Integrating the Media Computation API with Pythy, an Online IDE for Novice Python Programmers

Ashima Athri

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Stephen Edwards, Chair
Deborah Tatar
Steve Harrison

August 6, 2015
Blacksburg, Virginia

Keywords: computer science education, python, javascript, web-based, media computation

Integrating the Media Computation API with Pythy, an Online IDE for Novice Python Programmers

Ashima Athri

(ABSTRACT)

Improvements in both software and curricula have helped introductory computer science courses attract and retain more students. Pythy is one such online learning environment that aims to reduce software setup related barriers to learning Python while providing facilities like course management and grading to instructors. To further enable its goals of being beginner-centric, we want to integrate full support for media-computation-style programming activities. The media computation curriculum teaches fundamental computer science concepts through the activities of manipulating images, sounds and videos, and has been shown to be successful in retaining students and helping them gain transferable knowledge. In this work we tackle the first two installments of the problem namely, supporting image and sound-based media computation programs in Pythy. This involves not only client-side support that enables students to run media-computation exercises in the browser, but also server-side support to leverage Pythy's auto-grading facilities. We evaluated our implementation by systematically going through all 82 programs in the textbook that deal with image and sound manipulation and verifying if they worked in Pythy as-is, while complementing this with unit-tests for full test coverage. As a result, Pythy now supports 65 out of the 66 media-computation methods required for image and sound manipulation on both the client and the server-side, and 81 out of the 82 programs in the media-computation textbook can be executed as-is in Pythy.

Acknowledgments

I would like to thank my advisor, Dr. Stephen Edwards, for his support, encouragement, positive feedback, and belief that I could do it. I learnt a lot from our conversations during the weekly meetings. I would also like to thank my committee members Dr. Deborah Tatar and Dr. Steve Harrison for their support throughout the process.

Words can not express the gratitude for my mother Suma Athri without whom I would never have made it here.

I thank my husband Aditya Kulkarni for being my friend, guide and partner throughout the highs and the lows.

Finally, I thank my friends and fellow graduate students Zahra Ghaed, Mariam Umar and Eman Badr who continue to keep my spirits up as they did throughout graduate life, Sharon Kinder-Potter for her kind words and for being ever ready to help out, and Tony Allevato who built Pythy and laid the foundation for this work; it was an honor to work on the same project and learn from your work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the recognition of the applicability of computing in many fields, computer science education has become a pre-requisite for almost all fields of study. Learning to program, however, is an inherently difficult task. The student has to not only learn a new language with its own syntax and semantics, but has to learn to understand a problem, outline an algorithmic solution for it, and use the constructs of the language to solve it. To make matters more challenging, students have to grapple with cryptic error messages, logical errors that are hard to debug for inexperienced programmers and confusion about whether the error message is because of the code that they wrote or because the software they're using has not been setup properly. Many works of research focus on fixing the curricula so as to make courses more interesting and relevant to students from all majors. Very little work exists on fixing the problems on the software side of things.

Pythy was developed to make this software side of the introductory programming experience more palatable to novice students. Its main aims were to deal with the problems of difficultly in setting up the programming environment, not being beginner-friendly, unclear error reporting, lack of infrastructure for course organization and no support for in-class examples. It is web-based and aids in the teaching of Python to novice programmers [21] by eliminating the software setup phase, auto-saving student-written code in the cloud, providing clear, helpful and instant feedback in the form of automatic grading, and providing better support for assignment organization and in-class examples. It also provides an instructor friendly interface to add students to the course, create an assignment definition, give in-class examples and manage grade data.

Pythy, as part of its effort to be beginner friendly, has some support for the media computation curriculum [32] which centers on manipulation of media (images, sounds, and video) as a way to introduce computer science concepts to students. The media computation curriculum [23] is a major shift away from the way that an introductory course on computer science has traditionally been taught and has achieved great results in terms of student engagement, retention [31] and pass rates [30].

In its current state, Pythy only provides basic support for the image manipulation portion and minimal support for the sound manipulation part of the media computation API [32]. This basic support itself, in some cases, is not consistent with published media computation API [10] [16] in terms of method behavior and signature on both the client side (the student view) and the server side (the grading view).

Due to this mismatch, instructors are forced to deviate from the course textbook, "Introduction to Computing and Programming in Python, A Multimedia Approach", when presenting the material to students. This may lead to confusion on the part of the students as they may not know which method to use or how a particular method behaves. Thus, the students may not get the full benefit of the media computation curriculum and the instructor has to put in more effort to ensure that students can use Pythy in combination with media computation.

1.1 Problem Statement

The problem that we focus on is **how to integrate media computation with an online Python learning environment like Pythy.** This is further broken down into four parts:

1. How to add support for image manipulation in Pythy?
2. How to add support for sound manipulation in Pythy?
3. How to provide support for grading media computation based assignments in Pythy?
4. What are the changes required in Skulpt, the library that helps run Python in the browser, in order to be able to run media computation-based programs in Pythy?

1.2 Solution

This thesis describes the method in which complete coverage for the media computation API for both images and sound was achieved on both the client-side and the server-side.

The percentage of coverage was verified through the following two methods:

1. Running snippets of code from the textbook manually and verifying that the results are as expected.
2. Behavior Driven Development (BDD) style tests [26] that document the capabilities of the code along with verifying code behavior.

As an added benefit, this work also documents how to make a library for Skulpt, helping those who want to integrate a new curriculum or API into Pythy.

Pythy now supports 37 out of 37 image manipulation methods (in both procedural and object oriented forms), 18 out of 19 sound manipulation methods (in both procedural and object oriented forms) and 10 out of 10 utility methods, on both the server side and the client side.

1.3 Thesis Organization

The remainder of this document is organized as follows. Chapter 2 goes through some competing frameworks along with existing work that served as the foundation for this project. Chapter 3 talks about the process of integrating the media computation API into Pythy and the challenges that were faced while doing so. Chapter 4 details the process in which the solution was verified and provides screenshot-based comparison between Pythy and JES outputs. Finally, Chapter 5 discusses the project's contributions and points to some ways in which the work can be improved.

Chapter 2

Related Work

2.1 Contextualized Learning

It has been shown that students are more motivated and engaged, learn more, and have a higher sense of competence and higher levels of aspiration when they are presented educational material in a meaningful and interesting context [20]. This has also been shown to be true in case of introductory programming [22].

Popular examples of the contextualized approach to teaching introductory programming are—using the Alice and Scratch environments, developing smartphone applications, and following the media computation curriculum.

Alice [1] provides students a 3D virtual world that they can manipulate using Alice programs. These Alice programs are constructed by dragging and dropping constructs and customizing them with the help of the provided menu. Alice has been proven to be effective in improving student performance of 'at risk' students in CS1 [25].

Scratch [14] [29], similar to Alice, requires students to work with a 2D environment using a drag and drop interface. The USP of Scratch is the ability to showcase your finished work on the public website. It has more than 10 million publicly shared projects demonstrating its attractiveness and popularity.

Reardon and Tangney [28] used smartphones in conjunction with studio-based learning to teach introductory programming. This provided students the opportunity to develop applications for a context relevant to them and was shown to have motivated the students.

Media computation [23] is a curriculum that is based on the premise that media in the digital format serves as a good context to introduce computer science concepts to students. It was originally built to attract non-CS majors to introductory CS courses but has also been proven to be an effective tool in a majors' CS1 course [30]. As opposed to Alice or Scratch

it uses Python which is a general purpose language and hence makes transition to real-world programming easier.

2.2 Online Python Learning Environments

With Python rising in popularity as a language for teaching programming and the web becoming recognized as a great medium for it, several online Python learning environments have emerged. Three examples of such environments are Interactive Python, PythonAnywhere and Online Python Tutor.

PythonAnywhere [12] enables students to create Python consoles, in a Python environment of their choosing, that can be accessed through the browser, but live on Amazon EC2 servers. These consoles can be shared with teachers and other fellow students. It also allows users to create database instances and build web applications in Django, a Python web application framework, along with the ability to host them. The focus of PythonAnywhere is to simplify setting up of Python with all the required packages on students' machines.

Interactive Python [13] allows teachers to create interactive textbooks with embeded executable code, code visualizations, form-based questions, code-based questions, videos, polls, comment boxes and many other helpful features inline with the textbook content.

Online Python Tutor [9] is a web-based tool that helps students understand programs by visualizing the stack frames corresponding to each program statement. This tool works not only for programs written in Python but also for other languages like Java, JavaScript, TypeScript and Ruby.

While these frameworks do remove software-related barriers to learning programming in Python similar to Pythy, they do not support course management activities like course creation, uploading assignments, automatic grading and the ability to download grades for a term which are possible in Pythy.

2.3 JES

JES, the Jython Environment for Students [4], is a programming environment built around media computation and is used in the course textbook [24], to explain the behavior of the programs. JES is built on top of the JVM with the core written in Java and the UI code written in Jython, an implementation of Python on the JVM [6]. It makes use of the Swing and AWT frameworks for media manipulation, the IDE functionalities, and the media viewers. Swing and AWT being mature frameworks have strong support for media manipulation and as a result it is easier to build the media computation API using them.

Pythy on the other hand uses a third party library called Skulpt [15] to convert Python

code into JavaScript runnable by the browser. It is a young framework with little to no documentation and very few examples of how to plug libraries into it.

Using JavaScript for media manipulation introduces another challenge, that of the maturity of the image and sound API in the browser. The HTML5 Canvas and related API are relatively new additions to browsers, the first public draft of HTML5 being published in 2008 [18], and do not support some methods, like drawing ellipses, that are provided in Java out-of-the-box. The first public draft for Web Audio was published in December 2011 [19], and there is currently no browser support for conversion between different sound formats, like mp3 to wav, and this codec must be programmed manually.

Pythy, unlike JES which is only an IDE, also needs the API to be implemented both on the client side (the student view) and the server side (the grading view) with exactly the same behavior while taking into account the differences in the languages (JavaScript and Python) and the execution environments (browser vs a server machine, asynchronous vs synchronous etc.).

2.4 Pythy

Pythy is a learning environment developed with the broader goal of preventing software-related issues from negatively affecting the introductory programming experience.

It has been shown to be easier to use, more beginner friendly, and to have better support for assignment organization and in-class examples as compared to JES [21].

It is a web application with Ruby on Rails handling the server-side of the work and JavaScript the client-side.

To begin with, students must create an account with Pythy. They then login using this account to join courses, access course-related assignments, in-class examples, work done so-far on assignments and things that they have tried out previously on the scratchpad.

As all the work is stored on the server and is version controlled, students can go back and forth between old and the current versions of their work. They can also use the grading tool to check how they are doing.

In relation to media computation, students have access to a media library in the cloud which they can use to store media (pictures and sounds) related to the course. The media library can be accessed by clicking on the cloud button on the top right (see Figure 2.1).

Clicking on the media thumbnail inserts the link to it in the student's work area and clicking on the open button next to the media opens it in the corresponding media viewer. Students can delete existing media and upload new ones through this media library interface.

This is different from file-system-based access to media files in JES where the student has

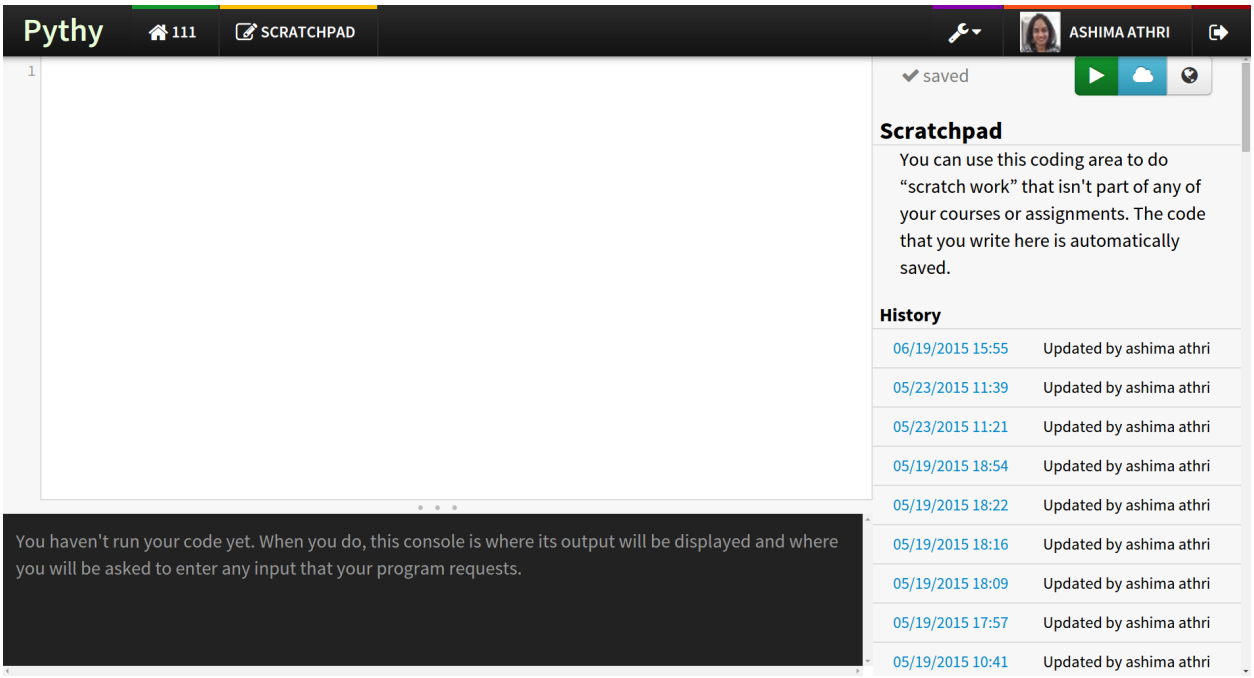
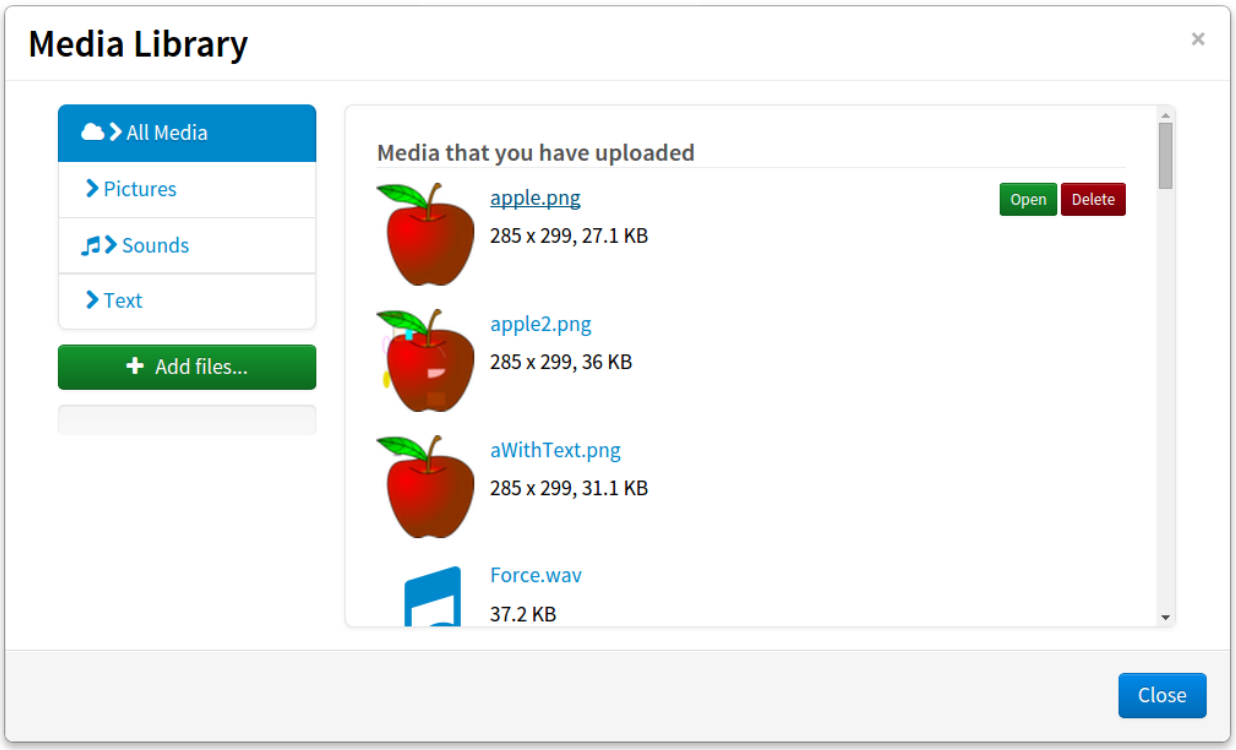
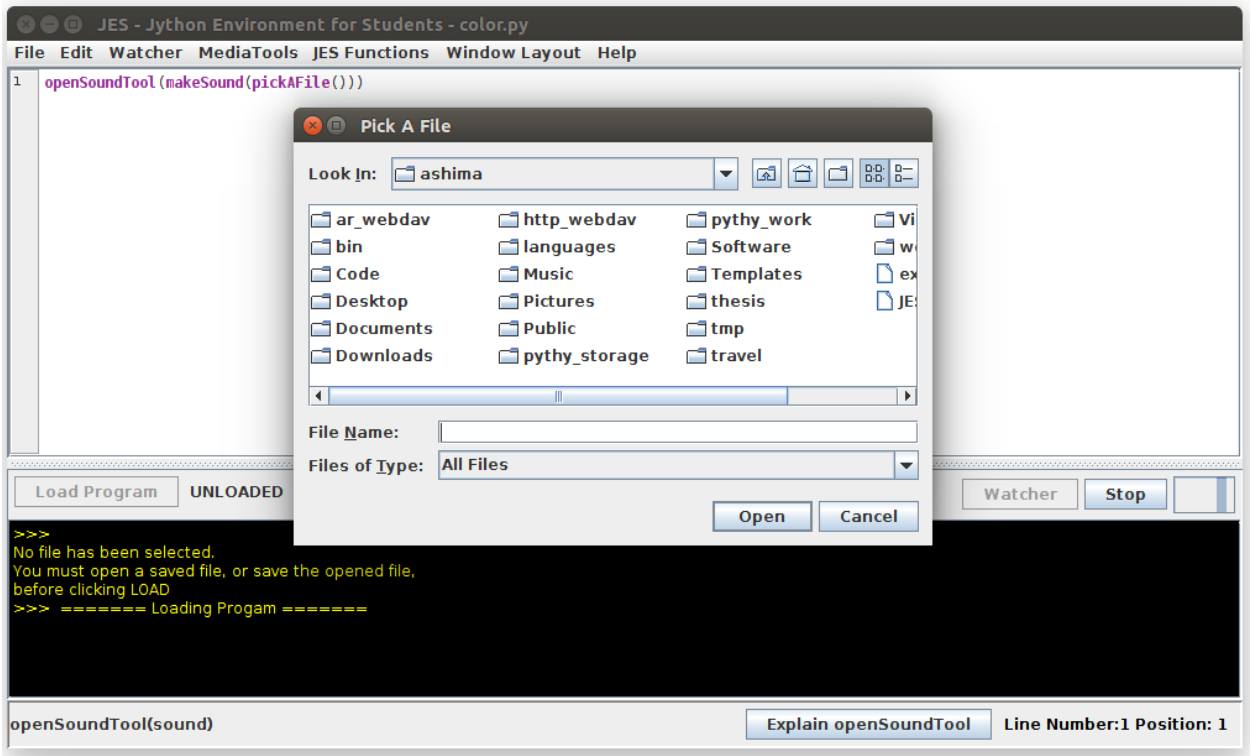


Figure 2.1: The Pythy Scratchpad

to either know the absolute path of the file or traverse the directory structure through a file-explorer to find the file they were looking for. All paths in Pythy are relative to the student's media library and have a maximum nesting level of 0. The difference between the two interfaces can be seen in Figure 2.2.



(a) The Pythy Media Library



(b) The JES File Explorer, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 2.2: Selecting a File

2.5 Skulpt

Skulpt is a JavaScript implementation of Python [15]. Skulpt implements the Python2.6 grammar with some support for Python3 semantics.

It should be noted that Skulpt is not a pure translator that takes Python code and produces JavaScript that can be immediately run in a browser. Skulpt-generated JavaScript needs the Skulpt library to run. The Skulpt library has methods that perform type checking, argument checking, conversion between JS and Python functions, between JS and Python classes etc.

An important requirement was the capability to interrupt the execution of the Python code to keep the other elements in the DOM responsive. JavaScript has a single-threaded event-based model of execution [2]. This meant that the interpreter would monopolize the single thread of execution while executing student-written code leading to the appearance of the browser being 'hung' during this time, i.e. none of the other elements on the page would respond to user activity.

It was needed to be able to make the Python execution yield so that other events on the page could be processed by the browser's event loop. At the time of working on integrating the media computation API into Pythy, no support for this was available. Thus, the desired yield behavior was implemented in Skulpt and is described in Section 2.5.2.

In some cases, for example taking user input or for method calls like blockingPlay, it was necessary to simulate serial behavior while at the same time keeping the browser responsive. Almost all i/o in JavaScript is asynchronous, relying on callbacks to ensure that the desired behavior is applied at the right time. However, Python code is implicitly serial. To simulate this kind of serial behavior, support for futures was added in Skulpt. This is described in Section 2.5.3.

As these two features had already been implemented, part 4 of the problem statement was mostly resolved and Skulpt was ready to be used with media computation programs.

2.5.1 Skulpt-generated Code and Execution Model

Listing 2.2 shows the skulpt-generated JavaScript for the simple Python program in Listing 2.1.

Listing 2.1: hello_world.py

```
1 print("hello world")
```

Every Python module, method or class is represented by a JS function like "$scopeX", with X being a number. This function contains an infinite while loop with a switch case whose case statements correspond to basic blocks. The switch variable controls which block will be

executed next, thereby controlling the overall execution pattern. This control variable along with the locals and the globals form the execution state of a method, class or module.

Listing 2.2: hello_world.js

```
Sk._execModule(function($moddata) {
  $moddata.scopes["$scope0"] = (function($modname) {
    var $frm = Sk._frameEnter(0);
    var $ctx = $frm.ctx,
    $exc = $ctx.$exc || [];
    $gbl = $ctx.$gbl || {};
    $loc = $ctx.$loc || $gbl;
    $err = undefined;
    $gbl.__name__ = $modname;
    $ctx.$exc = $exc; $ctx.$gbl = $gbl; $ctx.$loc = $loc;
    if (Sk.retainGlobals) {
      if (Sk.globals) {
        $gbl = Sk.globals;
        Sk.globals = $gbl
      } else { Sk.globals = $gbl; }
    } else { Sk.globals = $gbl; }
    try {
      while (true) {
        try {
          switch ($frm.blk) {
            case 0:
              /* --- module entry --- */
              // line 1:
              // print("hello world")
              // ^
              Sk.currLineNo = 1;
              Sk.currColNo = 0;
              Sk.currFilename = "hello_world.py";
              $ctx.$loadname1 = $ctx.$loc.print !== undefined ?
                                $ctx.$loc.print :
                                Sk.misceval.loadname("print", $gbl);
              $ctx.$str2 = new Sk.builtins["str"]("hello world");
              $frm.blk = 1; /* jump */
              Sk.yield();
              continue;
              throw new Sk.builtin.SystemError("unterminated block");
            case 1:
              /* --- before call --- */
              $ctx.$call3 = Sk.misceval.callsim($ctx.$loadname1,
                                                $ctx.$str2);
              $frm.blk = 2; /* jump */
              Sk.yield();
              continue;
              throw new Sk.builtin.SystemError("unterminated block");
            case 2:
              /* --- after call --- */
```

```
47                  // line 1:
48                  // print("hello world")
49                  // ^
50                  Sk.currLineNo = 1;
51                  Sk.currColNo = 0;
52                  Sk._frameLeave();
53                  return $loc;
54                  throw new Sk.builtin.SystemError("unterminated block");
55              }
56            } catch (err) {
57              if ($exc.length > 0 &&
58                  !(err instanceof SuspendExecution)) {
59                $err = err; $frm.blk = $exc.pop();
60                Sk._frameLeave();
61                Sk.yield();
62                continue;
63              } else { throw err; }
64            }
65          }
66        } catch (err) {
67          if (err instanceof Sk.builtin.SystemExit &&
68              !Sk.throwSystemExit) {
69            Sk.builtin.print_(err.toString() + "\n");
70            return $loc;
71          } else { throw err; }
72        }
73      });
74      var $scope0 = $moddata.scopes["$scope0"];
75      return $scope0(new Sk.builtin.str("__main__"));
76  });
```

2.5.2 Support for Interrupted Execution

There are two requirements to support interrupted execution of Skulpt-generated JavaScript; the ability to:

1. Pause and resume execution of the Python program.
2. Save the execution state of the Python program when paused; this could then be used later on, when the execution was resumed.

It was also necessary to decide at what point in execution the Python program should be paused. The point just before a jump was chosen to be the checkpoint. At this checkpoint, it is checked to see if the execution time of the Python code has exceeded a pre-defined, configurable limit, which by default is 100ms. If it has, then a SuspendExecution exception is thrown to signal that it is time to give up the thread to process DOM events.

For requirement 1, a monitoring function was introduced that is reponsible for controlling the execution of the Python program. This monitoring function executes the Skulpt-generated code while catching any exceptions thrown by it. If the exception is of the type SuspendExecution, it reschedules it's own execution at a later time using the JavaScript setTimeout method.

For requirement 2, it was necessary to be able to save the state of the program up to the pause point. As discussed in Section 3.2.1, the execution state of each method, class or module is the combination of the next block number, the locals and the globals. The execution state of each method that has been entered but not returned from up to the point when the program was paused (threw an exception) needed to be saved. This was done by saving each method's execution state on a stack when the method is entered and popping it off when the method is returned from. This stack can be considered to be the execution state of the program and is saved just before throwing the SuspendExecution exception.

When the next run of the monitoring function executes, it starts execution of the Skulpt-generated code all over again from $scope0. However, this time, everytime a new scope is entered, one item is popped off the stack starting from the bottom (in reverse order), basically using the stack like a queue. This helps in re-creating the execution as it was when the program was last paused.

For example, if the stack at the time of resuming the program looked like Figure 2.3, then when $scope0 is entered, it will retrieve the first item from the stack as its state. Now, it will have the next block number to execute and the local and global definitions. This block number will correspond to the block in which the method corresponding to $scope1 was called. When $scope1 is entered, it will retrieve the second item from the stack which is its state at the time the program was paused. This process will go on until the stack is empty, meaning that all the methods are at the state they were in when the program was paused. Note that no other blocks were executed during this procedure, so no redundant computation was performed.

2.5.3 Support for Futures

The future construct is used when we want to simulate Python's implicit synchronousity for asynchronous activities in JavaScript. For example, when students use the input function, the program must wait till the user hits enter. In JavaScript, the program is explicitly synchronized with the help of callbacks. For the purpose of comparison, examples of processing user input in JavaScript and Python are given in Listing 2.3 and Listing 2.4 respectively.

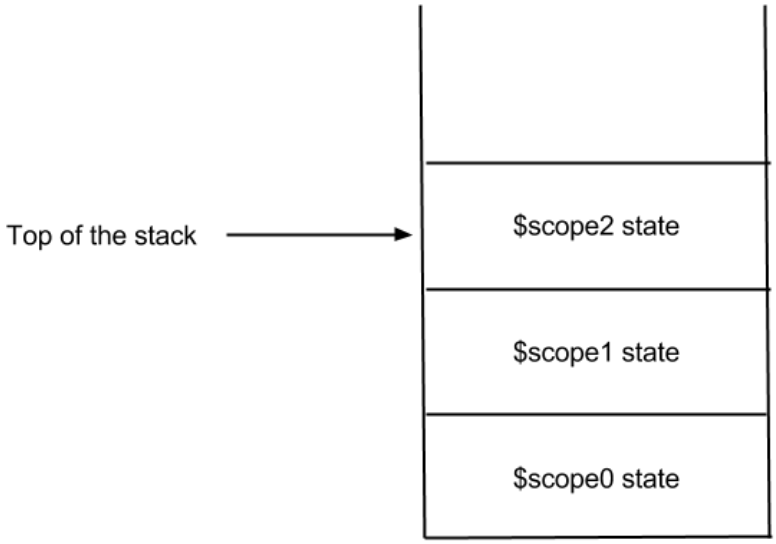


Figure 2.3: The Execution State of a Python Program

Listing 2.3: input.html

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <input>
    <script>
      (function () {
        var inputBox, onChange;

        inputBox = document.getElementsByTagName("input")[0];
        onChange = function () { window.res = inputBox.value; };
        inputBox.addEventListener("change", onChange);
      }());
    </script>
  </body>
</html>
```

Listing 2.4: input.py

```
res = input()
```

The Listing 2.5 outlines the usage of the future construct in Skulpt using the processing of user input as an example. It takes a function, the one that performs asynchronous activities, as its argument. This function must take a callback as an argument which it must call at the end with the result of the asynchronous computation. This will be the result assigned to res in line 13. The intended effect is that line 13 in the listing looks very similar to the Python code in Listing 2.4 and has the intended synchronous, blocking behavior.

Listing 2.5: input_with_future.js

```
(function () {
  var inputBox, onChange, res;

  inputBox = document.getElementsByTagName("input")[0];

  onChange = function (continueWith) {
    inputBox.addEventListener("change", function () {
      continueWith(inputBox.value);
      inputBox.removeEventListener("change");
    });
  };

  res = Sk.future(onChange);
}());
```

In order to support the blocking behavior, it was necessary to have the ability to do two things:

1. Pause and resume execution of the Python program.
2. Preserve the assignment semantics.

For requirement 1, the infrastructure that was built to support interruptable execution was leveraged. In this case however, instead of the monitoring function rescheduling itself for execution at a later time using the setTimeout function, it passes itself as a callback to the asynchronous function. This way, the program appears to be 'blocked' waiting for the asynchronous function to complete while all the other DOM elements are still responsive. When the asynchronous activity is complete, the callback (which is the monitoring function) is called with the result of the activity as the argument.

This result is stored as part of the global state of the program. When the program resumes execution and reaches the point when Sk.future was called, the result is ready in the form of global state and is the return value of Sk.future. Note that Sk.future is called twice during the execution of the Python program for everytime it appears in the code, that is for every asynchronous activity it wraps. The first time it is called (the pause phase), it saves the execution state of the program while the monitoring function performs the asynchronous activity and saves the result. The second time it is called (the resume phase), it returns this result. Hence requirement 2 is satisfied.

2.5.4 Writing a Library in Skulpt

A basic skeleton of a library module in skulpt is given in Listing 2.6. The function $builtinmodule is executed every time the corresponding module is imported from the Python program. The $builtinmodule must be a function that returns the module object, mod in this case. The module object is just a plain JavaScript object that may contain Python-specific classes and functions. In the example, the module contains two classes, one inheriting from the other, and a global function.

Listing 2.6: A Skulpt library module

```
var $builtinmodule = function(name) {
  var mod;

  mod = {};

  mod.myClass = Sk.misceval.buildClass(mod, function($gbl, $loc) {
    $loc.__init__ = new Sk.builtin.func(function(self, arg1, arg2) {
      ...
    });
  }, "myClass", []);

  mod.subClass = Sk.misceval.buildClass(mod, function($gbl, $loc) {
    ...
  }, "subClass", [mod.myClass]);

  mod.globalFunc = new Sk.builtin.func(function (self, arg1) {
    ...
  });

  return mod;
};
```

Chapter 3

API Integration

The media computation API consists of methods that students can use to manipulate images, sounds and video. This work only deals with image and sound suppport. Sections 3.1 and 3.2 talk about the implementation of the image and sound media-computation API in the context of a browser. Section 3.3 discusses the implementation of the API on the server-side along with the helper methods added to make writing media-computation-based tests easier. Finally, Section 3.4 talks about the modifications that we made to Skulpt in order to support Python 3 semantics.

3.1 Image API Integration

The HTML5 Canvas API was used to implement the Image API on the client-side. All methods from Tables 3.1, 3.4, 3.2, 3.3 and 3.7 had already been implemented except for makeBrighter, getAllPixels, setColorWrapAround and getColorWrapAround. Although most of the image API had been implemented, due to various reasons, the methods did not work as expected, produced undefined behavior or threw errors when used.

One of the major reasons for this was that the implemented methods did not explicitly unbox values they received as parameters. According to Python semantics, everything is an object. So, the values received as parameters in Skulpt methods are actually Skulpt representations of the corresponding Python objects. Before using these parameters, they must first be unboxed using Skulpt's *Sk.unwrapo* method. The previous implementation used these values directly and hence the code did not work as expected or threw errors. The reverse side also holds true, that is, before returning a value to the 'Python-world', it must be boxed into the correct object type. A significant amount of effort was spent, as part of this work, going through each method, unboxing parameters before their usage and boxing return values into the correct object type. This was also true for methods of the sound API. An improvement to this manual approach would be to delegate the task of parameter unboxing

to the machinery responsible for calling builtin or library methods. An alternative approach that would include boxing return values would be to use aspect oriented programming to add advices that run before and after the library method was executed.

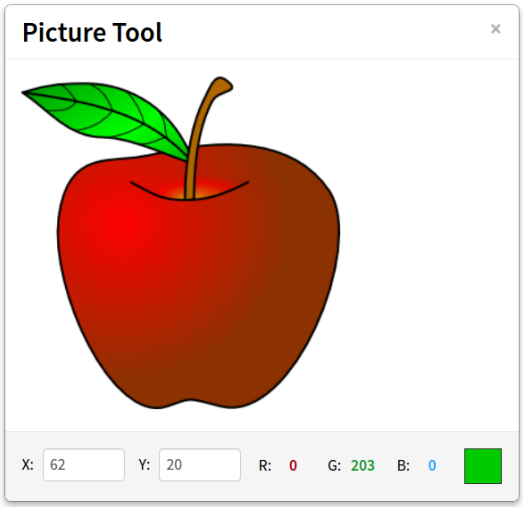
Some other methods were inconsistent with the published API due to their method signatures. The method getPixels returned a generator instead of a list and the method makeColor took exactly 3 arguments when the published API method allowed variable number of arguments (1 or 3). The method makeStyle interprets its parameters in a way that is different from what the published API does. It treats the concatenation of the parameters font, emphasis and style as a JavaScript font string. However, this does not match the values accepted by the published API. So, we have added logic to map the correct parameter values, accepted by the published API, to values that can be used to construct the JavaScript font string.

Another cause for inconsistency was JavaScript behavior that was not accounted for. In JavaScript pixels are represented as a quartet of red, green, blue and alpha (rgba) values. The picture is represented as an array that has the rgba values of all pixels and is arranged in the following way. The pixel at position (x, y) has its red value stored at index $y * 4 * pictureWidth + x * 4$, green value at $y * 4 * pictureWidth + x * 4 + 1$, blue value stored at $y * 4 * pictureWidth + x * 4 + 2$ and alpha value stored at $y * 4 * pictureWidth + x * 4 + 3$. Everytime we change one of the r, g or b values, we must make sure that alpha is set to 255, else if alpha is 0, that pixel will be set to transparent which looks black in the case of the HTML5 canvas. Some images have a transparent background and hence will have alpha set to 0. Initially when these images are loaded, the background will remain white but manually setting the pixels will cause the alpha value to be multiplied with the r, g and b value of that pixel, causing that pixel to look black instead of the set color. Previously, this behavior was not taken into account in the method setRed, setGreen, setBlue and setColor.

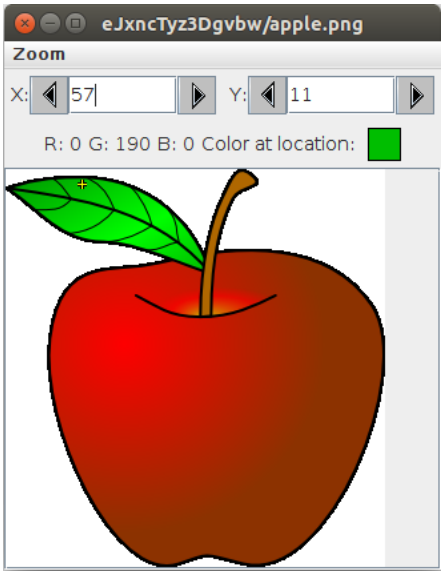
Other minor corrections were, changing the output of the __str__ to match the result of the published API, and fixing spelling errors like renaming copyInfo to copyInto.

We also noticed that many methods—getColor, setColor, addArc, addArcFilled, addLine, addOval, addOvalFilled, addRect, addRectFilled, addText, addTextWithStyle, duplicatePicture, getHeight, getWidth, getPixel, getPixelAt, getPixels, repaint and show, did not have a corresponding object oriented counterpart. Also, the procedural methods that did have an object oriented counterpart would call the counterpart from within their code, most probably to prevent code duplication. This however would increase the number of function calls during execution of student written code, as students mostly used the procedural version, and slow down the students' program. In order to prevent code duplication while having both the procedural and object-oriented versions without slowing down performance, we created a container object that held all the common methods and simply extended the module and the class with this object. Thus, when instructors wish to introduce students to the object-oriented methodology of writing programs, students have access to all the methods they used previously without sacrificing program speed.

An image viewer similar to the JES Media Tool had already been implemented. The comparison between the Pythy and JES image viewers can be seen in Figure 3.1. With the help of this tool, students can examine each pixel of an image to check its RGB values. The image of the apple is taken from https://upload.wikimedia.org/wikipedia/en/5/54/Red_Apple.png.



(a) Pythy



(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 3.1: The Picture Tools

Table 3.1: The Media Computation Image API - Picture

#	**Method Signature**
1	addArc(picture, startX, startY, width, height, start, angle[, color]) picture.addArc(startX, startY, width, height, start, angle[, color])
2	addArcFilled(picture, startX, startY, width, height, start, angle[, color]) picture.addArcFilled(startX, startY, width, height, start, angle[, color])
3	addLine(picture, startX, startY, endX, endY[, color]) picture.addLine(startX, startY, endX, endY[, color])
4	addOval(picture, startX, startY, width, height[, color]) picture.addOval(startX, startY, width, height[, color])
5	addOvalFilled(picture, startX, startY, width, height[, color]) picture.addOvalFilled(startX, startY, width, height[, color])
6	addRect(picture, startX, startY, width, height[, color]) picture.addRect(startX, startY, width, height[, color])
7	addRectFilled(picture, startX, startY, width, height[, color]) picture.addRectFilled(startX, startY, width, height[, color])
8	addText(picture, xpos, ypos, text[, color]) picture.addText(xpos, ypos, text[, color])
9	addTextWithStyle(picture, xpos, ypos, text, style[, color]) picture.addTextWithStyle(xpos, ypos, text, style[, color])
10	copyInto(smallPicture, bigPicture, startX, startY) smallPicture.copyInto(bigPicture, startX, startY)
11	duplicatePicture(picture) picture.duplicate()
12	getHeight(picture) picture.getHeight()
13	getPixels(picture) picture.getPixels()
14	getAllPixels(picture) picture.getAllPixels()
15	getPixel(picture, xpos, ypos) picture.getPixel(xpos, ypos)
16	getWidth(picture) picture.getWidth()
17	makeEmptyPicture(width, height[, color])
18	makePicture(path) Picture(path)
19	repaint(picture) picture.repaint()
20	setAllPixelsToAColor(picture, color) picture.setAllPixelsToAColor(color)
21	show(picture) picture.show()

Table 3.2: The Media Computation Image API - Color

#	**Method Signature**
1	distance(color1, color2) color1.distance(color2)
2	makeBrighter(color) color.makeBrighter()
3	makeDarker(color) color.makeDarker()
4	makeLighter(color) color.makeLighter()
5	makeColor(red[, green, blue]) Color(red[, green, blue])

Table 3.3: The Media Computation Image API - Style

#	**Method Signature**
1	makeStyle(fontName, emphasis, size) Style(fontName, emphasis, size)

Table 3.4: The Media Computation Image API - Pixel

#	**Method Signature**
1	getColor(pixel) pixel.getColor()
2	getRed(pixel) pixel.getRed()
3	getGreen(pixel) pixel.getGreen()
4	getBlue(pixel) pixel.getBlue()
5	getX(pixel) pixel.getX()
6	getY(pixel) pixel.getY()
7	setColor(pixel, color) pixel.setColor(color)
8	setRed(pixel, redValue) pixel.setRed(redValue)
9	setGreen(pixel, greenValue) pixel.setGreen(greenValue)
10	setBlue(pixel, blueValue) pixel.setBlue(blueValue)

3.2 Sound API Integration

The Web Audio API was used to implement the Sound API on the client-side. Previously, this API was supported minimally. It was possible to create sounds from scratch and play them. However, loading, manipulating and playing existing mp3 and wav audio files, which is what most programs in the textbook depend on, was not possible. The changes we've made as part of this work has enabled that. The methods that were not supported previously but are now supported are makeSound, getSamples and stopPlaying.

Many changes and improvements were similar to those done for the image API and for the same reasons. We added object-oriented counterparts for methods like duplicateSound, getDuration, getLength, getNumSamples, getSampleObjectAt, getSampleValue, getSampleValueAt, getSamplingRate, and setSampleValue to ensure uniformity and improve program speed. We also fixed inconsistencies in the output of the __str__ methods of both the Sound and the Sample class. Again, a significant amount of effort went into unboxing parameters and boxing return values into correct types.

As this software is intended for beginner programmers, it is important to guard against incorrect values to prevent errors that students can not understand. For methods like getSampleObjectAt, getSampleValueAt and getSampleValue, we had to explicity make sure that the index provided was within bounds as JavaScript does not do bounds checking. Similarly for their 'set' counterparts, along with checking the index, we had to make sure that the new sample value was between -32768 and 32767 and cap the value if it was not.

As the Web Audio API is still in its nascent stage, the functionality it provides is very basic and in some cases can not be customized as per the user's requirements. For example, the Web Audio API resamples audio files with the default sampling rate of the browser (usually 44100 Hz). This means that if a sound was originally sampled at any other frequency, we will end up with a different number of samples after the browser has resampled the sound. This entails changing values in programs that hardcode sample numbers, like Program 67 in the textbook. This also means that the value of a given sample in Pythy will be different from the value of the sample at the same index in JES.

Another reason for sample values being different in Pythy and JES is the fact that the Web Audio API stores sample values as 32 bit floating point numbers between -1 and 1 regardless of the bit depth of the audio file. The bit depth is the number of bits in each sample. Audio files may have a bit depth of 8, 16, 24, 32 etc. On the other hand, JES stores sample values as integers with their range being determined by the bit-depth of the audio file. To maintain a consistent return type, we convert between 32 bit floating point and 16 bit integers while setting and getting sample values. This may lead to a slightly different sample value due to rounding errors.

As JavaScript is single threaded, we needed to find a way, other than using multiple threads, to protect the sound from being modified while it's being played. We did this by cloning the

sound and playing the clone instead. This way, the original sound can be modified while it's being played without affecting the playback.

When the sound is loaded in the browser using the Web Audio API, it gets converted to an in-browser representation. In order to support the capability of saving sound files, we needed a way to convert this in-browser representation to the wav or mp3 format. As this functionality is not provided by the Web Audio API out-of-the-box, we had to create a method to perform the encoding. This method only converts an in-browser representation of the sound into a 16bit PCM mono or stereo WAV file. If the original sound was an 8bit PCM WAV file then it will get doubled in size. We do not convert to the mp3 format on the client-side, instead marking it to be converted on the server.

The only sound API method that was not implemented was playNote because it was not used much in the textbook examples and required a lot of programming effort to include.

We also developed a sound viewer that was not available previously. The comparison between the Pythy and JES sound viewers can be seen in Figure 3.2. The Pythy viewer replicates all the features available in the JES version. Using this tool, students can play the entire sound, a particular section of it, or after/before a specified sample number. Students can examine the value of each sample of the sound by clicking on the sample and looking at the its value. They can also go sample by sample using the next and previous buttons. The waveform of the sound is initially fit to the width of the modal in the browser. This means that all samples are not represented in it. Students can zoom in/out to view the waveform at a higher/lower resolution.

The sound viewer was built on an HTML5 canvas. Currently, it scales the canvas linearly with the length of the waveform and uses the native HTML scrollbar to view different parts of the sound. However, browsers have limitations on the width of a canvas [8] (32,767 pixels for Google Chrome and Mozilla Firefox, and 8,192 pixels for IE). This means that if a sound is too big (greater than 32767 samples in case of Chrome and Firefox) and the student zooms in beyond a certain limit, the browser will not be able to draw the waveform. To counter this, an improvement would be to use a fixed-size canvas with a dynamically drawn scrollbar and redraw the viewport everytime a scroll event is detected.

Table 3.5: The Media Computation Sound API - Sound

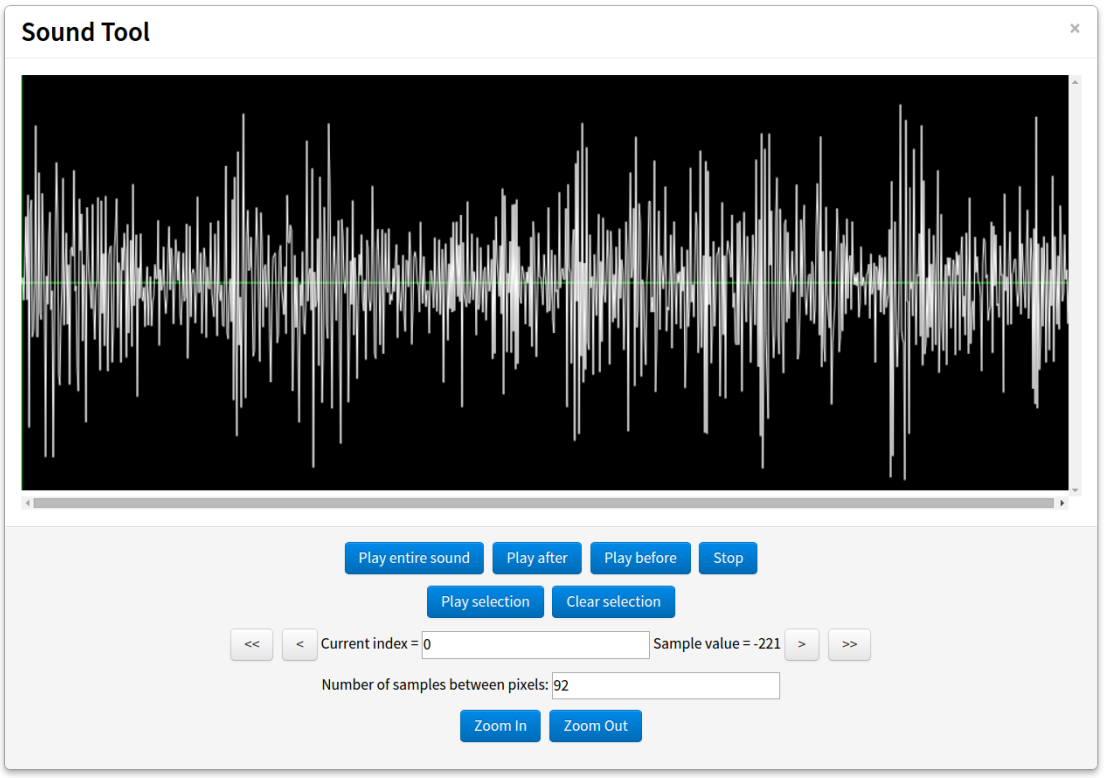
#	**Method Signature**
1	blockingPlay(sound) sound.blockingPlay()
2	duplicateSound(sound) sound.duplicate()
3	getDuration(sound) sound.getDuration()
4	getLength(sound) sound.getLength()
5	getNumSamples(sound) sound.getNumSamples()
6	getSampleObjectAt(sound, index) sound.getSampleObjectAt(index)
7	getSamples(sound) sound.getSamples()
8	getSampleValueAt(sound, index) sound.getSampleValueAt()
9	getSamplingRate(sound) sound.getSamplingRate()
10	makeEmptySound(numSamples[, samplingRate]) Sound(numSamples[, samplingRate])
11	makeEmptySoundBySeconds(duration[, samplingRate])
12	makeSound(path) Sound(path)
13	play(sound) sound.play()
14	setSampleValueAt(sound, index, value) sound.setSampleValueAt(index, value)
15	stopPlaying(sound) sound.stopPlaying()
16	playNote(note, duration[, intensity])

Table 3.6: The Media Computation Sound API - Sample

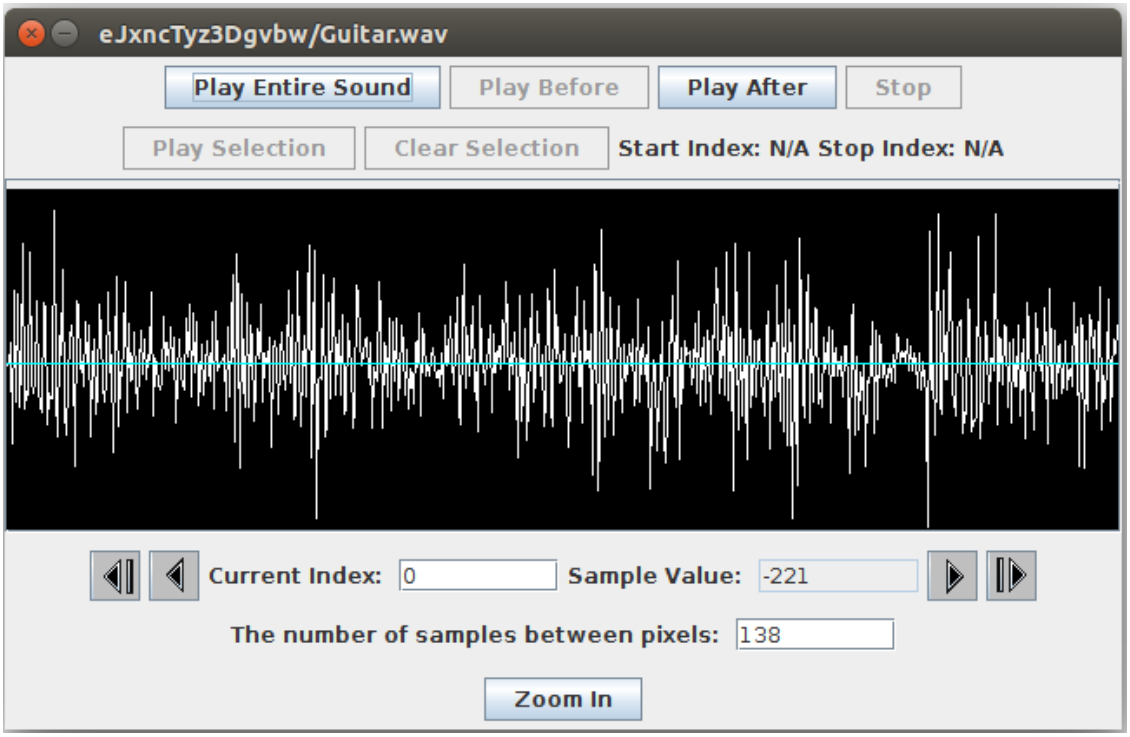
#	**Method Signature**
1	getSampleValue(sample) sample.getSampleValue()
2	getSound(sample) sample.getSound()
3	setSampleValue(sample, value) sample.setSampleValue()

Table 3.7: The Media Computation Sound API - Utility

#	**Method Signature**
1	setColorWrapAround(flag)
2	getColorWrapAround()
3	openPictureTool(picture)
4	pickAFile()
5	pickAColor()
6	writePictureTo(picture, path)
7	openSoundTool(sound)
8	writeSoundTo(sound, path)
9	setMediaPath(path)
10	getMediaPath()



(a) Pythy



(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 3.2: The Sound Tools

3.3 Support for Server-side Grading

All student-code is saved on the server. When the student requests their code to be graded by clicking on the 'magic wand' icon at the top right, the reference PyUnit tests written by the instructor are run against the student's code on the server. This requires that the methods used by the students should work on the server end too.

One way to resolve this problem would be to simply use JES's backend as our server-side implementation of the media computation API. The main drawback of this approach lies in the fact that JES is written in Jython. Jython currently supports Python 2.7, with Jython 2.7.0 final released just recently in May 2015 [5]. The roadmap of the Jython project seems outdated [7] and very little information is given about Python 3 support. Instructors however want to make the shift towards using Python 3 syntax in their courses. In order to remain flexible with respect to Python version support, we decided to stick to native cpython. This meant that we had to implement the media computation API from scratch on the server-side.

The Python imaging library, pillow [11], and the wave module [17] were used to implement the media computation image and sound API on the server-side. As this API is used only for grading, the methods show, repaint, play, blockingPlay, writePictureTo, writeSoundTo, openPictureTool, openSoundTool, and stopPlaying have been implemented as NOOPs.

The methods pickAColor and pickAFile return values that must be preset using the corresponding set functions, setPickedColor and setPickedFile. The set functions use a queue to store the picked colors and files, and return them in the same order as they were 'picked'.

As the wave module can only deal with wav files, we convert mp3 files to the wav format using the lame codec just after retrieving them from the url. Also, the wave module provides an binary-like interface to the wav file where the methods deal with bytes instead of samples. We must construct samples objects from the binary data using the bit depth (or sample width) of the wav file as the number of bytes that represent one sample.

The helper methods assertImagesSimilar and runFile had been implemented as part of the initial support for media computation in Pythy. We added another method to the TestCase class to make sound comparison easier. The descriptions of these methods are given below.

1. assertImagesSimilar
   This method is an assertion helper that compares two images to check if they are similar. It does this by comparing their heights and widths first. Then it goes on to compare the red, green and blue values of each pixel of the two images. The comparison is not strict but allows a tolerance of 1. This could be made configurable in the future. It also keeps track of which ones of the red, green and blue channels contained the unexpected values and indicates this in the error message returned to the student.

2. runFile
   This method runs the Python code in the specified file (if omitted, main.py is used)

under a restricted environment, using the specified input string as the content of stdin and capturing the text sent to stdout.

It returns a 2-tuple (studentLocals, output), where studentLocals is a dictionary containing the state of the local variables, functions, and class definitions after the program was executed, and output is the text that the program wrote to stdout.

3. assertSoundsSimilar
   This method is an assertion helper that compares two sounds to check if they are similar. It is very similar to the assertImagesSimilar helper method. If first compares the length of the sounds and then goes on to compare the sounds sample-by-sample. If some values are not within the tolerance limit, a descriptive error is returned to the student.

3.4 Modifications to Skulpt

Skulpt implements Python 2.7, and as mentioned in the previous section, we needed to support Python 3 syntax in Pythy. Completely shifting over to Python 3 would be an expensive and time consuming endeavor. Instead, we enumerated the most used Python 3 features in introductory computer science courses by trying out example programs from a Python 3 based textbook, The Practice of Computing Using Python [27]. We found that that 'print' was by far the most-used Python 3 feature. In Python 3, 'print' is no longer a keyword like it was upto Python 2.7. It is now a built-in function that supports two optional keyword arguments, 'sep' (separator) and 'end' (endmarker). So, we implemented this change in Skulpt by removing 'print' from the grammar and replacing it with a builtin-in function.

Chapter 4

Evaluation

To verify if all the API methods behave correctly, we used two layers of testing. The first was executing all 82 media computation examples from Chapter 2 to Chapter 8 of the course textbook in Pythy. The second layer was automated testing where we wrote and ran unit tests for each API method in both JavaScript and Python.

4.1 Systematically Validating the API through Example Programs

Tables 4.1, 4.2 and 4.3 list all the examples between Chapters 2 and Chapter 8 of the textbook, the chapters related to image and sound manipulation, and show that all these examples can be executed in Pythy. A total of 6 examples, 3 for image manipulation and 3 for sound manipulation, highlighted in purple in the tables, have been selected to showcase that all but one of the programs from the textbook can be run directly, with no changes, in Pythy. These programs were selected as together they cover a large set of the API methods. In some cases the main program is composed of many sub-programs, for example, the first program selected is Program 31, but it also needs programs 9, 11, 16 and 30 to work.

The listings in this section may have some lines highlighted in red or blue. The red highlights indicate that the line needs to be included only when the program is run in JES and the blue highlights indicate that the line should be included only in Pythy. Finally, next to each listing is the corresponding output in both Pythy and JES. As can be seen, 5 out of the 6 programs can be run with no changes.

Table 4.1: Example Programs from the Textbook

Program #	**Description**	**Works in Pythy**
1	Pick and Show a Picture	✓
2	Pick and Play a Sound	✓
3	Show a Specific Picture	✓
4	Play a Specific Sound	✓
5	Show the Picture File Whose Filename is Input	✓
6	Play the Sound File Whose Filename is Input	✓
7	Play a Sound File While Showing a Picture	✓
8	Show the Picture Provided as Input	✓
9	Reduce the Amount of Red in a Picture by 50%	✓
10	Increase the Red Component by 20%	✓
11	Clear the Blue Component from a Picture	✓
12	Making a Sunset	✓
13	Making a Sunset As Three Functions	✓
14	Lighten the Picture	✓
15	Darken the Picture	✓
16	Create the Negative of the Original Picture	✓
17	Convert to Grayscale	✓
18	Convert to Grayscale with Weights	✓
19	Lighten the Picture Using Nested Loops	✓
20	Mirror Pixels in a Picture Along a Vertical Line	✓
21	Mirror Pixels Horizontally, Top to Bottom	✓
22	Mirror Pixels Horizontally, Bottom to Top	✓
23	Mirror the Temple of Hephaestus	✓
24	Copying a Picture to a Canvas	✓
25	Copying a Picture to a Canvas Another Way	✓
26	Copy Elsewhere into the Canvas	✓
27	Cropping a Picture onto a Canvas	✓
28	Cropping the Face Into the Canvas Differently	✓
29	Creating a Collage	✓
30	A General Copy Function	✓
31	Improved Collage with the General Copy Function	✓
32	Rotating (Flipping) a Picture	✓
33	Rotating a Picture	✓
34	Scaling a Picture Down (Smaller)	✓
35	Scaling the Picture Up (Larger)	✓
36	Turning Katie into a Redhead	✓
37	Color Replacement in a Range	✓

Table 4.2: Example Programs from the Textbook (continued)

Program #	**Description**	**Works in Pythy**
38	Reduce Red-Eye	✓
39	Convert a Picture to Sepia Tones	✓
40	Posterizing a Picture	✓
41	Posterize to Two Gray Levels	✓
42	A Simple Blur	✓
43	Create a Simple Line Drawing Using Edge Detection	✓
44	Create a Simple Line Drawing Using Edge Detection	✓
45	Blending Two Pictures	✓
46	Subtract the Background and Replace It with a New One	✓
47	Chromakey: Replace All Blue with the New Background	✓
48	Chromakey, Shorter	✓
49	Draw Lines by Setting Pixels	✓
50	Adding a Box to a Beach	✓
51	An Example of Using Drawing Commands	✓
52	Gray Posterize to Two Levels, with else	✓
53	Draw the Gray Effect	✓
54	Draw the Picture in Figure 5.26	✓
55	Draw the Picture in Figure 5.27	✓
56	Increase an Input Sound's Volume by Doubling the Amplitude	✓
57	Decrease an Input Sound's Volume by Halving the Amplitude	✓
58	Change a Sound's Volume by a Given Factor	✓
59	Normalize the Sound to a Maximum Amplitude	✓
60	Set All Samples to Maximum Values	✓
61	Increase an Input Sound's Volume Using range	✓
62	Increase the Volume, Then Decrease It	✓
63	Merging Words Into a Single Sentence	✓
64	Splice the Preamble to Have United People	✓
65	Create a Sound Clip	✓
66	General Copy	✓
67	Using the General Clip and Copy	✓
68	Play the Given Sound Backwards (Reverse It)	✓
69	Mirror a Sound Front to Back	✓
70	Blending Two Sounds	✓
71	Make a Sound and a Single Echo of It	✓
72	Creating Multiple Echoes	✓

Table 4.3: Example Programs from the Textbook (continued)

Program #	**Description**	**Works in Pythy**
73	Creating a Chord	✓
74	Double the Frequency of a Sound	✓
75	Halve the Frequency	✓
76	Shifting the Frequency of a Sound: BROKEN!	✓
77	Shifting the Frequency of a Sound	✓
78	Playing a Sound in a Range of Frequencies	✓
79	Generate a Sine Wave at a Given Frequency and Amplitude	✓
80	Add Two Sounds Together	✓
81	Square Wave Generator for a Given Frequency and Amplitude	✓
82	Generate Triangular Waves	✓

4.1.1 Program 9, 11, 16, 30 & 31: Creating a Collage

This program is a cleaner version of Program 29: Creating a Collage. It involves creating a collage by pasting manipulated versions of pictures of flowers next to each other. The manipulations are—creating a negative, decreasing the red and removing the blue component from a picture. The pasting is done by setting the colors of the pixels of a blank canvas to the color of the corresponding pixel in the picture to be pasted. The images of the flowers were taken from the mediacomputation.org website and are available at http://coweb.cc.gatech.edu/mediaComp-teach/uploads/1/mediasources-4ed.zip.

This program was chosen because it is a combination of five programs from the textbook, and it demonstrates the use of many API methods like getPixels, getRed, getGreen, getBlue, makeColor, setColor, setBlue, setRed, getWidth, getHeight, getPixel, getColor, makePicture, __str__, makeEmptyPicture, and writePictureTo.

The only change necessary to run this program in Pythy is to explicity import the media module. It is trivial to configure Pythy to include this module before executing student code. However, such an implicit import may cause problems due to name conflicts when non-media-computation programs are executed. In cases where Pythy is being used only for media computation related work, it is thus possible to implicitly import this module and not require the student to do so. Thus, we do not consider this a change that prevents students from running media-computation programs in Pythy as-is.

Listing 4.1: Program 9, 11, 16, 30, & 31: Creating a Collage

```
1  from media import *
2
3  def negative(picture):
4    for px in getPixels(picture):
5      red = getRed(px)
6      green = getGreen(px)
7      blue = getBlue(px)
8      negColor = makeColor(255 - red, 255 - green, 255 - blue)
9      setColor(px, negColor)
10
11 def clearBlue(picture):
12   for p in getPixels(picture):
13     value = getBlue(p)
14     setBlue(p, 0)
15
16 def decreaseRed(picture):
17   for p in getPixels(picture):
18     value = getRed(p)
19     setRed(p, value * 0.5)
20
21 def copy(source, target, targX, targY):
22   targetX = targX
23   for sourceX in range(0, getWidth(source)):
24     targetY = targY
25     for sourceY in range(0, getHeight(source)):
26       px = getPixel(source, sourceX, sourceY)
27       tx = getPixel(target, targetX, targetY)
28       setColor(tx, getColor(px))
29       targetY = targetY + 1
30     targetX = targetX + 1
31
32 def createCollage2():
33   flower1 = makePicture(getMediaPath("flower1.jpg"))
34   print(flower1)
35   flower2 = makePicture(getMediaPath("flower2.jpg"))
36   print(flower2)
37   canvas = makeEmptyPicture(500, 110)
38   print(canvas)
39   copy(flower1, canvas, 0,
40       getHeight(canvas) - getHeight(flower1) - 5)
41   copy(flower2, canvas, 100,
42       getHeight(canvas) - getHeight(flower2) - 5)
43   negative(flower1)
44   copy(flower1, canvas, 200,
```

```
45          getHeight(canvas) - getHeight(flower1) - 5)
46      clearBlue(flower2)
47      copy(flower2, canvas, 300,
48          getHeight(canvas) - getHeight(flower2) - 5)
49      decreaseRed(flower1)
50      copy(flower1, canvas, 400,
51          getHeight(canvas) - getHeight(flower1) - 5)
52      return canvas
53
54  setMediaPath("/path-or-url/to/media-folder-or-media-library")
55  writePictureTo(createCollage2(), "createCollage2.jpg")
```

(a) Pythy

(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 4.1: The output of Program 31: Creating a Collage

4.1.2 Program 18 & 39: Convert a Picture to Sepia Tones

This program converts a color picture into a sepia-tinted one by first converting it to grayscale and then changing the red and blue value depending on the original red value of the pixel. It uses the API methods getPixels, getRed, getBlue, getGreen, setColor, makeColor, setBlue, setRed, makePicture and writePictureTo.

This program was chosen to reinforce the fact that Pythy can be used as a drop-in for JES for image manipulation and to show that Pythy produces results similar to JES even with pictures that have a wider range of colors like landscape photographs.

Figure 4.2: The original picture - The pond facing the CS Department at Virginia Tech

(a) Pythy

(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 4.3: The output of Program 39: Convert a Picture to Sepia Tones

Listing 4.2: Program 18 & 39: Convert a Picture to Sepia Tones

```
1  from media import *
2
3  def grayScaleNew(picture):
4    for px in getPixels(picture):
5      newRed = getRed(px) * 0.299
6      newGreen = getGreen(px) * 0.587
7      newBlue = getBlue(px) * 0.114
8      luminance = newRed + newGreen + newBlue
9      setColor(px, makeColor(luminance, luminance, luminance))
10
11 def sepiaTint(picture):
12   grayScaleNew(picture)
13
14   for p in getPixels(picture):
15     red = getRed(p)
16     blue = getBlue(p)
17
18     if(red < 63):
19       red = red * 1.1
20       blue = blue * 0.9
21
22     if(red > 62 and red < 192):
23       red = red * 1.15
24       blue = blue * 0.85
25
26     if(red > 191):
27       red = red * 1.08
28       if(red > 255):
29         red = 255
30       blue = blue * 0.93
31
32     setBlue(p, blue)
33     setRed(p, red)
34
35 setMediaPath("/path-or-url/to/media-folder-or-media-library")
36 picture = makePicture("beach.jpg")
37 sepiaTint(picture)
38 writePictureTo(picture, "sepiaTone.jpg")
```

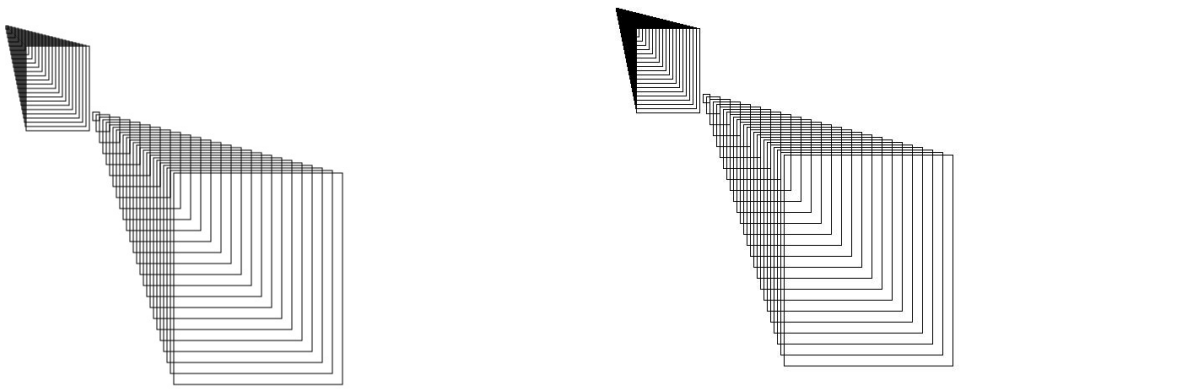
4.1.3 Program 55: Drawing a Picture

This program draws many rectangles of slightly varying sizes close to each other to create two rectangular cones. It uses the API methods makeEmptyPicture, addRect, show and writePictureTo.

This program was chosen because it demontrates the drawing function addRect as opposed to manipulating pixels of an existing picture through setColor or other set functions.

Listing 4.3: Program 55: Drawing a picture

```
1  from media import *
2
3  def coolPic2():
4    canvas = makeEmptyPicture(640, 480)
5    for index in range(25, 0, -1):
6      addRect(canvas, index, index, index*3, index*4)
7      addRect(canvas, 100+index*4, 100+index*3, index*8, index*10)
8    show(canvas)
9    return canvas
10
11 setMediaPath("/path-or-url/to/media-folder-or-media-library")
12 writePictureTo(coolPic2(), "coolPic2.jpg")
```



(a) Pythy

(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 4.4: The output of Program 55: Drawing a picture

4.1.4 Program 65, 66 & 67: Using the General Clip and Copy

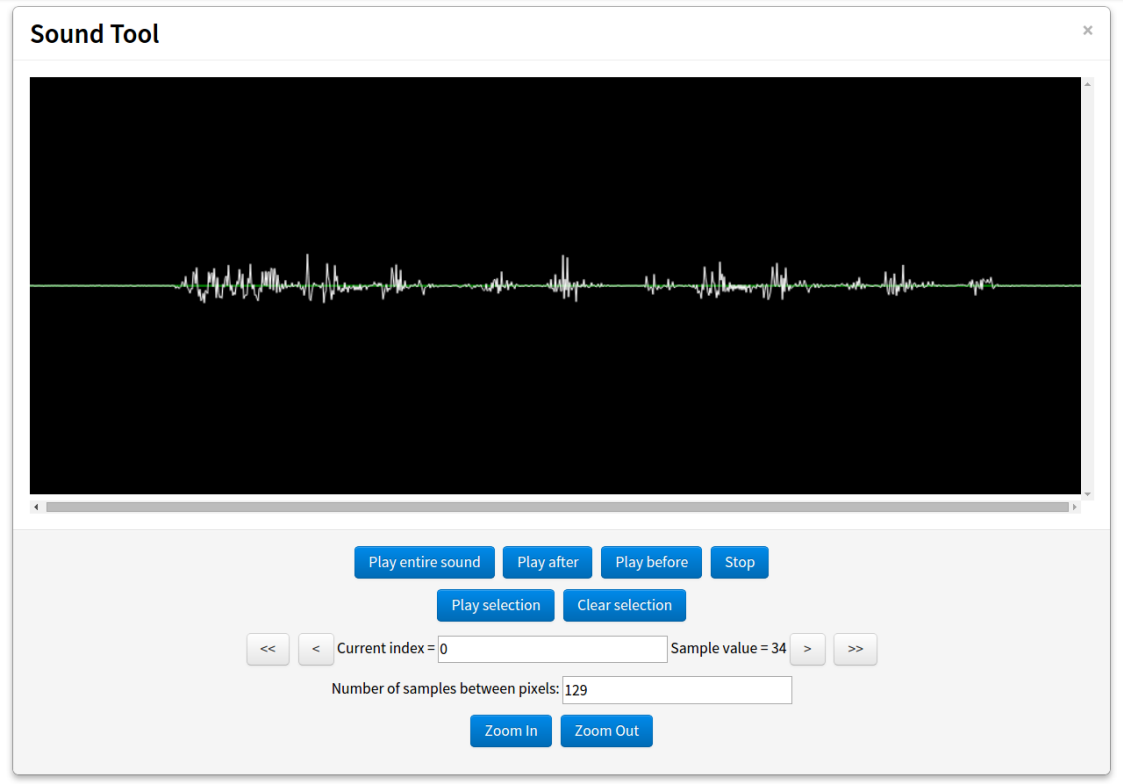
This is an improved version of Program 64 due to the factoring of clipping and copying into separate methods. It produces a new sound by inserting a clipped part of the original sound that says the word 'united' into the original sound at another place. It uses the API methods makeEmptySound, getSampleValueAt, setSampleValueAt, getLength, makeSound, and openSoundTool.

This is the only program that needs changes before being run in Pythy but the changes are in the values passed to the API methods, not in the API methods themselves. This is due to a limitation in the current implementation of the Web Audio API [3]. By default, when an audio file is decoded into a JavaScript object, it is automatically resampled to 44100Hz and there is currently no option to disable this automatic resampling. As the sampling rate of all audio files used in the textbook examples is 22050Hz, we have to update the sample number appropriately for the example to work in Pythy. In fact we simply have to double the sample numbers as the sampling rate for Pythy is twice that of JES.

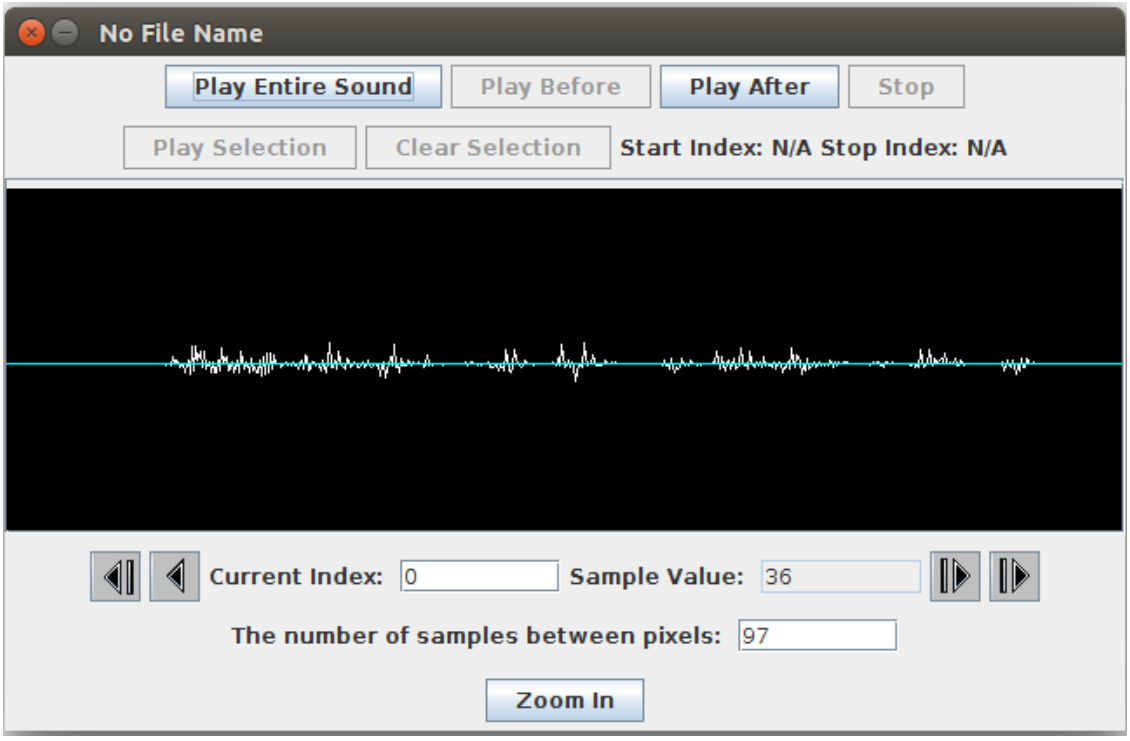
The JavaScript representation of the decoded audio file is an array of 32 bit floating point data with values ranging from -1 to 1. The media computation API however, expects samples to take values between -32768 to 32767. To satisfy this requirement, we map values in the range -1 to 1 to the range -32767 to 32767 when setting and getting sample values. This, along with the difference in sampling rates, sometimes leads to the sample values in JES not matching those in Pythy, as is seen in Figure 4.5.

Listing 4.4: Program 65, 66 & 67 : Using the General Clip and Copy

```
1  from media import *
2
3  def clip(source, start, end):
4    target = makeEmptySound(end - start)
5    targetIndex = 0
6    for sourceIndex in range(start, end):
7      sourceValue = getSampleValueAt(source, sourceIndex)
8      setSampleValueAt(target, targetIndex, sourceValue)
9      targetIndex += 1
10   return target
11
12 def copy(source, target, start):
13   targetIndex = start
14   for sourceIndex in range(0, getLength(source)):
15     sourceValue = getSampleValueAt(source, sourceIndex)
16     setSampleValueAt(target, targetIndex, sourceValue)
17     targetIndex += 1
18
19 def createNewPreamble():
20   file = getMediaPath("preamble10.wav")
21   preamble = makeSound(file)
22   united = clip(preamble, 33414, 40052)
23   united = clip(preamble, 66818, 80104)
24   start = clip(preamble, 0, 17407)
25   start = clip(preamble, 0, 34814)
26   end = clip(preamble, 17408, 55510)
27   end = clip(preamble, 34815, 111020)
28   len = getLength(start) + getLength(united) + getLength(end)
29   newPre = makeEmptySound(len)
30   copy(start, newPre, 0)
31   copy(united, newPre, 17407)
32   copy(united, newPre, 34815)
33   copy(end, newPre, getLength(start) + getLength(united))
34   return newPre
35
36 setMediaPath("/path-or-url/to/media-folder-or-media-library")
37 openSoundTool(createNewPreamble())
```



(a) Pythy



(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 4.5: The output of Program 67: Using the General Clip and Copy

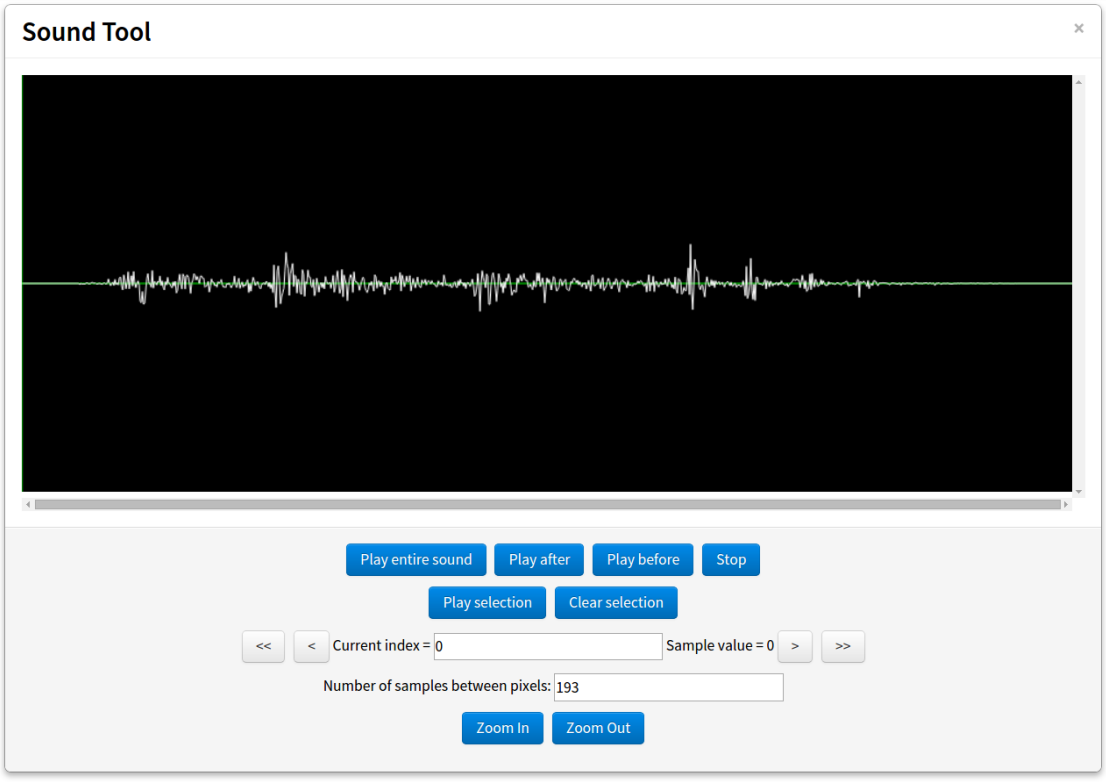
4.1.5 Program 72: Creating Multiple Echoes

This program creates the effect of echoes. It adds *num* echoes, each at a delay of *delay*∗*num* samples after the sound starts playing. Both *num* and *delay* are supplied by the caller. It uses the API methods makeSound, getLength, makeEmptySound, getSampleValueAt, setSampleValueAt, play and openSoundTool.

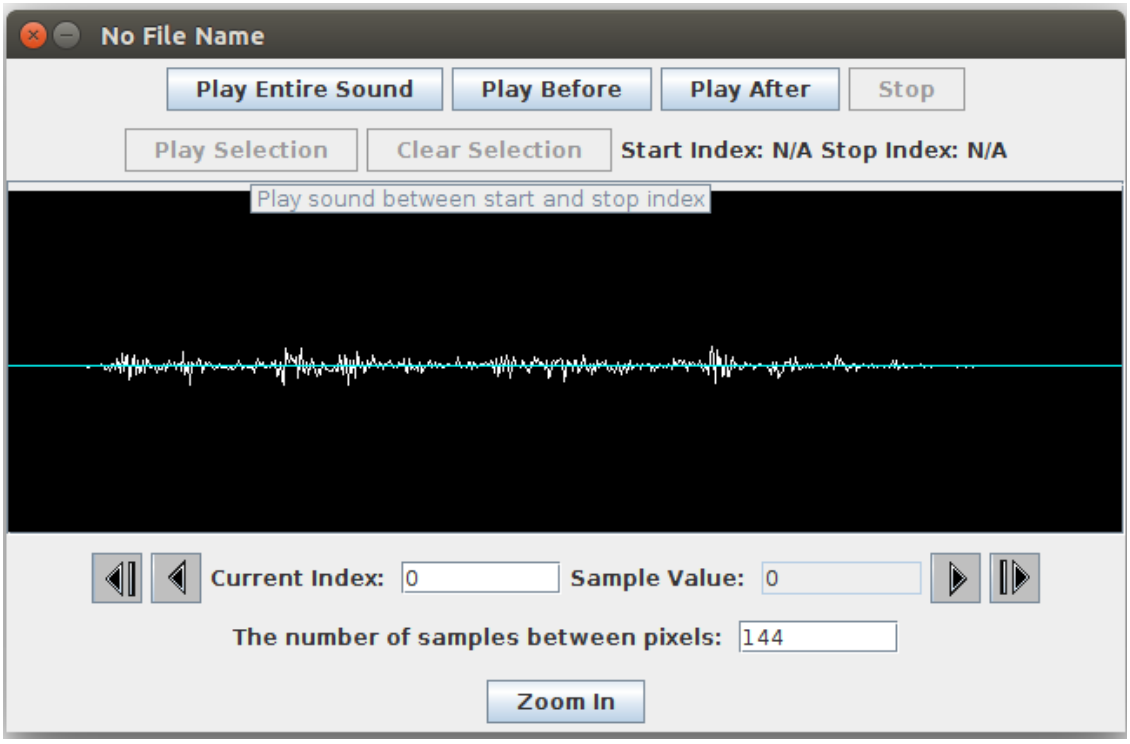
This program was chosen to demonstrate that the result is correct and the sound maintains its integrity even after repeated modifications of the original sound.

Listing 4.5: Program 72: Creating Multiple Echoes

```
1  from media import *
2
3  def echoes(sndFile, delay, num):
4    s1 = makeSound(sndFile)
5    ends1 = getLength(s1)
6    ends2 = ends1 + (delay * num)
7    s2 = makeEmptySound(ends2)
8
9    echoAmplitude = 1.0
10   for echoCount in range(1, num):
11     echoAmplitude = echoAmplitude * 0.6
12     for posns1 in range(0, ends1):
13       posns2 = posns1 + (delay * echoCount)
14       values1 = getSampleValueAt(s1, posns1) * echoAmplitude
15       values2 = getSampleValueAt(s2, posns2)
16       setSampleValueAt(s2, posns2, int(values1 + values2))
17   play(s2)
18   return s2
19
20 setMediaPath("/path-or-url/to/media-folder-or-media-library")
21 openSoundTool(echoes(getMediaPath("test.wav"), 5000, 5))
```



(a) Pythy



(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

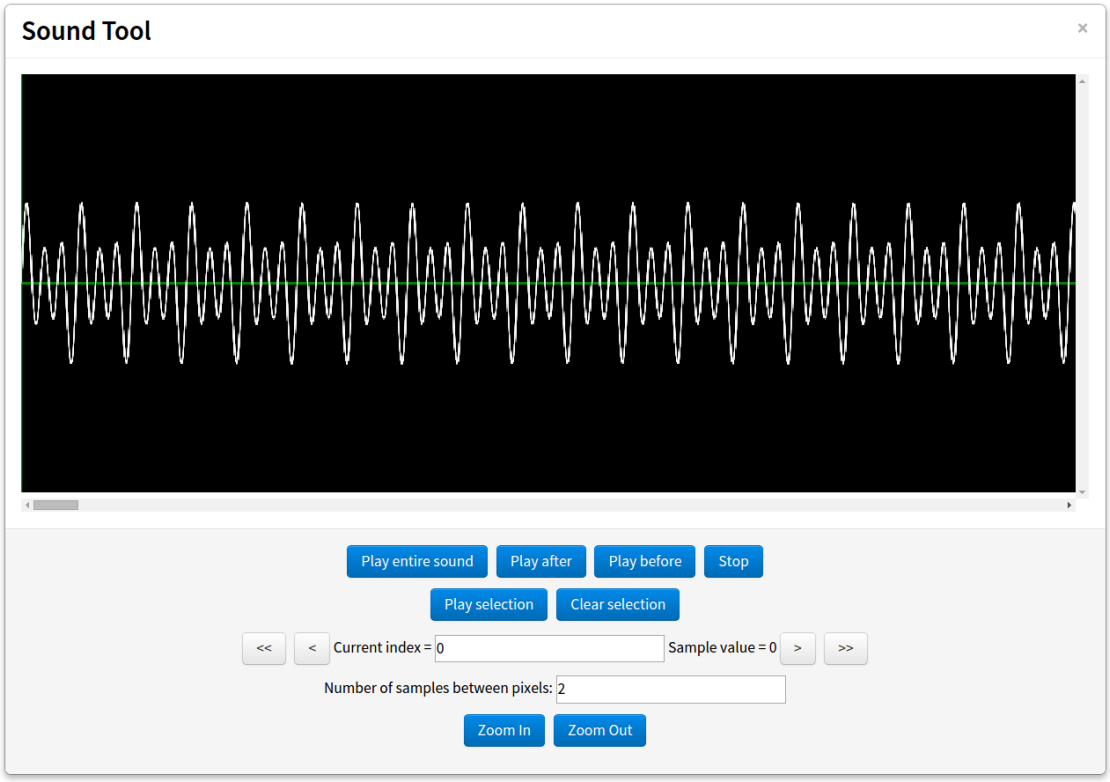
Figure 4.6: The output of Program 72: Creating Multiple Echoes

4.1.6 Program 79 & 80: Adding Sine Waves

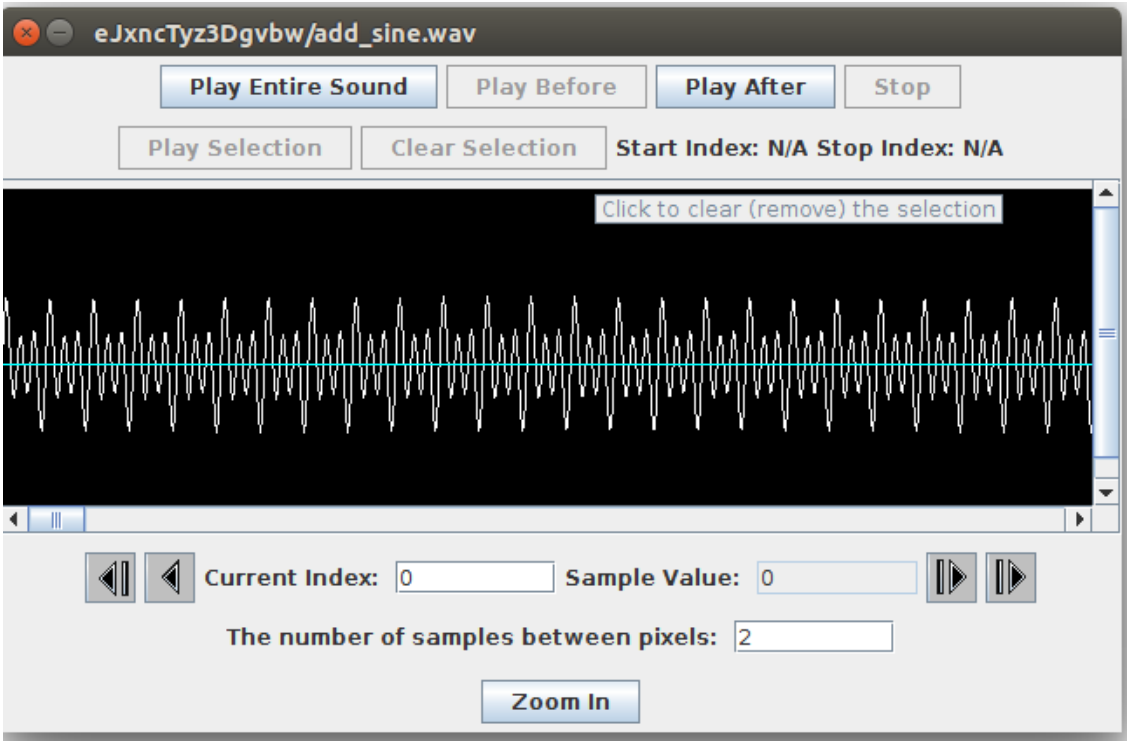
This program generates sine waves of different frequencies and amplitudes and adds them. It uses the API methods getLength, getSampleValueAt, setSampleValueAt, makeSound, getSamplingRate. As students only have permission to store media under their own media library url, only the filename is taken as the argument in the write methods and not the url to the filename. This program was chosen to demonstrate the generative aspects of the API like building a sound from scratch.

Listing 4.6: Program 79 & 80: Adding sine waves

```
1  from media import *
2  from math import *
3
4  def addSounds(sound1, sound2):
5    for index in range(0, getLength(sound1)):
6      s1Sample = getSampleValueAt(sound1, index)
7      s2Sample = getSampleValueAt(sound2, index)
8      setSampleValueAt(sound2, index, s1Sample + s2Sample)
9
10 def sineWave(freq, amplitude):
11   mySound = "sec1silence.wav"
12   buildSin = makeSound(mySound)
13   sr = getSamplingRate(buildSin)
14   interval = 1.0 / freq
15   samplesPerCycle = interval * sr
16   maxCycle = 2 * pi
17
18   for pos in range(0, getLength(buildSin)):
19     rawSample = sin((pos / samplesPerCycle) * maxCycle)
20     sampleVal = int(amplitude * rawSample)
21     setSampleValueAt(buildSin, pos, sampleVal)
22
23   return buildSin
24
25 setMediaPath("/path-or-url/to/media-folder-or-media-library")
26 f440 = sineWave(440, 2000)
27 f880 = sineWave(880, 4000)
28 f1320 = sineWave(1320, 8000)
29 addSounds(f880, f440)
30 addSounds(f1320, f440)
31
32 writeSoundTo(f440, getMediaPath("add_sine.wav"))
33 writeSoundTo(f440, "add_sine.wav")
34 openSoundTool(makeSound(getMediaPath("add\_sine.wav")))
```



(a) Pythy



(b) JES, https://code.google.com/p/mediacomp-jes/downloads/list

Figure 4.7: The output of Program 79 and 80: Adding sine waves

4.2 Automated Testing

We use the Mocha JavaScript framework along with the Chai assertion library and Sinon library for mocks, spies and stubs for unit testing the JavaScript media computation API. On the server-side we used unittest, the Python unit testing framework.

In total there are 312 unit tests on the client-side, 205 for images and 107 for sounds, and 183 unit tests on the server-side, 124 for images and 59 for sounds. There are fewer unit tests on the server-side because they are written in pure Python and hence no argument-checking tests are needed.

Behavior Driven Development or BDD [26] tests focuses on testing behavior instead of the implementation. Each test includes a short description of the behavior or scenario being tested along with the setup, execution, validation and cleanup code. This clearly documents the behavior that is covered by our implementation of the API. An example of a BDD-style test is given in Listing 4.7 and the output produced by the Mocha test runner for the Color class is given in Figure 4.8.

Listing 4.7: BDD-style tests for the makeColor method

```
1  describe("makeColor", function () {
2    it("should take the red value and use it as gray if green " +
3       "and blue values are not provided", function () {
4      var execFunc, color;
5
6      execFunc = function () { mod.makeColor(); };
7      assert.throws(execFunc, Sk.builtin.TypeError,
8       "makeColor() takes between 1 and 3 " +
9       "positional arguments but 0 were given");
10
11     execFunc = function () { mod.makeColor(20); };
12     assert.doesNotThrow(execFunc, Sk.builtin.TypeError);
13
14     color = mod.makeColor(123);
15     assert.strictEqual(color._red, 123);
16     assert.strictEqual(color._green, 123);
17     assert.strictEqual(color._blue, 123);
18   });
19
20   it("should optionally take the green value", function () {
21     var execFunc, color;
22
23     execFunc = function () { mod.makeColor(20, 55); };
24     assert.doesNotThrow(execFunc, Sk.builtin.TypeError);
25
26     color = mod.makeColor(20, 55);
27     assert.strictEqual(color._red, 20);
28     assert.strictEqual(color._green, 55);
29     assert.strictEqual(color._blue, 20);
```

```
30    });
31
32    it("should optionally take the blue value", function () {
33      var execFunc, color;
34
35      execFunc = function () { mod.makeColor(20, 55, 75); };
36      assert.doesNotThrow(execFunc, Sk.builtin.TypeError);
37
38      color = mod.makeColor(20, 55, 75);
39      assert.strictEqual(color._red, 20);
40      assert.strictEqual(color._green, 55);
41      assert.strictEqual(color._blue, 75);
42    });
43
44    it("should take a color and return a new color with the " +
45        "same rgb values",
46        function () {
47      var color;
48
49      color = mod.makeColor(20, 14, 106);
50      newColor = mod.makeColor(color);
51      assert.notStrictEqual(color, newColor);
52      assert.strictEqual(newColor._red, 20);
53      assert.strictEqual(newColor._green, 14);
54      assert.strictEqual(newColor._blue, 106);
55    });
56  });
```

As we had already documented the behavior of the code on the client-side, we did not use the BDD-style for Python tests. They are just PyUnit tests.

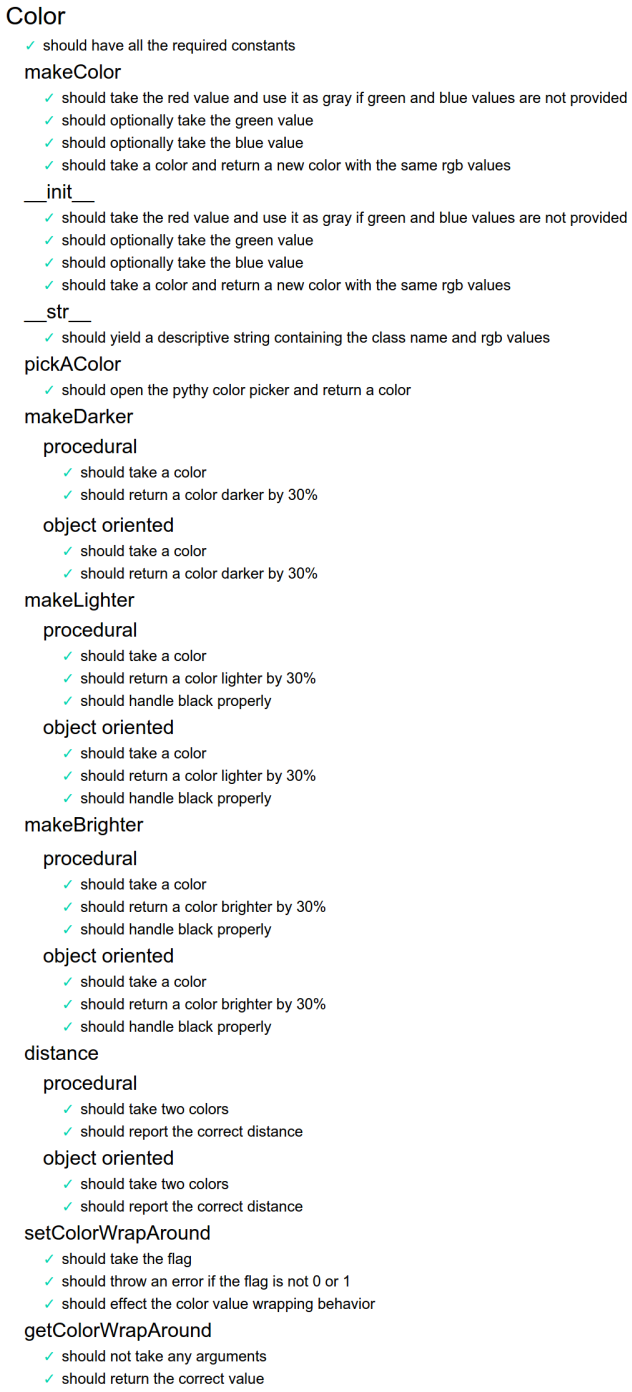


Figure 4.8: Output of Test Runner for class Color

4.3 Server-side Grading

In addition to verifying the client-side behavior, we also had to verify if all the API methods on the server-side behaved as expected. For this purpose we selected two exercise problems from the textbook, one for image manipulation and one for sound manipulation, and wrote Pythy-style tests for them. The problem statements are presented next to their test code.

4.3.1 Examples

Problem 5.1

Write a function called *changeColor* that takes as input a picture and an amount to increase or decrease a color by and a number 1 (for red), 2 (for green), or 3 (for blue). The amount will be a number between -0.99 and 0.99.

- $changeColor(pict, -0.10, 1)$ should decrease the amount of red in the picture by 10%.
- $changeColor(pict, 0.30, 2)$ should increase the amount of green in the picture by 30%.
- $changeColor(pict, 0, 3)$ should do nothing at all to the amount of blue (or red or green) in the picture.

Listing 4.8: Tests for problem 5.1

```
1  import pythy
2  from media import *
3
4  class P3ReferenceTests(pythy.TestCase):
5    def setUp(self):
6      self.filename = 'http://url/to/image.png'
7      self.picture = Picture(self.filename)
8      (self.studentLocals, self.output) = self.runFile()
9
10   def testRedIncrease(self):
11     self.helper(0.50, 1)
12
13   def testRedDecrease(self):
14     self.helper(-0.50, 1)
15
16   def testGreenIncrease(self):
17     self.helper(0.75, 2)
18
19   def testGreenDecrease(self):
```

```
20      self.helper(-0.75, 2)
21
22    def testBlueIncrease(self):
23      self.helper(0.30, 3)
24
25    def testBlueDecrease(self):
26      self.helper(-0.30, 3)
27
28    def helper(self, amount, rgb):
29      resultPicture =
30        self.studentLocals['changeColor'](self.picture, amount, rgb)
31      self.assertImagesSimilar('',
32                               self.expected_picture(amount, rgb),
33                               resultPicture)
34
35    def expected_picture(self, amount, rgb):
36      expected = Picture(self.filename)
37      if rgb is 1:
38        getMethod = getRed
39        setMethod = setRed
40      elif rgb is 3:
41        getMethod = getBlue
42        setMethod = setBlue
43      elif rgb is 2:
44        getMethod = getGreen
45        setMethod = setGreen
46
47      for pixel in getPixels(expected):
48        color = getMethod(pixel)
49        changedColor = int(color + color * amount)
50        setMethod(pixel, changedColor)
51      return expected
```

Problem 7.13

Write a function that interleaves two sounds.

It starts with 2 seconds of the first sound and then 2 seconds of the second sound. Then, it continues with the next 2 seconds of the first sound and the next 2 seconds of the second sound and so on until both sounds have been fully copied to the target sound.

Listing 4.9: Tests for problem 7.13

```
import pythy
from media import *
from unittest import mock

class P3ReferenceTests(pythy.TestCase):
  def setUp(self):
    self.filename1 = 'http://url/to/sound1.wav'
    self.filename2 = 'http://url/to/sound2.wav'
    self.sound1 = Sound(self.filename1)
    self.sound2 = Sound(self.filename2)
    (self.studentLocals, self.output) = self.runFile()

  def testCorrectInterleave(self):
    resultSound =
      self.studentLocals['interleave'](self.sound1, self.sound2)
    self.assertSoundsSimilar(self.expected_sound(), resultSound)

  def expected_sound(self):
    sound1 = Sound(self.filename1)
    sound2 = Sound(self.filename2)
    expected = makeEmptySound(sound1.getLength() +
                              sound2.getLength())
    samplesPerTwoSeconds = expected.getSamplingRate() * 2
    s1 = 0
    s2 = 0
    i = 0
    while(i < getLength(expected)):
      for j in range(0, samplesPerTwoSeconds):
        if(s1 >= getLength(sound1)):
          break
        value = getSampleValueAt(sound1, s1)
        setSampleValueAt(expected, i, value)
        s1 += 1
        i += 1
      for j in range(0, samplesPerTwoSeconds):
        if(s2 >= getLength(sound2)):
          break
        value = getSampleValueAt(sound2, s2)
        setSampleValueAt(expected, i, value)
        s2 += 1
        i += 1
    return expected
```

Chapter 5

Conclusions and Future Work

By integrating the image and sound portion of the media computation API with Pythy, we have moved one step forward towards making Pythy a drop-in replacement for JES. Next, we discuss the contributions of this work and ways in which it can be improved upon.

5.1 Contributions

All methods required for sound and image manipulation, as part of the media computation curriculum, are now available in Pythy. Students can now follow the course textbook and execute the examples as-is in Pythy and instructors do not need expend extra effort to account for deviations from the API used in the textbook. In summary:

1. **Problem: How to add support for image manipulation in Pythy?**
   By fixing the problems with the existing implementation—boxing/unboxing of parameters, incorrect return values, accounting for JS-specific behavior, adding object-oriented counterparts of methods, using the HTML5 Canvas and its associated JavaScript API to implement the remaining methods on the client-side, and using the pillow module to implement all the methods of the image API on the server-side, we were able to completely integrate the image API with Pythy.

   The existence and behavior of each method was rigorously tested using automated tests and by running all image manipulation related programs from the textbook in Pythy.

2. **Problem: How to add support for sound manipulation in Pythy?**
   By fixing the problems with the existing implementation, using the Web Audio API to implement the remaining methods on the client-side, and using the wave module to implement all the methods of the sound API on the server side, we were able to completely integrate the sound API with Pythy.

The existence and behavior of each method was rigorously tested using automated tests and by running all sound manipulation related programs from the textbook in Pythy.

3. **Problem: How to provide support for grading media computation based assignments in Pythy?**
   This was achieved by implementing the API methods, from scratch, on the server-side and by using the helper methods runFile, assertSoundsSimilar and assertImagesSimilar for testing.

4. **What are the changes required in Skulpt, the library that helps run Python in the browser, in order to be able to run media computation-based programs in Pythy?**
   We implemented the Python 3 'print' as a built-in method that accepts keyword arguments to make Pythy comptabile with basic Python 3 usage.

5.2 Future Work

An obvious extension to this work would be to incorporate the video API into Pythy. This could be done using Flash or the HTML5 video element along with the HTML5 canvas element.

We noticed that using the media computation methods on Pythy is much slower as compared to running them on JES. This is largely due to the function calls needed to convert to and from JavaScript objects and their corresponding Skulpt-based Python representations. Optimizing these calls will improve the performance of running media manipulation code on Pythy. This will help run tests in faster and also present the result of students' code faster.

In order for Pythy to be completely in tune with JES, it must behave consistently when errors occur, not just when programs behave as expected. Although we have verified that media-computation programs behave as expected in Pythy, we still need to evaluate Pythy's behavior when incorrect code is executed. This involves raising the right exceptions with matching messages in some cases and ignoring the errors and purposely doing the wrong thing (for educational purposes) in others.

It would also be a good idea to look at standard testing libraries for other languages as well as customized extensions to these libraries which are specifically used to evaluate student-written code, like Web-CAT, and try to integrate features from them into Pythy's grading infrastructure.

Finally, adding an interactive method reference or 'help' feature for media-computation methods, similar to the one present in JES, would be very beneficial to students.

Bibliography

[1] Alice. http://www.alice.org/.

[2] Concurrency model and event loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[3] Github issue 30 for The Web Audio API - decodeaudiodata: option to disable automatic resampling to context rate. https://github.com/WebAudio/web-audio-api/issues/30.

[4] JES - Jython Environment for Students. http://code.google.com/p/mediacomp-jes/.

[5] Jython 2.7.0 final released! http://fwierzbicki.blogspot.fi/2015/05/jython-270-final-released.html.

[6] The Jython Project. http://www.jython.org/.

[7] Jython roadmap. https://wiki.python.org/jython/RoadMap.

[8] Maximum size of a canvas element. http://stackoverflow.com/questions/6081483/maximum-size-of-a-canvas-element.

[9] Online python tutor. http://www.pythontutor.com/.

[10] Picture functions in JES. https://github.com/gatech-csl/jes/blob/master/jes/help/JESHelp/7_Picture_Functions.html.

[11] Python imaging library (fork) 2.8.2. https://pypi.python.org/pypi/Pillow/2.8.2.

[12] pythonanywhere: A python learning environment with everything ready to go. https://www.pythonanywhere.com/details/education.

[13] Runestone interactive: Interactive python. http://interactivepython.org/.

[14] Scratch - imagine, program, share. https://scratch.mit.edu/.

[15] Skulpt. http://www.skulpt.org/.

[16] Sound functions in JES. `https://github.com/gatech-csl/jes/blob/master/jes/help/JESHelp/5_Sound_Functions.html`.

[17] wave - read and write wav files. `https://docs.python.org/2/library/wave.html`.

[18] HTML 5, A vocabulary and associated APIs for HTML and XHTML. `http://www.w3.org/TR/2008/WD-html5-20080122/`, January 2008.

[19] Web Audio API. `http://www.w3.org/TR/2011/WD-webaudio-20111215/`, December 2011.

[20] Diana I. Cordova and Mark R. Lepper. Intrinsic motivation and the process of learning: Beneficial effects of contextualization, personalization, and choice. *Journal of Educational Psychology*, 88(4):715—730, December 1996.

[21] Stephen H. Edwards, Daniel S. Tilden, and Anthony Allevato. Pythy: Improving the introductory python programming experience. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 641–646, New York, NY, USA, 2014. ACM.

[22] A. Forte and M. Guzdial. Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses. *IEEE Transactions on Education*, 48(2):248–253, 2005.

[23] Mark Guzdial. A media computation course for non-majors. *SIGCSE Bull.*, 35(3):104–108, June 2003.

[24] Mark J. Guzdial and Barbara Ericson. *Introduction to Computing and Programming in Python, A Multimedia Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2009.

[25] Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the effectiveness of a new instructional approach. *SIGCSE Bull.*, 36(1):75–79, March 2004.

[26] Dan North. Introducing BDD. `http://dannorth.net/introducing-bdd/`.

[27] William F. Punch and Richard Enbody. *The Practice of Computing Using Python*. Addison-Wesley Publishing Company, USA, 1st edition, 2010.

[28] Susan Reardon and Brendan Tangney. Smartphones, studio-based learning, and scaffolding: Helping novices learn to program. *Trans. Comput. Educ.*, 14(4):23:1–23:15, December 2014.

[29] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, November 2009.

[30] Beth Simon, Päivi Kinnunen, Leo Porter, and Dov Zazkis. Experience report: Cs1 for majors with media computation. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 214–218, New York, NY, USA, 2010. ACM.

[31] Allison Elliott Tew, Charles Fowler, and Mark Guzdial. Tracking an innovation in introductory cs education from a research university to a two-year college. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 416–420, New York, NY, USA, 2005. ACM.

[32] Daniel S. Tilden. Design and evaluation of a web-based programming tool to improve the introductory computer science experience. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, 2013.