MyPyTutor: an interactive tutorial system for Python

Peter J. Robinson

School of Information Technology and Electrical Engineering,
The University of Queensland,
Brisbane, Australia,
Email: pjr@itee.uq.edu.au

Abstract

MyPyTutor is an interactive tutorial system for the Python programming language. The system provides support for both students doing tutorial problems and tutorial developers. MyPyTutor can be used in two modes: stand-alone or with online support. For courses where MyPyTutor is used for assessment it can be set up so that students submit correct solutions to problems online for marks.

Keywords: Automatic marking, interactive tutorials, Python, code analysis

1 Introduction

For some years we have been using xTutor from MIT (Ehrmann et al., 2006) for interactive tutorials, first for Scheme and more recently for Python. One of the disadvantages of xTutor is that the student enters Python code into a text box that provided little support for the kinds of support expected in an IDE such as syntax highlighting, layout and error feedback. It is also difficult to provide questions about GUIs where the student can see the results of their code. Also, students need to be registered, i.e. have a username and password, in order to use xTutor and students need to be logged on to the system in order to do the tutorial problems. On the other hand xTutor supports several kinds of tutorial questions other than coding questions such as multiple choice questions.

Python installations come with Tkinter (a GUI library based on Tcl/Tk) and an IDE written in Tkinter called IDLE. Most of our students, particularly Windows users, use IDLE for writing and testing their code. Furthermore, we use IDLE in lectures when writing code. One of the key design decisions for MyPyTutor is to provide, as much as possible, a familiar interface for students. Consequently, MyPyTutor is written in Tkinter and uses the same code edit window as is used in IDLE. Further, various kinds of errors, such as syntax errors, produce the same error highlighting in the edit window as when using IDLE. Also, by using Tkinter we are able to write tutorial questions about GUIs (written in Tkinter) where, as part of checking the code, the result of running the code can appear as a window, giving the student visual feedback.

Unlike xTutor, where everything is run on a server, MyPyTutor is an application that students download to their machines. The collection of tutorial problems are also downloaded. This means that MyPyTutor can be run in a stand-alone mode if it is simply used as a resource for students, or if a student wants to work on problems without being connected. If MyPyTutor is used for assessment then it can be set up so that students can log on and submit answers to problems.

MyPyTutor supports only coding problems, i.e. students write their answer as code and the system checks their code for correctness.

In order to provide more feedback to students other than just telling them if they are right or wrong, we often use parsing of student code to either give more information about where the student went wrong or possibly provide suggestions on better ways of solving the problem. These techniques are not specific to MyPyTutor but could be used in any interactive system in any language for which examination of the parse tree is easy to do.

As discussed in Douce et al. (2005), most modern automatic systems for assessing student programs are web based (like xTutor) and so typically have similar advantages and disadvantages as discussed above for xTutor. Douce et al. point out that security is an issue for automatic marking systems. In particular, malicious or badly behaving student code can be problematic. This can be avoided by, for example, sandboxing the test code (as is done in xTutor). MyPyTutor does not suffer from this problem as all test code is run on the student machine.

MyPyTutor is currently being used for very introductory problems and so the testing, apart from analysis of the parse tree, is quite straightforward. In principle, though, it would be possible to write more sophisticated testing code as used in other systems such as AutoGrader (Helmick, 2007) and Web-CAT (Edwards, 2003).

In the next section we describe the student view of MyPyTutor. In Section 3 we describe the developers and course administrators view. In Section 4 we discuss the design of tutorial problems and in Section 5 we consider testing using parsing. In Section 6 we describe the implementation and we conclude with some remarks.

2 Student View

MyPyTutor consists of two windows. One window is for editing code and is the same as the IDLE edit window apart from some minor changes to the menus. The other window consists of two text widgets. The top one displays the problem being worked on as a rendered HTML document. The bottom text widget is used to output the results of testing the student code.

It is used to display a "Correct" message if the code successfully passed the tests or some kind of error or warning message otherwise. Any output from

student code will also be displayed in this widget. The Problems menu allows the student to choose a problem to work on. If MyPyTutor is set up for online use, the Online menu is present and allows the user to log on and off, change their password, submit the current (correct) problem answer, upload/download their (partial) solution to the problem, and view the submission status of the problems.

Each student can configure MyPyTutor by choosing fonts and the folders where the tutorial problems and the student answers for each problem is to be found. MyPyTutor supports multiple problem sets for students doing more that one course using MyPyTutor.

Typically, students start up the application and choose a problem to work on. If they have previously saved their code for the problem then the code will be automatically loaded from the answers folder on their machine. They might wish to overwrite this by downloading a previously uploaded version of their code (if they are logged on).

After working on the code they can check their code for correctness. If the code passes the test, and they are logged on, they can submit their answer to the server for marks. If the code fails the test they can edit their code and check again.

When students checks correct code, MyPyTutor will respond with a message saying that the code is correct, and if MyPyTutor is set up for answer submission then it will also remind the students to submit their answers. If the code is incorrect, MyPyTutor responds by displaying an error message of some kind. All exceptions produced by student code are displayed and exceptions such as syntax errors or name errors will cause the offending line of code to be highlighted. The other error messages displayed are based on the results of testing the code. This could simply be a message saying that the code is producing the wrong value and what the correct value should be. It could be an error message that gives more feedback on where the student has gone wrong based on, for example, the use of parsing (see later). In some cases the system might provide some feedback as a warning and then continue testing. This typically occurs when the analysis of the student code suggests that the student may be, for example, using the wrong test for termination of recursion. In this situation it may be possible that the student code is correct but not using the expected test.

When MyPyTutor is set up for online use (and for marks), due dates are associated with each section in the Tutorials menu. If the student submits after the due date, the problem is marked as late and is not counted towards marks.

Figures 1 and 2 given an example of MyPyTutor in action. The user has chosen a problem to work on and has pressed the hint button to reveal the hint. The user has added the last line in the edit window (the other lines are pre-loaded). The user has checked the code and the output shows the result of checking the code. When the user corrects this problem by replacing the print with a return then, on checking again, the output window will display an error message such as "Wrong: for input 7 the correct result is 49, you got 14". This problem is used as an example in Section 5.

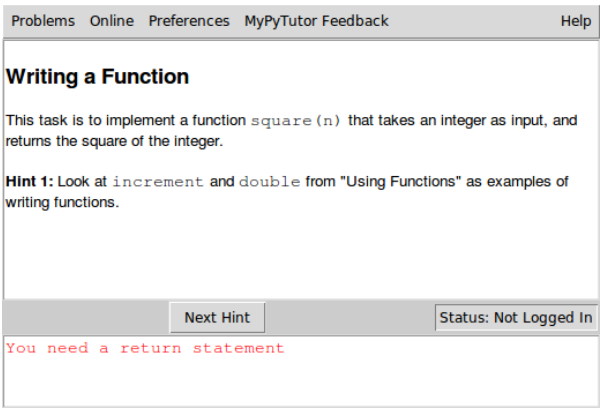

Figure 1: MyPyTutor Example: Problem Description and Output Window

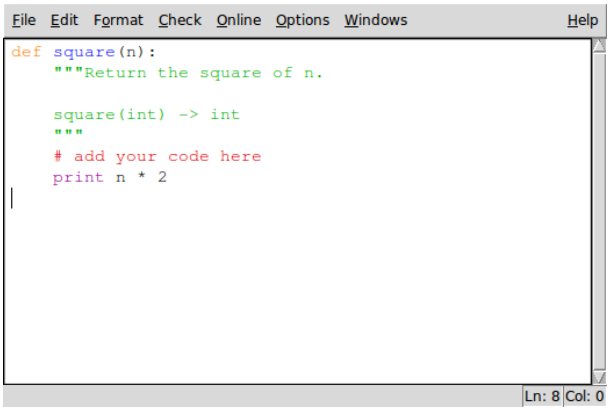

Figure 2: MyPyTutor Example: Edit Window

3 Developer and Administrator View

The developer is the person who creates individual tutorial problems and collects some or all of these problems into a tutorial set. The administrator is the person running a course where MyPyTutor is used for assessment. The MyPyTutor system comes with several tools to support the developer and administrator. All these tools are written in Tkinter and share some of the MyPyTutor code. The developer has the use of three tools: a problem creator tool, a problem checking tool, and a tutorial creator tool.

The problem creator tool is used to construct tutorial problems. It has two edit windows. One window is where the problem description is written in HTML source. The other window is where the testing code is written.

The problem testing tool is a cut-down version of MyPyTutor that the developer can use to test to see if the problem he/she has written behaves correctly from a student point of view. These two tools are designed to work together, making it easy to edit and check problems.

The tutorial creator tool is used by the developer to create a collection of tutorial problems to be used in a course. This consists of an editor window in which the developer can list the tutorial sections and each problem within each section together with any auxiliary files (e.g. GIF files for images in the HTML). If the system is intended for online use, the first line is the URL of the MyPyTutor server and due dates are added to the section headers.

When the developer has finished designing the tutorial set, the tutorial set can then be exported for use by the student. This consists of copying all the required files to the designated folder, encrypting the test code part of each problem, and zipping the contents of the folder.

The system has been designed in such a way that, if it is used in online mode, any modifications to the problems can be uploaded to the server and when a student next logs on, the updated problem set will be downloaded to replace the old version. This means

that any errors found can be corrected and an update problem set distributed to students is a straightforward manner.

The administrator tool is used to manage student information when MyPyTutor is used in online mode. The administrator typically starts by registering students (i.e. create usernames and passwords) individually or from a class list in a particular format. At any time, the administrator can change the password of a student. The administrator can search for matching patterns in the user file (e.g. search for family name or username).

As was discussed earlier, if a student submits a problem late then no marks are awarded for that problem. The administrator has the ability to unset the late flag on a problem so that it will be counted.

The administrator can also extract results from the server. At the end of the course (or indeed at any time) the administrator can collect the marks for all the students and can also obtain information on the percentage of the class that have submitted each question. As a very rough guide to determining how hard students found a particular problem the average number of times a given problem was checked in the current session can be displayed. Although this is a very rough estimate of difficulty, it seems to provide a good guide in practice. This information might be used by the problem designer, for example, to reword the problem description or add extra hints for problems students appear to be having problems with.

4 Problem Design

Writing the problem description is reasonably straightforward. It is written in HTML source with a restricted number of tags. Images in GIF format can be embedded in the problem description. It's also possible to add hints. If a problem has one or more hints then a Hints button appears and each time the button is pressed the next hint is displayed in the problem description widget. Hints are also written in restricted HTML.

The other component of problem design is the test code. Any number of different tests can be run on the student code and any test can be repeated a number of times (for example, if random numbers are used to generate values). Tests can be straightforward, for example, simply comparing the answer produced by student code with the expected answer. The tests can also be quite sophisticated. For example, the student code can be parsed and tests carried out based on the structure of the code (see later).

For the introductory course in which MyPyTutor is currently being used, the problems are all very straightforward and so, if simply testing the answer is all that is required, the test code could be very simple. However, experience with using xTutor suggests that simply giving the results of the tests to beginner students is not very helpful. In response, many of the problems use parsing to provide better feedback. As students and problems become more sophisticated, parsing is probably less necessary and so the testing code would typically be much simpler.

An example of test code is given in Figure 3. The problem asks the student to assign to the variable `prod` the product of the variables `item1` and `item2`.

The comments such as `#{global}#` are headers for the different blocks of the test code. The main part of the testing code is the test blocks (with the `#{test}#` header). There needs to be at least one of these blocks. When the student code is checked, the code of each test block is run until either an error occurs or all tests have completed. There are three blocks that may appear before the first test block.

```
#{global}#
import random
#{test}#
#{start}#
item1_save = random.randint(2,100)
item2_save = random.randint(2,100)
#{init}#
item1 = item1_save
item2 = item2_save

#{code}#
import sys
if prod == item1_save*item2_save:
   correct()
else:
   print_error("Wrong - You got %s \
the correct result is %d \n\
The right hand of the assignment should be \
an expression involving item1 and item2" \
% (str(prod), item1_save*item2_save)
```

Figure 3: Test Code Example

The `#{global}#` block contains code that is common to all tests. In this case it imports the random module but it could be, for example, a class definition to be used in testing or some function definitions students are expected to use. The `#{preload}#` block (not used in this example) contains code that is copied into the edit window when this problem is chosen. It might, for example, be a partially defined function that the student needs to complete (Figure 2 shows an example). MyPyTutor uses a timeout (default one second) to stop runaway student code after the given time. For some problems this may not be enough time. The block of the form `#{timeout = secs}#` (with no body) allows the designer to specify the timeout for running the tests.

Each test block can contain three subblocks: the start block, the init block, and the code block. The first two are optional. When a test is executed, the code in the start block is first executed in an environment that cannot be accessed when running the student code. Then the init code is run in environment that has access to the start environment. Then the student code is executed followed by the code in the code block. The reason for this rather complicated setup is to prevent, in this example, the student code

```
prod, item1, item2 = 0, 0, 0
```

from being a solution.

By keeping a copy of the values of `item1` and `item2` in an environment that can't be accessed by the student code, then the proper test can be carried out. This is discussed in more detail in Section 6.

When using random numbers for testing it is often necessary to run a given test multiple times in order to be reasonably confident the student code is correct. This can be done by using the header `#{test repeats = 3}#`, say, rather than `#{test}#`.

Exception handling is an important issue when testing student code. To hide details of the test code all exceptions produced by the testing code (that are not explicitly handled in the test code) are trapped and the error message `'Error: report to maintainer'` is output. This should be avoided at all costs as it gives no information to the student about the cause of the problem. It is therefore important to structure the test code so this does not happen.

Below is a fragment of test code for a problem that calls a function `f` defined by the student.

```
try:
```

```
    result = f(test_arg)
    ok = True
except Exception, e:
    print_exception(e)
    ok = False

if not ok:
    pass
elif result == .....
```

In this example, if no exception is thrown by the student code then the result is tested for correctness. If, however, an exception is thrown the exception is passed to a MyPyTutor function that displays the exception and further testing is short-circuited.

Now consider an example where an integer value is expected as the result of calling a student defined function (as in the above example). It is tempting to write test code as follows

```
if result != 42:
    print_error("Wrong: you got %d \
the correct answer is 42" % result)
```

where `print_error` is a MyPyTutor function that displays the message. The problem is that, if the student returns something other than an integer (`None`, for example), then an exception will be thrown by the formatting. This can be avoided by using either `str` or `repr` as follows.

```
if result != 42:
    print_error("Wrong: you got %s \
the correct answer is 42" % repr(result))
```

Another place where exceptions can crop up is in walking the parse tree of the student code. If the problem designer assumes too much about the shape of the student code then it's easy to produce an exception if the student writes "unexpected" code. So, although parsing student code can provide excellent feedback, the parsing code must be carefully crafted to avoid this problem.

5 Parsing Student Code

Parsing is a powerful tool for finding certain kinds of errors and for providing more detailed feedback to students than simply saying what the student code produced and what the correct answer was. The ideas presented below are not specific to MyPyTutor or to Python.

In the following example we present fragments of a relatively straightforward example using parsing. The tutorial problem is to write a square function that takes an integer and returns its square. This is an introductory example and as such the student might be having trouble with return statements, the use of arithmetic operators and the use of formal parameters to functions. By walking the parse tree of the student code we can check if this is being done correctly or provide some feedback.

Python comes with a module called `compiler` that has a function for constructing the abstract syntax tree (AST) of the student code. It also has a function that takes an AST and a code visitor object (defined by the problem designer) and visits nodes according to the methods defined in the code visitor class. An example of such a class is given in Figure 4.

The code for using the parse tree is given below.

```
ast = compiler.parse(user_text)
visitor = CodeVisitor()
compiler.walk(ast, visitor)
if not visitor.has_return:
```

```
class CodeVisitor:
    def __init__(self):
        self.arg1 = None
        self.in_defn = False
        self.has_return = False
        self.use_mult = False
        self.use_power = False
        self.in_mult = False
        self.in_power = False
        self.use_arg1 = False

    def visitFunction(self,t):
        if t.name == 'square':
            self.in_defn = True
            if len(t.argnames) == 1:
                self.arg1 = t.argnames[0]
            for n in t.getChildNodes():
                compiler.walk(n, self)
            self.in_defn = False

    def visitReturn(self,t):
        if self.in_defn:
            self.has_return = True
            for n in t.getChildNodes():
                compiler.walk(n, self)

    def visitPower(self,t):
        if self.in_defn:
            self.use_power = True
            self.in_power = True
            for n in t.getChildNodes():
                compiler.walk(n, self)
            self.in_power = False

    def visitMul(self,t):
        if self.in_defn:
            self.use_mult = True
            self.in_mult = True
            for n in t.getChildNodes():
                compiler.walk(n, self)
            self.in_mult = False

    def visitName(self,t):
        if self.in_defn and \
                (self.in_power or \
                     self.in_mult):
            self.use_arg1 = \
                self.arg1 == t.name
```

Figure 4: Code Visitor Class

```
    print_error('You need a return \
statement')
elif not (visitor.use_mult or \
              visitor.use_power):
    print_error('You should use either \
* or **')
elif not visitor.use_arg1:
    print_warning('Are you using the \
variable %s in the body of the \
definition?'% visitor.arg1)
```

By doing this we give feedback if, for example, a return statement is not used. This is a common mistake by beginners who often use a print statement rather than a return statement.

As another example, if the question asked the student to write a for loop to solve a problem but the student produced a correct solution using a while loop then straightforward testing is not enough to detect the non-use of a for loop. By parsing we can give feedback that a for loop should be used.

6 Implementation

MyPyTutor is implemented in Python using the Tkinter GUI library. One important aspect of the implementation of MyPyTutor relates to the fact that the application itself and the tutorial problems are loaded on to the student machine. This differs from server or web based systems like xTutor where all the code lives on the server. Because the xTutor code is on the server, the student does not have access to the implementation of xTutor or the problem testing code. In MyPyTutor, however, the student has all the code on their machine. Consequently, attempts to hide the application code and the problem testing code has been made. Firstly, apart from the top-level interface to the MyPyTutor application, the support code is compiled to `pyc` files. We believe this makes it sufficiently difficult for students to decompile the support code in order to understand how the code works.

In order to hide the problem testing code, when the tutorial developer exports the set of tutorial problems, the testing code for each problem is encrypted using a moderately strong and fast algorithm. When a student chooses to work on a given problem MyPyTutor renders the HTML problem description and decrypts the testing code ready for use.

Admittedly this is "security by obfuscation" but we feel it is good enough to hide the code from all but the most talented students.

The MyPyTutor code edit window inherits from the IDLE code edit window. By doing this, students use the same editor for doing MyPyTutor problems as they do when doing their assignments or other Python coding. Furthermore, MyPyTutor traps exceptions such as Syntax errors, extracts the line number for the error and highlights that line in the same way as is done in IDLE.

For rendering the HTML for the problem description, the HTMLParser module is used. We currently support the following tags.

```
h1 h2 h3 h4 h5
b it tt
br p ul li
pre
img
span
```

The `img` tag is of the form `img src="filename.gif"` - i.e. the image must be a GIF file. Currently, only `span` with a colour style of `red`, `green` or `blue` is supported.

The key part of the implementation of MyPyTutor is the testing of student code. When the testing code is run on the student code, first a global environment is created.

```
global_env = \
    {'user_text': self.user_text,
     'print_warning':self.print_warning,
     'print_error':self.print_error,
     'print_exception':self.print_exception,
     'correct':self.correct,
     'master':self.master}
```

This environment provides the testing code with access to MyPyTutor code: `user_text` is the student code; `print_warning` displays a warning in the output text widget; `print_error` and `print_exception` display messages and set a flag that will terminate testing; `correct` displays a message that the code is correct and sets a flag to signal this; and `master` provides a mechanism for students to write GUI code that will be displayed in a window.

```
 1: test_globals = global_env.copy()
 2: try:
 3:     exec test.get('start', '') in \
 4:         test_globals
 5: except Exception, e:
 6:     raise TestError(e)
 7: save_globals = test_globals.copy()
 8: locs = {}
 9: try:
10:     exec test.get('init', '') in \
11:         test_globals,locs
12: except Exception, e:
13:     raise TestError(str(e))
14: exec self.user_text in locs
15: test_globals.update(locs)
16: test_globals.update(save_globals)
17: try:
18:     exec test.get('code', '') in \
19:         test_globals
20: except NameError,e:
21:     raise(NameError(e))
22: except Exception, e:
23:     raise TestError(str(e))
```

Figure 5: MyPyTutor Testing Code

The core part of the testing code is listed in Figure 5. First the global environment is copied into another environment that will be modified during testing. On lines 3 and 4 the start code (if present) is evaluated in the `test_globals` environment, updating that environment. That environment is then copied to `save_globals` for later use. Then on line 8 an empty (local) environment is created and on lines 10 and 11, the init code (if present) is evaluated relative to the given global and local environments, updating the local environment. Next, on line 14, the student code is evaluated in the local environment, updating that environment. On lines 15 and 16, the local environment is merged with the global environment and, just in case the student redefines something in `global_env` or in the start code, `test_globals` is then updated again with `save_globals`. Finally, on line 18 and 19, the test code is evaluated in the `test_globals` environment.

By manipulating environments in this way, the student cannot gain access to, or modify, the information used for testing but the testing code can access all required information needed to test the student code.

When MyPyTutor is run with an online component it is possible to automatically update both the tutorial problems and even MyPyTutor itself. Whenever a student logs on, the date stamp of the tutorial problems is compared with the date stamp on the server. If the server has a more recent version then it lets MyPyTutor know that a newer version is available and in response MyPyTutor downloads the zip file and overwrites the old version with the new one. The student is notified that an update has happened. The same thing happens with MyPyTutor itself, except the version numbers are used for comparison.

7 Remarks

MyPyTutor was used for the first time for assessment in a first year software engineering course in semester 1, 2010. Several errors were discovered with the tutorial problems. Some errors were to do with poor wording and others were to do with code testing as discussed earlier. Because of the automatic update facilities of MyPyTutor it was easy to fix errors and

distribute a new version quickly. The ability to update MyPyTutor itself also turned out to helpful as students discovered several errors when using the application.

MyPyTutor provides a feedback facility so that students can provide feedback on both MyPyTutor itself and on specific tutorial problems. Feedback from students was mostly very positive.

More details of MyPyTutor, developer tools, and the current database of problems can be obtained from the author.

References

Stephen C. Ehrmann, Steven W. Gilbert and Flora McMartin (2006), Factors Affecting the Adoption of Faculty-Developed Academic Software: A Study of Five iCampus Projects, TLT Group, http://icampus.mit.edu/index.shtml (last referenced 13/08/2010).

Christopher Douce, David Livingstone and James Orwell (2005), Automatic Test-Based Assessment of Programming: A Review, *in* 'Journal of Educational Resources in Computing', Vol. 5, No. 3, ACM Press, New York.

S.H. Edwards (2003), Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance, *in* 'Proceedings of the International Conference on Education and Informations Systems: Technologies and Applications', pp. 421-426.

Michael T. Helmick (2007), Interface-based Programming Assignments and Automatic Grading of Java Programs, *in* 'ITiCSE 07: Proceedings of the 12th annual conference on Innovation and technology in computer science education', ACM Press, New York, pp. 63-67 .