CheckPy Documentation

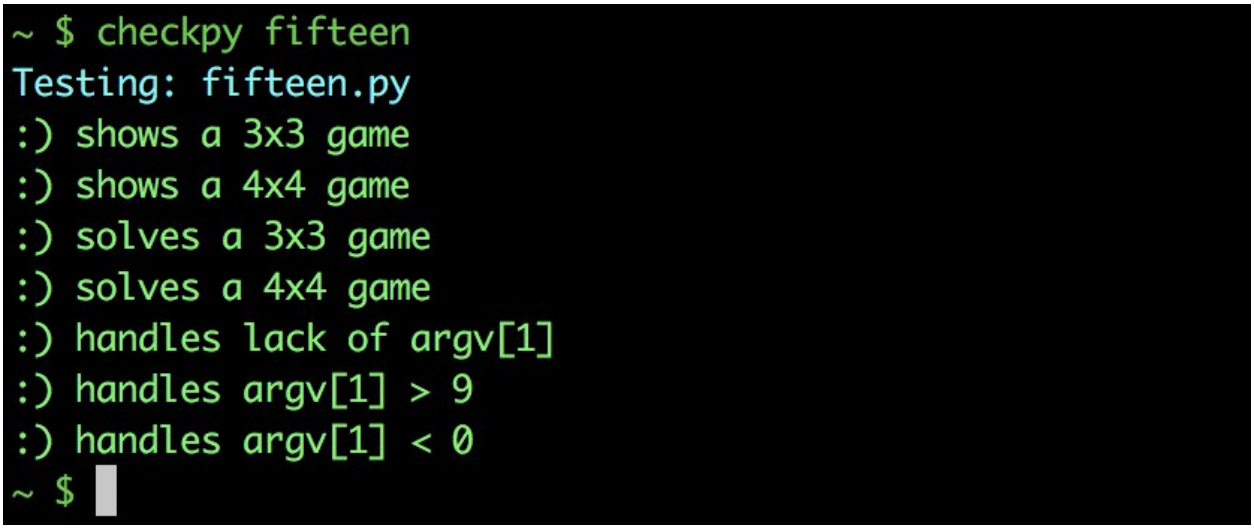
Release 0.4

Jelle van Assema (JelleAs)

Sep 20, 2023

Contents:

An education oriented testing framework for Python. Developed for courses in the Minor Programming at the University of Amsterdam.



CHAPTER 1

Introduction to CheckPy

1.1 Installation

```
pip install checkpy
```

1.2 Writing and running your first test

First create a new directory to store your tests somewhere on your computer. Then navigate to that directory, and create a new file called `helloTest.py`. CheckPy discovers tests for a particular source file (i.e. `hello.py`) by looking for a test file starting with a corresponding name (`hello`) and ending with `test.py`. So CheckPy uses `helloTest.py` to test `hello.py`.

Now open up `helloTest.py` and insert the following code:

```
import checkpy.tests as t
import checkpy.lib as lib
import checkpy.assertlib as asserts

@t.test(0)
def exactlyHelloWorld(test):
    def testMethod():
        output = lib.outputOf(test.fileName)
        return asserts.exact(output.strip(), "Hello, world!")

    test.test = testMethod
    test.description = lambda : "prints exactly: Hello, world!"
```

Next, create a file called `hello.py` somewhere on your computer. Insert the following snippet of code in `hello.py`:

```
print("Hello, world!")
```

Now there's only one thing left to do. We need to tell CheckPy where the tests are located. You can do this by calling CheckPy with the *-register* flag and by providing an absolute path to the directory `helloTest.py` is located in. Say `helloTest.py` is located in `\Users\foo\bar\tests\` then call CheckPy like so:

```
checkpy -register \Users\foo\bar\tests\
```

Alright, we're there. We got a test (`helloTest.py`), a Python file we want to test (`hello.py`), and we've told CheckPy where to look for tests. Now navigate over to the directory that contains `hello.py` and call CheckPy as follows:

```
checkpy hello
```

1.3 Writing simple tests in CheckPy

Tests in CheckPy are instances of `checkpy.tests.Test`. These `Test` instances have several abstract methods that you can implement or rather, overwrite by binding a new method. These methods are executed when CheckPy runs a test. For instance you have the `description` method which is called to produce a description for the test, the `timeout` method which is called to determine the maximum alotted time for this test, and ofcourse the `test` method which is called to actually perform the test. This `Test` instance is automatically provided to you when you decorate a function with the `checkpy.tests.test` decorator. In our hello-world example this looked something like:

```
@t.test(0)
def exactlyHelloWorld(test):
```

Here the `t.test` decorator (`t` is short for `checkpy.tests`) decorates the function `exactlyHelloWorld`. This causes CheckPy to treat `exactlyHelloWorld` as a *test creator* function. That when called produces an instance of `Test`. The `t.test` decorator accepts an argument that is used to determine the order in which the result of the test is shown to the screen (lowest first). The decorator then passes an instance of `Test` to the decorated function (`exactlyHelloWorld`). It is up to `exactlyHelloWorld` to overwrite some or all abstract methods of that one instance of `Test` that it receives.

Lets start with the necessities. CheckPy requires you to overwrite two methods from every `Test` instance. These methods are `test` and `description`. The `description` method should produce the description, that can be just a string, for the user to see. In our hello-world example we used this `description` method:

```
test.description = lambda : "prints exactly: Hello, world!"
```

Depending on whether the test fails or passes, the user sees this string in red or green respectively. The other method we have to overwrite, the `test` method, should return `True` or `False` depending on whether the tests passes or fails. You are free to implement this method in any which way you want. CheckPy just offers some useful tools to make your testing life easier. Again, looking back at our hello-world example, we used this `test` method:

```
def testMethod():
    output = lib.outputOf(test.fileName)
    return asserts.exact(output.strip(), "Hello, world!")

test.test = testMethod
```

So what's going on here? Python doesn't support multi statement lambda functions. This means that if you want to use multiple statements, you have to resort to named functions, i.e. `testMethod()`, and then bind this named function to the respective method of the `Test` instance. You can put the above test method in a single statement lambda function, but readability will suffer from it. Especially once we move on to some more complex test methods.

Now there are just 2 lines of code in this `testMethod`. First we take the output of something called `test.fileName`. `test.fileName` just refers to the name of the file the user wants to test, CheckPy will automatically

set this for you. `lib.outputOf` is a function that gives you all the print-output of a python file. Thus what this line of code does is simply run the file that the user wants to test, and then return the print output as a string.

The line `return asserts.exact(output.strip(), "Hello, world!")` is equivalent to `return output.strip() == "Hello, world!"`. The `checkpy.assertlib` module, that is renamed to `asserts` here, simply offers a collection of functions to perform assertions. These functions do nothing more than return `True` or `False`.

That's all there is to it. You simply write a function and decorate it with the *test* decorator. Overwrite a couple of methods of `Test`, and you're good to go.

1.4 Testing functions

Let's make life a little more exciting. CheckPy can do a lot more besides simply running a Python file and looking at print output. Specifically CheckPy lets you import said Python file as a module and do all sort of things with it and to it. Lets focus on Functions for now.

For an assignment on (biological) virus simulations, we asked students to do the following:

Write a function `generateVirus(length)`. This function should accept one argument `length`, that is an integer representing the length of the virus. The function should return a virus, that is a string of random nucleotides (`'A'`, `'T'`, `'G'` or `'C'`).

This is just a small part of a bigger assignment that ultimately moves towards a simulation of viruses in a patient. We can use CheckPy to test several aspects of this assignment. For instance to test whether only the nucleotides ATGC occurred we wrote the following:

```
@t.test(10)
def onlyATGC(test):
    def testMethod():
            generateVirus = lib.getFunction("generateVirus", test.fileName)
            pairs = "".join(generateVirus(10) for _ in range(1000))
            return asserts.containsOnly(pairs, "AGTC")

    test.test = testMethod
    test.description = lambda : "generateVirus produces viruses consisting only of A,␣
↪T, G and C"
```

To test whether the function actually exists and accepted just one argument, we wrote the following:

```
@t.test(0)
def isDefined(test):
  def testMethod():
    generateVirus = lib.getFunction("generateVirus", test.fileName)
    return len(generateVirus.arguments) == 1

  test.test = testMethod
  test.description = lambda : "generateVirus is defined and accepts just one argument"
```

1.5 Testing programs with arguments

Taking a closer look at the `checkpy.lib` module we find three functions that allow you to interact with dynamic components and results from the program we are testing. All these functions (`outputOf`, `getFunction` and `getModule`) take in the same optional arguments that let you change the dynamic environment in which the code is tested. Zooming in on `outputOf`:

```
def outputOf(
            fileName,
            src = None,
            argv = None,
            stdinArgs = None,
            ignoreExceptions = (),
            overwriteAttributes = ()
    ):
  """
  fileName is the file name you want the stdout output of
  src can be used to ignore the source code of fileName, and instead use this string
  argv is a collection of elements that are used to overwrite sys.argv
  stdinArgs is a collection of arguments that are passed to stdin
  ignoreExceptions is a collection of exceptions that should be ignored during␣
↪execution
  overwriteAttributes is a collection of tuples (attribute, value) that are␣
↪overwritten
      before trying to import the file
  """
```

Lets see what we can do with this. For an assignment we asked students to write a program that prints out how many liters of water were used while showering. The program should prompt the user for the number of minutes they shower, and then print out many liters of water were used. We told them 1 minute of showering equaled 12 liters used.

For this assignment we wrote the following test:

```
@t.test(10)
def oneLiter(test):
  def testMethod():
    output = lib.outputOf(
        test.fileName,
        stdinArgs = [1],
        overwriteAttributes = [("__name__", "__main__")]
    )
    return asserts.contains(output, "12")

  test.test = testMethod
  test.description = lambda : "1 minute equals 12 bottles."
```

The above test runs the student's file, pushes the number 1 in stdin and sets the attribute `__name__` to `"__main__"`. It does not ignore any exceptions, that means that CheckPy will fail the test if an exception is raised and kindly tell the user what exception was raised. Argv is set to the default (just the program name).

1.6 Customizing output

An instance of `Test` has a couple of methods that you can use to show the user exactly what you want the user to see. We have already seen the `.description()` method that you can overwrite with a function that should produce the description of the test. This description then turns green or red, with a happy or sad smiley depending on whether the test fails or passes. Besides the `.description()` method you also find the `.success(info)` and `.fail(info)` methods. These methods take in an argument called `info` and should produce a message for the user to read when the test succeeds or fails respectively. This message is printed directly under the description. Take the following test:

```
@t.test(0)
def failExample1(test):
```

(continues on next page)

(continued from previous page)

```
  def testMethod():
    output = lib.outputOf(test.fileName)
    line = lib.getLine(output, 0)
    return asserts.numberOnLine(42, line)

  test.test = testMethod
  test.description = lambda : "demonstrating the use of .fail()!"
  test.fail = lambda info : "could not find 42 in the first line of the output"
```

The above test looks for the number 42 on the first line of the output. If the test fails it will print that it could not find 42 in the output. Okay, this is a little boring, CheckPy just prints a static description if the test fails. So let's spice things up. Take the following test:

```
@t.test(0)
def failExample2(test):
  def testMethod():
    output = lib.outputOf(test.fileName)
    line = lib.getLine(output, 0)
    return asserts.numberOnLine(42, line), lib.getNumbersFromString(line)

  test.test = testMethod
  test.description = lambda : "demonstrating the use of .fail()!"
  test.fail = lambda info : "could not find 42 on the first line of output, only␣
↪these numbers: {}".format(info)
```

This test also looks for 42 on the first line of the output. If this test fails however it will also print what numbers it did find on that one line of output. Here's what's happening. The `.test()` method can return a second value besides simply a boolean indicating whether the passed. This value is passed to the `.fail(info)` and `.success(info)` methods. So you can use this second return value to customize what `.fail(info)` and `.success(info)` do. Here the implementation of `.fail(info)` simply prints out whatever `info` it receives.

CHAPTER 2

Installation

```
pip install checkpy
```

CHAPTER 3

Features

- Customizable output, you choose what the users see.
- Support for blackbox and whitebox testing.
- The full scope of Python is available when designing tests.
- Automatic test distribution, CheckPy will keep its tests up to date by periodically checking for new tests.
- No infinite loops! CheckPy kills tests after a predefined timeout.

CHAPTER 4

Usage

```
usage: checkpy [-h] [-module MODULE] [-download GITHUBLINK]
               [-register LOCALLINK] [-update] [-list] [-clean] [--dev]
               [file]

checkPy: a python testing framework for education. You are running Python
version 3.6.2 and checkpy version 0.4.0.

positional arguments:
  file                  name of file to be tested

optional arguments:
  -h, --help            show this help message and exit
  -module MODULE        provide a module name or path to run all tests from
                        the module, or target a module for a specific test
  -download GITHUBLINK  download tests from a Github repository and exit
  -register LOCALLINK   register a local folder that contains tests and exit
  -update               update all downloaded tests and exit
  -list                 list all download locations and exit
  -clean                remove all tests from the tests folder and exit
  --dev                 get extra information to support the development of
                        tests
```