = [check_files])
def check_compilation():
        # Similar to checking if files exist, checking to see if a student's code
        # compiles is also a very common test, so Galah Interact has a function to
        # make it very simple.
    return interact.standardtests.check_compiles(student_files)

@harness.test("Program prints out hello world.", depends = [check_compilation])
def check_output():
        # This is the first time you actually get a good look at a TestResult
        # object. The standard tests above actually return an instance of this
        # class. The brief argument is some text that is always displayed no matter
        # what the result of the testing is. You should try to describe the test
        # that is being run briefly here, so students are not confused. The
        # default message argument is only displayed if you don't add any other
        # messages to the TestResult object.
        result = interact.TestResult(
                brief = "This test ensures that your program prints out a single line
↪"
                        "of output that reads `Hello World!`.",
                default_message = "**Great job!** Your program correctly prints out "
                                  "`Hello World!`.",
                max_score = 10
        )

        # Interact has a handy function that will automatically compile and run
        # some code files. If the code has already been compiled by interact, it
        # will not recompile it, but rather it will use the executable that was
        # already compiled, so don't worry about extraneous work being done here.
        # run_program returns a tuple and it is being unpacked into the variables
        # output, stderr, and return_code.
        output, stderr, return_code = \
                interact.execute.run_program(student_files)

        # Get a list of lines in the output, ignoring any blank lines. The thing
        # surrounded by square brackets is called a list comprhension and is a very
        # useful Python feature that you should familiarize yourself with.
        output_lines = [i for i in output.splitlines() if i]

        # Here is my first actual test. I see if the user printed out any extra
        # lines.
        if len(output_lines) != 1:
                # By adding this message, the default message I defined when I created
                # the TestResult instance will not be shown. Notice how I insert␣
↪nlines
                # into the messages, you can do this with any number of values. dscore
                # is a special value that you can set to assign a "change of score",␣
↪or
                # delta score, to the message. Here I am signifying that if this␣
↪message
                # is added, the score should shrink by 5.
                result.add_message(
                        "Your program output {nlines} line(s), remember, your program
↪"
                        "should print out exactly 1 line of output.",
```

```
                        nlines = len(output_lines),
                        dscore = -5
                )

        # This is another Python construct that may look strange to those
        # unfamiliar with it. Notice that the for loop actually has an else attached
        # to it. The code in the else block will only be executed if we never break
        # from within the for loop.
        for i in output_lines:
                if i == "Hello World!":
                        break
        else:
                # If we add this message along with the one above they will simply be
                # displayed one after the other in a bulleted list (even if only one
                # is displayed they will still be put into a bulletted list for
                # consitency, though it is possible to override this behavior).
                result.add_message(
                        "Your program did not print out a line that reads `Hello␣
↪World!`.",
                        dscore = -5
                )

        # calculate_score() is a convenience function that automatically tallies up
        # the dscores of the various messages that have been added and assigns an
        # appropriate total score to the TestResult. It takes a couple of optional
        # arguments to suite most styles of grading, so check out the reference
        # material on this function.
        result.calculate_score()

        # Test functions always return TestResult. All of the standard tests return
        # TestResult objects (and you can actually inspect the TestResult objects
        # they return in order to customize the grading scale and such).
        return result

# This function will execute each of the test functions in the proper order
# based on their dependencies and save the results within the Harness object.
harness.run_tests()

# This function will output the results (either as readable text if you used
# the --test . . arguments to start the test harness, or as JSON appropriate
# for Galah's test servers to read in if not). Make sure to set the max_score
# appropriately because unfortunately the harness will not be able to correctly
# guess the appropriate max_score in all cases (notably when some test functions
# aren't run due to dependencies failing).
harness.finish(max_score = 21)
```

You can download the code above `here`.

When you execute a test harness file directly during testing/development, you should always execute it with the command line arguments `--mode test`. If you do not do this, when you run the test harness it will appear to freeze up. It is actually waiting for JSON from standard input. This is how the test servers in Galah communicate with test harnesses. If you encounter this, just use `Ctrl+C` to kill the program and start it again properly.

See also:

For more information on command line arguments, check out *Test Harness Command Line Interface*.

Advanced users can also use the Sheep Simulator to run the test harnesses as if the harness was actually within a Galah test server (which are called sheep). This should be unnecessary in most cases however, and the simulator was created

before Galah Interact had the capability to be run in this testing mode.

Unit Testing

One of the most interesting modules in Galah Interact is the *interact.unittest* module. It allows you to "load" any number of C++ files such that all of the native C++ classes and functions become available as Python classes and functions. This is done by leveraging the absolutely fantastic SWIG library which can automatically create bindings from C++ to a plethora of other languages. Please give the SWIG project as much support as you can as it is really a wonderful product.

In order to use the *interact.unittest* module, you need to make sure that you have SWIG installed, and that you have *Python development headers* installed, both of which are probably available through your distribution's package manager (`apt-get` or `yum` for example).

Basic Unit Testing

Once you have it installed, you can start poking around with the following code.

`main.cpp`

```
#include <iostream>

using namespace std;

class Foo {
        int a_;
public:
        Foo(int a) : a_(a) {
                // Do nothing
        }

        Foo() : a_(0) {
                // Do nothing
        }

        int get_a() const {
                return a_;
        }
};

int bar(int a, int b) {
        cout << "a: " << a << " b: " << b << endl;
        return a + b;
}

int main() {
        return 0;
}
```

`harness.py`

```
import interact

student_code = interact.unittest.load_files(["main.cpp"])
```

```
print "---Running bar(3, 4)---"
return_value = student_code["main"]["bar"](3, 4)
print return_value

print "---Creating new Foo objects---"
Foo = student_code["main"]["Foo"]
new_foo = Foo(3)
new_foo_default = Foo()

print "---Printing get_a() on each foo instance---"
print new_foo.get_a()
print new_foo_default.get_a()

print "---Printing a_ value directly on each foo instance---"
print new_foo.a_
print new_foo_default.a_
```

Running the above Python script gives the following output:

```
---Running bar(3, 4)---
a: 3 b: 4
7
---Creating new Foo objects---
---Printing get_a() on each foo instance---
3
0
---Printing a_ value directly on each foo instance---
3
0
```

It should be clear that this makes it fairly easy to unit test some student's code. You may notice that the `bar` function above prints out to standard output. This is a little problematic if you want to test what that function outputs, and it's actually even more problematic in that if your harness prints things out to standard output when in Galah, the test server will get angry because you'll make it so that the output is no longer proper JSON. To solve this, there is another handy library called *interact.capture*.

Using the `interact.capture` module

The *interact.capture* module exposes a function *capture_function* that forks a process before running a given function, and captures anything written to `stdout` or `stderr` (and even lets you control `stdin`). All while also allowing you to get the return value of the function and seeing any exceptions that are raised.

Using this function, we can test the `bar` function trivially.

```
import interact
from interact.capture import capture_function

student_code = interact.unittest.load_files(["main.cpp"])

captured_function = capture_function(student_code["main"]["bar"], 3, 4)

# Wait for the function to end.
captured_function.wait()

print "The function returned:", repr(captured_function.return_value)
print "The function wrote to stdout:", repr(captured_function.stdout.read())
print "The function wrote to stderr:", repr(captured_function.stderr.read())
```

Note that *capture_function* returns a special *CapturedFunction* object. You should briefly glance over its documentation to get an understanding of what it does and why it exists.

Running the above Python scripts outputs:

```
The function returned: 7
The function wrote to stdout: 'a: 3 b: 4\n'
The function wrote to stderr: ''
```

Go ahead and try to play around with *interact.unittest* and *interact.capture*. There is a lot of very cool things you can do with them!

CHAPTER 3

Reference Material

To get documentation on a specific function or module, or if you just want to browse through all of what Galah Interact has to offer, check out the below pages.

Interact Framework

Galah Interact is a framework for creating Test Harnesses that grade students' programming assignments. For general project information, check out the project page on GitHub: https://github.com/galah-group/galah-interact-python

Other contributers have given their valuable time to this project in order to make it better, and they are listed in the CONTRIBUTERS file in the root directory of the git repo.

`interact.core`

Functions and classes that are essential when using this library. Is imported by `interact/__init__.py` such that `interact.core.x` and `interact.x` are equivalent.

class `interact.core.`**Harness**

An omniscient object responsible for driving the behavior of any Test Harness created using Galah Interact. Create a single one of these when you create your Test Harness and call the `start()` method.

A typical Test Harness will roughly follow the below format.

```
import interact
harness = interact.Harness()
harness.start()

# Your code here
```

```
harness.run_tests()
harness.finish(max_score = some_number)
```

Variables

- **sheep_data** – The "configuration" values received from outside the harness (either from Galah or command line arguments). See *Test Harness Command Line Interface*.
- **execution_mode** – The mode of execution the harness is running in. Is set by *Harness.start()* and is None before it is set. For information on the different modes, check out *Test Harness Command Line Interface*.
- **tests** – A dictionary mapping test functions to *Harness.Test* objects. This is of type *ORDERED_DICT*.

class FailedDependencies (*max_score=10*)

A special *TestResult* used by *Harness.run_tests()* whenever a test couldn't be run do to one of its dependencies failing.

```
>>> a = interact.Harness.FailedDependencies()
>>> a.add_failure("Program compiles")
>>> a.add_failure("Program is sane")
>>> print a
Score: 0 out of 10

This test will only be run if all of the other tests it depends on pass first.
↪ Fix those tests *before* worrying about this one.

 * Dependency *Program compiles* failed.
 * Dependency *Program is sane* failed.
```

add_failure (*test_name*)

Adds a new failure message to the test result signifying a particular dependency has failed.

class Harness.**Test** (*name*, *depends*, *func*, *result=None*)

Meta information on a single test.

Harness.**finish** (*score=None*, *max_score=None*)

Marks the end of the test harness. When start was not initialized via command line arguments, this command will print out the test results in a human readable fashion. Otherwise it will print out JSON appropriate for Galah to read.

Harness.**run_tests**()

Runs all of the tests the user has registered.

Raises Harness.CyclicDependency if a cyclic dependency exists among the test functions.

Any tests that can't be run due to failed dependencies will have instances of *Harness.FailedDependencies* as their result.

Harness.**start** (*self, arguments = sys.argv[1:]*)

Takes in input from the proper source, initializes the harness with values it needs to function correctly.

Parameters **arguments** – A list of command line arguments that will be read to determine the harness's behavior. See below for more information on this.

Returns None

See also:

Test Harness Command Line Interface

`Harness.`**`student_file`**(*filename*)
Given a path to a student's file relative to the root of the student's submission, returns an absolute path to that file.

`Harness.`**`student_files`**(**args*)
Very similar to student_file. Given many files as arguments, will return a list of absolute paths to those files.

`Harness.`**`test`**(*name*, *depends=None*)
A decorator that takes in a test name and some dependencies and makes the harness aware of it all.

`interact.core.`**`ORDERED_DICT`**
An OrderedDict type. The stdlib's `collections` module is searched first, then the module ordereddict is searched, and finally it defaults to a regular `dict` (which means that the order that test results are displayed will be undefined). This is the type of *`Harness.tests`*. This complexity is required to support older versions of Python.

alias of `OrderedDict`

class `interact.core.`**`TestResult`**(*brief=None*, *score=None*, *max_score=None*, *messages=None*, *default_message=None*, *bulleted_messages=True*)
Represents the result of one unit of testing. The goal is to generate a number of these and then pass them all out of the test harness with a final score.

Variables

- **`brief`** – A brief description of the test that was run. This will always be displayed to the user.
- **`score`** – The score the student received from this test.
- **`max_score`** – The maximum score the student could have received from this test.
- **`messages`** – A list of *`TestResult.Message`* objects that will be joined together appropriately and displayed to the user. Use *`add_message()`* to add to this list.
- **`default_message`** – A string that will be displayed if there are no messages. Useful to easily create a "good job" message that is shown when no problems were detected.
- **`bulleted_messages`** – A boolean. If `True`, all of the messages will be printed out in bullet point form (a message per bullet). If `False`, they will simply be printed out one-per-line. You can set this to `False` if only one message will ever be displayed to the user, otherwise bullet points usually look better.

class `Message`(*text*, **args*, ***kwargs*)
A message to the user. This is the primary mode of giving feedback to users.

`TestResult.`**`add_message`**(**args*, ***kwargs*)
Adds a message object to the TestResult. If a Message object that is used, otherwise a new Message object is constructed and its constructor is passed all the arguments.

`TestResult.`**`calculate_score`**(*starting_score=None*, *max_score=None*, *min_score=None*)
Automatically calculates the score by adding up the `dscore` of each message and setting the score of the *`TestResult`* appropriately.

Parameters

- **`starting_score`** – This score is added to the sum of every message's `dscore`. If `None`, `max_score` is used.

- **max_score** – The `max_score` field of the object is set to this value. If `None`, the current `max_score` is used, i.e. no change is made.
- **min_score** – If the calculated score is less than this value, this value is used instead.

Returns `self`. This allows you to return the result of this function from test functions.

```
>>> a = TestResult(max_score = 4)
>>> a.add_message("Foo", dscore = -1)
>>> a.add_message("Bar!", dscore = -5)
>>> print a.calculate_score().score
-2
>>> print a.score
-2
>>> print a.calculate_score(min_score = 0).score
0
>>> print a.calculate_score(starting_score = 8, max_score = 6).score
2
>>> print a.max_score
6
```

`TestResult.`**`is_failing`**`()`

Returns The inverse of what *`is_passing()`* returns.

This function is most useful when dealing with a `TestResult` that you want to consider either passing or failing, and never anything in between.

See also:

`set_passing()` and *`is_passing()`*.

`TestResult.`**`is_passing`**`()`

Returns `True` if the score is not `0` (note this function *will* return `True` if the score is negative).

This function is most useful when dealing with a `TestResult` that you want to consider either passing or failing, and never anything in between.

See also:

`set_passing()` and *`is_failing()`*.

`TestResult.`**`set_passing`**`(`*passing*`)`

Parameters **`passing`** – Whether the test is passing or not.

Returns `self`. This allows you to return the result of this function directly, leading to more concise test functions.

This function sets `score` to either 1 (if `passing` is `True`) or 0 (if `passing` is `False`). It also sets the `max_score` to `1`.

See also:

`is_passing()` and *`is_failing()`*.

class `interact.core.`**`UniverseSet`**`(`*iterable=None*`)`

A special `set` such that every `in` query returns `True`.

```
>>> a = UniverseSet()
>>> "hamster" in a
True
>>> "apple sauce" in a
True
```

```
>>> 3234 in a
True
>>> "taco" not in a
False
```

interact.core.**json_module**()
A handy function that will try to find a suitable JSON module to import and return that module (already loaded).

Basically, it tries to load the `json` module, and if that doesn't exist it tries to load the `simplejson` module. If that doesn't exist, a friendly `ImportError` is raised.

interact.execute

interact.execute.**compile_program**(*files*, *flags=[]*, *ignore_cache=False*)
Compiles the provided code files. If ignore_cache is False and the program has already been compiled with this function, it will not be compiled again.

Parameters

- **files** – A list of files to compile.
- **flags** – A list of flags to pass to `g++`. See *create_compile_command()* for information on how exactly these are used.
- **ignore_cache** – If `True`, the cache will not be used to service this query, even if an already compiled executable exists. See below for more information on the cache.

Returns A two-tuple `(compiler output, executable path)`. If the executable was loaded from the cache, the compiler output will be `None`. If the program did not compile successfully, the executable path will be `None`.

Note: Note that this function blocks for as long as it takes to compile the files (which might be quite some time). Of coures if the executable is loaded from the cache no such long wait time will occur.

This function caches its results so that if you give it the same files to compile again it will not compile them over again, but rather it will immediately return a prepared executable. The cache is cleared whenever the program exits.

interact.execute.**create_compile_command**(*files*, *flags*)
From a list of files and flags, crafts a list suitable to pass into `subprocess.Popen` to compile those files.

Parameters

- **files** – A list of files to compile.
- **flags** – A list of flags to pass onto `g++`. `-o main` will always be passed after these flags.

Returns A list of arguments appropriate to pass onto `subprocess.Popen`.

```
>>> create_compile_command(["main.cpp", "foo.cpp"], ["-Wall", "-Werror])
["g++", "-Wall", "-Werror", "-o", "main", "main.cpp", "foo.cpp"]
```

interact.execute.**default_run_func**(*executable*, *temp_dir*)
Used by the *run_program()* to create a `Popen` object that is responsible for running the exectuable.

Parameters

- **executable** – An absolute path to the executable that needs to be run.

- **temp_dir** – An absolute path to a temporary directory that can be used as the current working directory. It will be deleted automatically at the end of the *run_program()* function. The executable will not be in the directory.

This function may be overriden to override the default `run_func` value used in the *run_program()* function.

Warning: You **must** pass in `subprocess.PIPE` to the `Popen` constructor for the `stdout` and `stdin` arguments.

You can use this function as a reference when creating your own run functions to pass into *run_program()*.

`interact.execute.`**run_program**(*files=None*, *given_input=''*, *run_func=None*, *executable=None*)
Runs a program made up of some code files by first compiling, then executing it.

Parameters

- **files** – The code files to compile and execute. *compile_program()* is used to compile the files, so its caching applies here.
- **given_input** – Text to feed into the compiled program's standard input.
- **run_func** – A function responsible for creating the `Popen` object that actually runs the program. Defaults to *default_run_func()*.
- **executable** – If you don't need to compile any code you can pass a path to an executable that will be executed directly.

Returns A three-tuple containing the result of the program's execution `(stdout, stderr, returncode)`.

interact.parse

This module is useful when attempting to roughly parse students' code (ex: trying to check that indentation was properly used). This module does not attempt to, and never will, try and fully parse C++. If such facilities are added to Galah Interact they will probably be added as a seperate module that provides a nice abstraction to Clang.

class `interact.parse.`**Block**(*lines*, *sub_blocks=None*)
Represents a block of code.

Variables

- **lines** – A list of `Line` objects that make up this block.
- **sub_blocks** – A list of `Block` objects that are children of this block.

`interact.parse.`**INDENT_EXCEPTED_LINES = ['public:', 'private:', 'protected:']**
Lines of code to ignore when looking for bad indentation. See *find_bad_indentation()* for more information.

class `interact.parse.`**Line**(*line_number*, *code*)
Represents a line of code.

Variables

- **code** – The contents of the line.
- **line_number** – The line number.

indent_level()
Determines the indentation level of the current line.

Returns The sum of the number of tabs and the number of spaces at the start of the line. Iff the line is blank (not including whitespace), `None` is returned.

static `lines_to_str` (*lines*)

Creates a single string from a list of `Line` objects.

Parameters **`lines`** – A list of `Line` objects.

Returns A single string.

```
>>> my_lines = [
    Line(1, "int main() {"),
    Line(2, "    return 0;"),
    Line(3, "}")
]
>>> Line.lines_to_str(my_lines)
"int main() {\n    return 0\n}\n"
```

static `lines_to_str_list` (*lines*)

Creates a list of strings from a list of `Line` objects.

Parameters **`lines`** – A list of `Line` objects.

Returns A list of strings.

```
>>> my_lines = [
    Line(1, "int main() {"),
    Line(2, "    return 0;"),
    Line(3, "}")
]
>>> Line.lines_to_str_list(my_lines)
[
    "int main() {",
    "    return 0;",
    "}"
]
```

classmethod `make_lines` (*lines*, *start=1*)

Creates a list of Line objects from a list of strings representing lines in a file.

Parameters

- **`lines`** – A list of strings where each string is a line in a file.
- **`start`** – The line number of the first line in `lines`.

Returns A list of line objects.

```
>>> Line.make_lines(["int main() {", "    return 0;", "}"], 1)
[
    Line(1, "int main() {"),
    Line(2, "    return 0;"),
    Line(3, "}")
]
```

`interact.parse.`**`cleanse_quoted_strings`** (*line*)

Removes all quoted strings from a line. Single quotes are treated the same as double quotes.

Escaped quotes are handled. A forward slash is assumed to be the escape character. Escape sequences are not processed (meaning " does not become ", it just remains as ").

Parameters **`line`** – A string to be cleansed.

Returns The line without any quoted strings.

```
>>> cleanse_quoted_strings("I am 'John Sullivan', creator of worlds.")
"I am , creator of worlds."
>>> cleanse_quoted_strings(
...     'I am "John Sullivan \"the Destroyer\", McGee", fear me.'
... )
"I am , fear me."
```

This function is of particular use when trying to detect curly braces or other language constructs, and you don't want to be fooled by the symbols appearing in string literals.

interact.parse.**find_bad_indentation**(*block*, *minimum=None*)

Detects blocks of code that are not indented more than their parent blocks.

Parameters

- **block** – The top-level block of code. Sub-blocks will be recursively checked.
- **minimum** – The minimum level of indentation required for the top-level block. Mainly useful due to this function's recursive nature.

Returns A list of `Line` objects where each `Line` had a problem with its indentation.

Note: Lines that match (after removing whitespace) lines in *INDENT_EXCEPTED_LINES* will be ignored.

```
>>> my_block = Block(
...     lines = [
...         Line(0,  "#include <iostream>"),
...         Line(1,  ""),
...         Line(2,  "using namespace std;"),
...         Line(3,  ""),
...         Line(4,  "int main() {"),
...         Line(15, "}")
...     ],
...     sub_blocks = [
...         Block(
...             lines = [
...                 Line(5,  '    cout << "{" << endl;'),
...                 Line(6,  "    if (true)"),
...                 Line(7,  "    {"),
...                 Line(9,  "    } else {"),
...                 Line(12, "    }"),
...                 Line(13, "        pinata"),
...                 Line(14, "    return 0")
...             ],
...             sub_blocks = [
...                 Block(
...                     lines = [
...                         Line(8, "        return false;")
...                     ]
...                 ),
...                 Block(
...                     lines = [
...                         Line(10, "        return true;"),
...                         Line(11, "oh noz")
...                     ]
...                 )
```

```
...                 ]
...             )
...         ]
... )
>>> find_bad_indentation(my_block)
[Line(11, "oh noz")]
```

interact.parse.**grab_blocks**(*lines*)

Finds all blocks created using curly braces (does not handle two line if statements for example).

Parameters **lines** – A list of Line objects.

Returns A single Block object which can be traversed like a tree.

```
>>> my_lines = [
...     Line(0, "#include <iostream>"),
...     Line(1, ""),
...     Line(2, "using namespace std;"),
...     Line(3, ""),
...     Line(4, "int main() {"),
...     Line(5, '    cout << "Hello world" << endl;'),
...     Line(6, "    return 0"),
...     Line(7, "}")
... ]
>>> grab_blocks(my_lines)
Block(
    lines = [
        Line(0, "#include <iostream>"),
        Line(1, ""),
        Line(2, "using namespace std;"),
        Line(3, ""),
        Line(4, "int main() {"),
        Line(7, "}")
    ],
    sub_blocks = [
        Block(
            lines = [
                Line(5, '    cout << "Hello world" << endl;'),
                Line(6, "    return 0")
            ],
            sub_blocks = None
        )
    ]
)
```

(Note that I formatted the above example specially, it won't actually print out so beautifully if you try it yourself, but the content will be the same)

interact.pretty

Module useful for displaying nice, grammatically correct output.

interact.pretty.**craft_shell_command**(*command*)

Returns a shell command from a list of arguments suitable to be passed into subprocess.Popen. The returned string should only be used for display purposes and is not secure enough to actually be sent into a shell.

interact.pretty.**escape_shell_string**(*str*)

Escapes a shell string such that it is suitable to be displayed to the user. This function **should not** be used to

actually feed arguments into a shell as this function **is not** secure enough.

interact.pretty.**plural_if**(*zstring*, *zcondition*)
Returns zstring pluralized (adds an 's' to the end) if zcondition is True or if zcondition is not equal to 1.

Example usage could be `plural_if("cow", len(cow_list))`.

interact.pretty.**pretty_list**(*the_list*, *conjunction='and'*, *none_string='nothing'*)
Returns a grammatically correct string representing the given list. For example...

```
>>> pretty_list(["John", "Bill", "Stacy"])
"John, Bill, and Stacy"
>>> pretty_list(["Bill", "Jorgan"], "or")
"Bill or Jorgan"
>>> pretty_list([], none_string = "nobody")
"nobody"
```

interact.standardtests

This module contains useful test functions that perform full testing on input, returning TestResult objects. These are typical tests that many harnesses need to perform such as checking indentation or checking to see if the correct files were submitted.

interact.standardtests.**check_compiles**(*files*, *flags=[]*, *ignore_cache=False*)
Attempts to compile some files.

Parameters

- **files** – A list of paths to files to compile.
- **flags** – A list of command line arguments to supply to the compiler. Note that `-o main` will be added after your arguments.
- **ignore_cache** – If you ask Galah Interact to compile some files, it will cache the results. The next time you try to compile the same files, the executable that was cached will be used instead. Set this argument to `True` if you don't want the cache to be used.

Returns A `TestResult` object.

```
>>> print interact.standardtests.check_compiles(["main.cpp", "foo.cpp"])
Score: 0 out of 10

This test ensures that your code compiles without errors. Your program was␣
↪compiled with g++ -o main /tmp/main.cpp /tmp/foo.cpp.

Your code did not compile. The compiler outputted the following errors:

```
/tmp/main.cpp: In function 'int main()':
/tmp/main.cpp:7:9: error: 'foo' was not declared in this scope
/tmp/main.cpp:9:18: error: 'dothings' was not declared in this scope
/tmp/main.cpp:11:19: error: 'dootherthings' was not declared in this scope

```
```

interact.standardtests.**check_files_exist**(**files*)
Checks to see if the given files provided as arguments exist. They must be files as defined by os.path.isfile().

Parameters ***files** – The files to check for existance. Note this is not a list, rather you should pass in each file as a seperate arugment. See the examples below.

Returns Returns a TestResult object that will be passing iff *all* of the files exist.

```
# The current directory contains only a main.cpp file.
>>> print check_files_exist("main.cpp", "foo.cpp", "foo.h")
Score: 0 out of 1

This test ensures that all of the necessary files are present.

 * You are missing foo.cpp and foo.h.
```

(Note that this function really does return a TestResult object, but TestResult.__str__() which transforms the TestResult into a string that can be printed formats it specially as seen above)

`interact.standardtests.`**`check_indentation`**(*files*, *max_score=10*, *allow_negative=False*)

Checks to see if code is indented properly.

Currently code is indented properly iff every block of code is indented strictly more than its parent block.

Parameters

- **`files`** – A list of file paths that will each be opened and examined.
- **`max_score`** – For every improperly indented line of code, a point is taken off from the total score. The total score starts at `max_score`.
- **`allow_negative`** – If True, a negative total score will be possible, if False, 0 will be the lowest score possible.

Returns A `TestResult` object.

```
>>> print open("main.cpp").read()
#include <iostream>

using namespace std;

int main() {
    if (true) {
    foo();
    } else {
        dothings();
        while (false) {
    dootherthings();
        }
        cout << "{}{}{{{{{}}}}{}{}{}}}}}}}}{{{{"<< endl;
}
return 0;
}

>>> print open("foo.cpp").read()
#include <iostream>

using namespace std;

int main() {
return 0;
}

>>> print check_indentation(["main.cpp", "foo.cpp"])
Score: 6 out of 10

This test checks to ensure you are indenting properly. Make sure that every time␣
↪you start a new block (curly braces delimit blocks) you indent more.
```

```
* Lines 14, 13, and 10 in main.cpp are not indented more than the outer block.
* Line 5 in foo.cpp is not indented more than the outer block.
```

interact.unittest

This module contains very useful functions you can use while unittesting student's code.

Note: In order to use the `unittest` module, you need to make sure that you have SWIG installed, and that you have *Python development headers* installed, both of which are probably available through your distribution's package manager (`apt-get` or `yum` for example).

exception `interact.unittest.`**`CouldNotCompile`**(*message*, *stderr*)

Exception raised when a student's code could not be compiled into a single library file.

Variables

- **`message`** – A short message describing the exception.
- **`stderr`** – The output that was received through standard error. This is output by `distutils.core.setup`.

`interact.unittest.`**`load_files`**(*files*)

Compiles and loads functions and classes in code files and makes them callable from within Python.

Parameters **`files`** – A list of file paths. All of the files will be compiled and loaded together.

Returns A `dict` where every file that was passed in is a key in the dictionary (without its file extension) and the value is another `dict` where each key is the name of a function or class in the file and the value is a callable you can use to actually execute or create an instance of that function or class.

Raises `EnvironmentError` if swig is not properly installed.

Raises *`CouldNotCompile`* if the student's code could not be compiled into a library file.

Warning: During testing, oftentimes the execution of loaded code's `main()` function failed. We haven't determined what the problem is yet so for now don't use this function to test `main()` functions (the *`interact.execute`* module should work well instead).

```
>>> print open("main.cpp").read()
#include <iostream>

using namespace std;

class Foo {
    int a_;
public:
    Foo(int a);
    int get_a() const;
};

Foo::Foo(int a) : a_(a) {
    // Do nothing
}
```

```
int Foo::get_a() const {
    return a_;
}

int bar() {
    Foo foo(3);
    cout << "foo.get_a() = " << foo.get_a() << endl;
    return 2;
}

int main() {
    return 0;
}
>>> students_code = interact.unittest.load_files(["main.cpp"])
>>> Foo = students_code["main"]["Foo"]
>>> bar = students_code["main"]["bar"]
>>> b = Foo(3)
>>> b.get_a()
3
>>> rvalue = b.bar()
foo.get_a() = 3
>>> print rvalue
2
```

If you want to test a function that prints things to stdout or reads from stdin (like the `bar()` function in the above example) you can use the *interact.capture* module.

`interact.unittest.`**swig_path = None**

The absolute path to the swig executable. When this module is imported, the environmental variable `PATH` is searched for a file named `swig`, this variable will be set to the first one that is found. This variable will equal `None` if no such file could be found.

interact.capture

This module provides the tools for easily running a Python function in a seperate process in order to capture its standard input, output, and error.

class `interact.capture.`**CapturedFunction**(*pid*, *stdin_pipe*, *stdout_pipe*, *stderr_pipe*, *return-value_pipe*)

The type of object returned by *capture_function()*. Provides access to a captured function's stdin, stdout, stderr, and return value.

Variables

- **pid** – The process ID of the process that is running/ran the function.
- **stdin** – A file object (opened for writing) that the captured function is using as stdin.
- **stdout** – A file object (opened for reading) that the captured function is using as stdout.
- **stderr** – A file object (opened for reading) that the captured function is using as stderr.
- **return_value** – Whatever the function returned. Will not be set until *CapturedFunction.wait()* is called. Will contain the value *CapturedFunction.NOT_SET* if it has not been set by a call to *CapturedFunction.wait()*.

The correct way to check if `return_value` is set is to compare with *`CapturedFunction.NOT_SET`* like so:

```
if my_captured_function.return_value is CapturedFunction.NOT_SET:
    print "Not set yet!"
else:
    print "It's set!"
```

NOT_SET = <interact.capture._NotSet instance>
A sentinel value used to denote that a `return_value` has not been set yet.

wait()
Blocks until the process running the captured function exits (which will be when the function returns). Sets `return_value`.

If the function raised an exception, this function will raise that exception.

interact.capture.**capture_function**(*func*, **args*, ***kwargs*)
Executes a function and captures anything it prints to standard output or standard error, along with capturing its return value.

Parameters

- **func** – The function to execute and capture.
- ***args, **kwargs** – The arguments to pass to the function.

Returns An instance of *`CapturedFunction`*.

```
>>> def foo(x, c = 3):
...     print x, "likes", c
...     return x + c
>>> a = capture_function(foo, 2, c = 9)
>>> a.stdout.read()
"2 likes 9\n"
>>> a.wait()
>>> print a.return_value
11
```

Test Harness Command Line Interface

When you run your test harness from the command line, certain command line arguments have significance, and are parsed when *`interact.core.Harness.start()`* is called.

You can find a brief description of the command line arguments available by running your harness with the `--help` option (ex: `./my_harness.py --help`). This document attempts to be a more thorough description of behavior.

Basic Usage

When you run your harness (that you created using the Galah Interact library) you use the following flags to control the behavior of the harness.

-m, --mode *MODE* Use this flag to set the execution mode of the harness. The default mode is `galah`. See *Execution Modes* for more information.

-s, --set-value *KEY VALUE* Use this flag (multiple times if desired) to set "configuration" values when in `test` mode. See *Configuration Values* for more information.

Examples

If we want to start a harness in `test` mode and let the harness guess all of the values...

```
./my_harness.py --mode test
```

If we want to start a harness in `test` mode and set the testables directory (where the student's code is) to another directory...

```
./my_harness.py --mode test --set-value testables_directory ./student1/
```

If we want to start a harness in `test` mode and set the testables directory along with providing a fake submission object...

```
./my_harness.py --mode test --set-value testables_directory ./student1/ --set-value␣
↪raw_submission "{'id': 'junk', 'user': 'john'}"
```

Execution Modes

The mode of execution (set with the `--mode` option) determines how the test harness will gather the information it needs to run, and how it outputs the results once they are available.

`galah` mode

In `galah` mode, when the test harness starts it will try to read in JSON from standard input, and when it finishes the output will be sent to standard output as JSON. This is not a very human-friendly process and generally you don't want to set it to this mode during development of a test harness.

If you harness seems like it's not responding when you start it, it's probably running in this mode and is waiting for an entire JSON object to be placed into stdin.

This mode exists for when the test harness is being run within Galah.

`test` mode

In `test` mode, the test harness will try to guess any values it needs (and you can specify values explictly using `--set-value`) and then print out the results in a human-friendly way. You should always use this mode during the development of your test harness.

Configuration Values

Test harnesses need certain information in order to function properly. In galah mode, you must specify them all with JSON, but in test mode the harness will attempt to guess some values, and you can override any of the guesses by using the `--set-value` command line argument. If you use `--set-value` with any argument that expects a list or dictionary, the value you set will be assumed to be JSON and will be deserialized.

These values are available in *`interact.core.Harness.sheep_data`* from within the test harness.

Note: The below keys are case-sensitive.

KEY (*DEFAULT VALUE*): DESCRIPTION

testables_directory (*current directory*): The directory that contains the student's code.

harness_directory (*directory of the running test harness*): The directory that contains the test harness itself.

raw_submission (*None*): A submission object with meta data on the student's submission. In Galah, this is a dictionary with at least the following fields:

```
{
    "id": "ID of submission in database",
    "assignment": "ID of assignment in database",
    "user": "username of student",
    "timestamp": "submission time in ISO format"
}
```

If specified with –set-value, the string supplied will be assumed to be a valid JSON object and will be deserialized.

raw_assignment (*None*): An assignment object with meta data on the assignment this harness is attached to. In Galah, this is a dictionary with at least the following fields:

```
{
    "name": "The name of the assignment",
    "due": "due date in ISO format",
    "due_cutoff": "cutoff date in ISO format",
    "hide_until": "hide until field in ISO format"
}
```

raw_harness (*None*): A harness object with meta data about the test harness. In Galah, this is a dictionary with at least the following fields:

```
{
    "config": "dictionary supplied when harness was uploaded",
    "id": "ID of harness in database"
}
```

actions (*instance of* `interact.core.UniverseSet`): A list of actions that the test harness should perform. This is not yet fully supported within Galah. As such there's not full support for it in Galah Interact yet.

CHAPTER 4

Quick Sample

To give you an idea of what using this library feels like, below is a simple but fairly complete test harness for an assignment where the students must make a `foo()` function that takes in two numbers and returns their sum.

`teaser.py`

```
#!/usr/bin/env python

import interact

harness = interact.Harness()
harness.start()

student_files = harness.student_files("main.cpp")

@harness.test("Proper files exist.")
def check_files():
  return interact.standardtests.check_files_exist(*student_files)

@harness.test("Program compiles correctly.", depends = [check_files])
def check_compilation():
  return interact.standardtests.check_compiles(student_files)

@harness.test("Proper indentation is used.", depends = [check_files])
def check_indentation():
  return interact.standardtests.check_indentation(student_files)

@harness.test("foo function works correctly.", depends = [check_compilation])
def check_foo():
  result = interact.TestResult(
    brief = "This test ensures that your `foo` function correctly takes in two "
            "integer parameters and returns their sum.",
    default_message = "**Great job!** Your function works great!",
    max_score = 10
  )
```

```
  student_code = interact.unittest.load_files(student_files)

  if "foo" not in student_code["main"]:
    result.add_message("You didn't create a `foo` function.", dscore = -10)
    return result.calculate_score()

  foo = student_code["main"]["foo"]
  try:
    if foo(3, 4) != 7:
      result.add_message("Your function did not give the correct sum.")
  except TypeError:
    result.add_message("Your function does not have the correct signature.")

  return result.calculate_score(min_score = 0)

harness.run_tests()

harness.finish(max_score = 31)
```

When running this harness with `./teaser.py --mode test`, and a correctly implemented `main.cpp` (`example here`), the following is output:

```
Score: 1 out of 1

This test ensures that all of the necessary files are present.

Great job! All the necessary files are present.
-------
Score: 10 out of 10

This test ensures that your code compiles without errors. Your program was compiled␣
↪with g++ -o main /home/john/Projects/galah-interact-python/docs/guides/examples/
↪teaser/main.cpp.

**Great job!** Your code compiled cleanly without any problems.
-------
Score: 10 out of 10

This test checks to ensure you are indenting properly. Make sure that every time you␣
↪start a new block (curly braces delimit blocks) you indent more.

**Great job!** We didn't find any problems with your indentation!
-------
Score: 10 out of 10

This test ensures that your `foo` function correctly takes in two integer parameters␣
↪and returns their sum.

**Great job!** Your function works great!
-------
Final result: 31 out of 31
```

CHAPTER 5

Licensing

The code is licensed under the Apache 2.0 license, which is a very permissive license. You can read a summary of its specific terms on the wikipedia page for the license here. The entire license text is also contained within this repository if you'd like to read the license itself.

In short, the license is very permissive and lets you do basically whatever you want with the code as long as you properly attribute the contributers. So as long as you don't rip out the code and say you wrote it, your probably staying within the terms of the license.

Note that this license is very different than the license that covers Galah itself. Please don't confuse the two.

Python Module Index

i

Index

R

S

T

U

W