Strong vs. Weak Typing
and
Scripting Languages

Due: April 8th, 2004

Team #2:
Leah Bidlake
Alastair Grant
David Hirtle

Table of Contents

1 Introduction

This report is an evaluation of the advantages and disadvantages of strong and weak typing for scripting languages. First, scripting languages are defined and their differences from traditional programming languages are highlighted. Then, general characteristics of strongly and weakly typed languages are outlined. Finally, the effects of strong and weak typing on scripting languages are discussed as observed through the development of a student management system in both Python, a strongly typed scripting language, and Perl, a weakly typed scripting language. Comparing the effects of strong or weak typing on the development of this application reveals significant detail concerning the advantages and disadvantages of each system of typing.

2 Scripting Languages

Scripting languages are used to write scripts, which are programs that are can be read and modified by the user. Scripts generally execute simple operations or control the operation of other programs. In essence, scripts could be written by practically any programming language, and scripting languages could be used to develop large-scale applications. However, many programming languages are not appropriate for writing scripts because they do not adequately meet the user readable and user modifiable condition. Conversely, many scripting languages are not adequate for increasingly complex applications (6).

One of the defining characteristics of scripting languages is that they are dynamically typed and so, as a side-effect, they are not compiled, but interpreted at run-time so that instructions can be executed immediately. This results in virtually no compile time, one of the major advantages of scripting languages. When programming in scripting languages, variables, functions, and methods typically do not require type declarations. This is because there are automated conversions or equivalence between

types, particularly between strings and other types (6). The disadvantage of scripting languages is that the programmer does not typically have the same ability to optimize a program for speed or memory usage as they would in traditional programming languages (7).

Code written in scripting languages is easier to reorganize to improve its internal structure (5). It is much easier to change a program that was written in a dynamically typed language then a “type safe” language. Dynamically typed languages such as scripting languages also have fewer problems with redeployment after changes are made to the code (5). Some scripting languages such as Perl support the change of scripts into programs (4, p.370). In this respect scripts can be thought of as a work in progress, and when they are fully developed they become programs.

Scripting languages can also be easily integrated with larger systems. Because of this scripting languages are sometimes referred to as glue languages which hold more complex systems together (6).

3 Strong vs. Weak Typing

A strongly typed language is one in which each name in a program has a single type associated with it and that type is known at compile time (4, p.194). Simply put, a programming language is strongly typed if type errors are always detected. To accomplish this, a strongly typed language must be able to detect when a variable is being misused, whether it is at compile-time or run-time. Compile-time checking is done on all variables that can contain only a single type, for example integers or structures in C. Run-time checking happens for any variables that could contain more than one type, for example unions in C.

A completely strongly typed language does not allow a floating-point number to be added to an integer because they are of different types and the compiler does not know

what the resulting type should be. For example, consider this code from a completely strongly typed language:

```
int a = 5;
float b = 5;
float c = b + a;  //causes type error: float cannot be added to int
```

Most strongly typed languages such as Java get around this by using coercion, forcing a variable to be another type. When the expression is encountered, the integer is coerced into a floating-point number so that it can be added to the other floating-point value. This is usually what the programmer wants to do anyway, but it removes the error detection provided by strong typing. Strong typing allows the compiler to optimize the code because it knows what type to make each variable. It also allows a programmer to express exactly what they want to do and increases the readability of the code.

Weak typing is considered a "friendlier" version of strong typing because it catches fewer errors during compilation. Many type rules do exist but there are also many exceptions or mechanisms in place to circumvent these rules. For example, the large number of coercions used by C and C++ lead it to be considered a weakly typed language. The above code snippet would be allowed with C and C++'s typing rules. Another weakly typed feature of these languages is that the compiler ignores "`typedefs`" when making type comparisons. For example, consider this C code:

```
typedef int StudentID;
sudentID s = 12345;
int a = s;  //This is allowed with C's typing, but is not allowed by
               a completely strongly typed language
```

Typically, the compiler is able to correctly infer the types of the variables and save the programmer from having to do so. If one were to consider how much time is spent defining variables of proper types, this leads to a considerable time saving. Weak typing does create more runtime errors and exceptions though, which leads to the view that weak typing is not as robust as strong typing.

Many people argue that the transition from C to C++ led to the growth of strong typing (1). In Kernigan and Ritchie's original C, integers and pointers used the same

underlying data type and could therefore be mixed. This led to vagueness in the code and subsequent errors. ANSI C and C++ resolved this by implementing a stronger typing system in response to these problems. Programmers who adopted C++ noticed a reduction in the number of errors because of these changes.

Typing is not a black and white system. A language is not simply strongly typed or weakly typed, but rather it can have features of both typing systems. For example, Java is fairly strongly typed, but it does use coercions, a weak typing feature, to allow mathematical operations on different numerical types. This is why Java is considered strongly typed, but not fully strongly typed. The fact that many languages are in this grey area is why there is so much debate among experts as to whether these languages are strongly typed or weakly typed.

4 Typing for Scripting Languages

Scripting languages succeed very well at masking the complexity of problems. This high-level nature has motivated their sometimes being referred to as “glue” languages whose primary function is to connect various (lower level) components together.

The process of writing code in scripting languages is far less burdensome than with traditional programming languages. A significant factor towards this simplicity is the fact that variables are only bound to data types when assigned values, as opposed to having types assigned at declaration time. In fact, this dynamic typing results in a great deal of flexibility because types are no longer locked in, as with statically typed languages. Declaring variables is not necessary, conversions are rarely needed, and generic programs are possible.

Perhaps most importantly, there are indications that scripting languages lead to increased productivity. For one thing, high level commands accomplish far more with much less. As a trivial example, consider the ubiquitous “hello world” program in Java.

```
class HelloWorld {
        public static void main (String args[]) {
         System.out.print("Hello World!");
        }
}
```

and then in TCL:

```
puts "Hello World!"
```

Clearly the TCL version is both quicker and easier to read, write and understand. This added expressiveness has its costs, however, to the point that many maintain that the benefit is not worth the potential loss. The greatest disadvantage of dynamically typed languages is that error detection is limited because type checking must occur at run time (also slowing down execution).

For the most part, those on both sides of the debate are willing to admit that, for certain applications, the other type of language may in fact be more natural. It is interesting to note that there is significant variation within scripting languages, as summarized below for a small subset:

Lua – dynamically and strongly typed

Ruby – dynamically and strongly typed

Python – dynamically and strongly typed

Perl – dynamically and weakly typed

TCL – dynamically and weakly typed

PHP – dynamically and weakly typed

While all dynamically typed, these languages are further divided up by their strong or weak typing. This second dimension of typing refers to how consistently type errors are detected as opposed to when they are detected, as is the case with dynamic versus static typing. In other words, values in strongly typed languages are independent and enforced, whereas in weakly typed languages the type of a value depends on processing.

We developed a system in both Python (a strongly typed scripting language) and Perl (a weakly typed scripting language) simultaneously. This system was a very simple student management system where a professor can create new student records and store the courses the students are enrolled in, update their grades and calculate their average grade over all courses. There were two goals for this experiment. The first was to determine if it is quicker to write a program in one language or the other. That is, is it faster to write in a strongly typed language or a weakly typed language? We gauged this in terms of development time between two programmers of similar skill. Secondly and most importantly, was to determine how many errors would be detected during run-time based on the typing used by the language. The Perl version of the program should be developed quicker and with fewer typing problems than the Python version based on the information presented.

4.1 Development of Application in Perl

The Perl version of the program (see Appendix A-1) was written in approximately 5 hours. This included planning, development and testing. This did not include time taken to learn the language. There were very few typing related problems encountered while developing the program. Perl has a single type of single-value variable called a scalar. This type can store strings, integers or floats. One of the major issues was ensuring the menu selections were not characters. Menus were presented as numbered lists such as:

```
--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
```

Menu selections were made using an integer. To compare the input with possible menu choices the following comparison was made:

```
if($input == 1)
```

If the input was a character or string a run-time error occurred because "==" cannot compare a string to a number (the "eq" comparator is needed). For this reason, it was

first necessary to ensure the input was a numeric value. One advantage of a single data type is that no casting or other forced type coercions were necessary when getting input from the user or performing mathematical calculations. A single data type can save programmers significant programming time as long as they are careful about the type of value that is being stored.

Since variables in Perl do not have primitive types they do not have to be declared and hence initialized. This could create a problem when attempting to keep a running total, such as when calculating the student's average grade. Without initializing the variable, the interpreter is not sure at what value to begin the total. The interpreter assumes it to be 0, which is correct in most cases. The "`use warnings`" safety guard provides run time warnings of un-initialized types so that they can be initialized. The "`use strict`" safety guard forces all variables to be declared using the "`my`" operator. If a programmer wrote their program without either of these safety guards there would be a number of issues hidden to them where the interpreter makes an assumption on what the programmer was attempting to do.

Perl does differentiate between single-value scalars and arrays. This created a problem when building the Student class because two of the subroutines had to return arrays. It was not possible to just return the array variable, since only the first value would be passed back. The following code has an `@` sign attached to the front to let the interpreter know that an array is being returned and not a scalar:

```
return @{ $stu->{COURSES} }
```

This is because there were no declarations of the instance variable types in the class constructor "`new()`" and there are no formal return types for the class methods. This means that a programmer has to be careful of what type is being returned or there may be unexpected behaviour.

Perl's first pass check never returned any type problems but there were a few type problems that occurred at run-time. The few errors related to typing that occurred during

the development and testing of the system reinforces the view of weak typing as a "friendlier" version of strong typing. By comparing the experiences of the development of the application in Python to those of the Perl equivalent, a number of issues relating to typing were identified.

4.2 Development in Python and its Differences

Python and Perl are good illustrations of how values in strongly typed languages are rigidly enforced, whereas in weakly typed languages the type of a value depends on how it is being manipulated. As an example, consider adding a string to an integer. Perl, being weakly typed, treats the integer 3 and the string "3" as interchangeable, not requiring any explicit conversion:

```
>>> print "3" + 3
6
>>> print "3".3
33
```

Python, on the other hand, is more strongly typed, and thus raises an exception unless types are explicitly converted:

```
>>> 3 + "3"
Traceback (most recent call last):
  File "", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 3 + 3
6
>>> "3" + str(3)
'33'
```

While developing the Python version of the application (see Appendix A-2), some type errors were encountered. For example, the index variable was unknowingly a string for this line:

```
courses = students[index-1].getCourses()
```

resulting in the following type error:

```
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in -toplevel-
    import test
```

```
  File "C:\Python23\test.py", line 139, in -toplevel-
    updateStudent()
  File "C:\Python23\test.py", line 82, in updateStudent
    courses = students[index-1].getCourses()
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

For Perl this would not have been reported because the string would have been implicitly coerced, the subtraction happening without complication. Another incident involved the line:

```
totalGrade = totalGrade + grades[count]
```

and the following error:

```
Traceback (most recent call last):
  File "C:\Python23\test.py", line 206, in -toplevel-
    viewStudent()
  File "C:\Python23\test.py", line 160, in viewStudent
    totalGrade = totalGrade + grades[count]
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In this case it was addition being attempted between an integer (totalGrade) and string (grades[count]). Again, this would not have been reported by Perl because of its weak typing, but in Python an explicit cast was necessary:

```
totalGrade = totalGrade + int(grades[count])
```

The majority of errors encountered while developing the application in Python were related to typing. In particular, strings and integers were mistakenly being used in each other's place, causing a number of run time errors. Python's strong typing does not allow implicit conversions as allowed by Perl's weaker typing.

5 Conclusions

Python's characteristic clean syntax did not seem to significantly affect development time in comparison to Perl. As expected, the number of typing errors encountered during development were fewer with the Perl version because of its weak typing. Yet both versions of the application were developed in approximately 5 hours.

The relatively small size of the application creates difficulties in gauging development time.

The strength of typing used by a language affects its readability and writability. In particular, the Perl version of the program is less readable but was much more flexible in terms of typing, resulting in increased writability. The application written in Python, though more readable, required more overhead because of explicit type conversion. Given that scripting languages are commonly used as glue languages, writability represents the higher priority of the two.

Strong and weak typing have both advantages and disadvantages. Weakly typed scripting languages save programmers time. The implicit type conversions of weakly typed languages may allow errors to go unnoticed. On the other hand, strongly typed languages tend to promote reliability by requiring explicit type conversions. This kind of tradeoff is present in every programming language design decision.

Bibliography

1. Johnson, Justin, (last updated 2003-11-03). "Weak vs Strong Typing", Homepage of: Justinalia. [Online]. Available: http://www.justinalia.com/cs/programming/flames/typing.html

2. Marinacci, Joshua, (last updated 2003-08-15). "Strong vs Weak Typing: Can't we have the best of both worlds?", Homepage of: Java.net. [Online]. Available: http://weblogs.java.net/pub/wlg/373

3. Ousterhout, John K., "Scripting: Higher Level Programming for the 21st Century" [Online]. Available: http://home.pacbell.net/ouster/scripting.html

4. Sebesta, Robert W., "Concepts of Programming Languages 6th Edition", Pearson Ecuation, Inc. 2004

5. Unknown, (last updated 2001-03-14). "Are scripting languages the wave of the future?", Homepage of: ITworld.com. [Online]. Available: http://www.itworld.com/AppDev/1262/itw-0314-rcmappdevint/pfindex.html

6. Unknown, (last updated 2004-03-03). "Scripting Language", Homepage of: Cunningham & Cunningham, Inc. [Online]. Available: http://c2.com/cgi/wiki?ScriptingLanguage

7. Unknown, "Scripting Programming Language", Homepage of: Wikipedia. [Online]. Available: http://c2.com/cgi/wiki?ScriptingLanguage

8. Unknown, "Weak Typing". Homepage of: Free Online Dictionary of Computing (FOLDOC). [Online]. Available: http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?weak+typing

Appendix A-1.1 – Perl

File: studentManagement.pl

```
#!/usr/bin/perl

#Created by Alastair Grant
#For CS4613 project

use warnings;
use Student;

sub createStudent();
sub updateStudent();
sub viewStudent();

print "Student Manager version 1.0\n";

my $file = "students.dat";
my $input;
my $done = 0;

while (!$done){
	# a menu
	print "\n--Main menu--\n";
	print "1 - Create a new student\n";
	print "2 - Update a student's grade\n";
	print "3 - View a student's grades\n";
	print "4 - Exit\n";

	#Input can contain either numbers or strings.
	$input = <>;
	print "\n";

	#validates the input
	#because it can contain numbers or strings, we have
	#to check to make sure it is not a string first
	#since strings cause problems for == comparisons
	if($input eq "\n" || !($input !~ /^[^0-9]*$/)){
		print "Select a valid option\n\n";
	}
	#process the options
	elsif($input == 1){
		createStudent();
	}

	elsif($input == 2){
		updateStudent();
	}
	elsif($input == 3){
		viewStudent();
	}
	elsif($input == 4){
		print "Exiting...\n";
```

```
            $done = 1;
      }
      #if it is a number, but out of range
      else{
            print "Select a valid option\n\n";
      }
}

#creates a new student record at the end of the file
sub createStudent(){
      my (@courses, @grades);
      my $doneCourses = 0;

      #gather the info in a student object
      $newStudent = Student->new();
      print "Enter new student name:\n";

      $input = <>;
      chomp($input);
      $newStudent->name($input);

      print "Enter course (x to stop)\n";
      while($doneCourses == 0){
            $input = <>;
            if($input eq "\n"){
                  print "Enter course (x to stop)\n";
            }
            elsif(!($input eq "x\n")){
                  chomp($input);
                  push @courses, $input;
                  push @grades, "0";
            }
            else{
                  $doneCourses = 1;
                  $newStudent->courses(@courses);
                  $newStudent->grades(@grades);
            }
      }

      #make the course list into a string
      my $courseList = join(",", $newStudent->courses);
      my $gradeList = join(",", $newStudent->grades);
      printf "Adding student %s with courses %s\n", $newStudent->name,
$courseList;

      #write the string representation of the student to a file
      open(OUT, ">>$file");

      printf OUT "%s-%s-%s\n", $newStudent->name, $courseList,
$gradeList;

      close(OUT)

}

sub updateStudent(){
      #read in the values from the file and display the students
```

```
my @filedata;
open(IN, "$file");

#declarations are not necessary but useful at times
my $record;
my @newStudents;
my @filepos;
my (@courses, @grades);
my $i = 0;

#store all the read in values in the student class.
while ($record = <IN>) {
      $newStudents[$i] = Student->new();
      ($name, $courseList, $gradeList) = split("-", $record);

      #Store it in a record
      $newStudents[$i]->name($name);

      @courses = split(",", $courseList);
      @grades = split(",", $gradeList);

      $newStudents[$i]->courses(@courses);
      $newStudents[$i]->grades(@grades);
      $i++;
}
close(IN);

#Find out which user we want to consider
for($j = 0; $j < $i; $j++){
      printf "%d - %s\n", ($j+1), $newStudents[$j]->name;
}
my $studentRec = "a";
#again validate the input, no characters or strings
while (!($studentRec !~ /^[^0-9]*$/)){
      print "Enter # of student to update: ";
      $studentRec = <>;
}
#get courses and grades for the selected student
@courses = $newStudents[$studentRec-1]->courses;
@grades = $newStudents[$studentRec-1]->grades;

#look up all courses and grades for the selected student
$numGrades = $#courses;
for($k = 0; $k <= $numGrades; $k++){
      printf "%d - %s  %s\n", ($k+1), $courses[$k], $grades[$k];
}
#validate the input
my $gradeRec = "a";
while (!($gradeRec !~ /^[^0-9]*$/)){
      print "Enter # of course grade to update: ";
      $gradeRec = <>;
}

#let the user input a new grade
print "Enter new grade: ";
my $newGrade = <>;
#remove it's \n at the end
```

```
	chomp($newGrade);
	#save it in the array
	$grades[($gradeRec-1)] = $newGrade;
	$newStudents[$studentRec-1]->grades(@grades);

	#update the students file.
	open(OUT, ">$file");
	for($m = 0; $m < $i; $m++){
		#make the course list into a string
		my $courseList = join(",", $newStudents[$m]->courses);
		my $gradeList = join(",", $newStudents[$m]->grades);
		chomp($gradeList);

		#write the values to a file
		printf OUT "%s-%s-%s\n", $newStudents[$m]->name,
$courseList, $gradeList;
	}
	close(OUT)
}

sub viewStudent(){
	#read in the values from the file and display the students
	my @filedata;
	open(IN, "$file");
	@filedata = <IN>;
	close(IN);

	my $record;
	my @newStudents;
	my (@courses, @grades);
	my $i = 0;

	#store all the read in values in the student class.
	foreach $record (@filedata){
		$newStudents[$i] = Student->new();
		($name, $courseList, $gradeList) = split("-", $record);

		#Store it in a record
		$newStudents[$i]->name($name);

		@courses = split(",", $courseList);
		@grades = split(",", $gradeList);

		$newStudents[$i]->courses(@courses);
		$newStudents[$i]->grades(@grades);

		$i++;
	}

	for($j = 0; $j < $i; $j++){
		printf "Name: %s\n", $newStudents[$j]->name;

		#get courses and grades
		@courses = $newStudents[$j]->courses;
		@grades = $newStudents[$j]->grades;
		$totalGrade = 0;
		$numGrades = $#courses;
```

```
            print "Course  Grade\n";

            #calculate an average for the student as well.
            for($k = 0; $k <= $numGrades; $k++){
                  printf "%s  %s\n", $courses[$k], $grades[$k];
                  $totalGrade = $totalGrade + $grades[$k];
            }
            printf "Average grade => %2.2f\n",
$totalGrade/($numGrades+1);
      }

}
```

File: Student.pm

```
#A class to represent a student
#Written by Alastair Grant
package Student;
      #a constructor
      sub new {

            my $stu = {};
            $stu->{NAME} = undef;
            $stu->{COURSES} = [];
            $stu->{GRADES} = [];
            bless($stu);
            return $stu;
      }

      #both get and set methods.  If there is an argument, it sets it
      #otherwise it just returns.
      sub name {
            my $stu = shift;
              if (@_) {
                  $stu->{NAME} = shift
            }
              return $stu->{NAME};
          }

      sub courses {
              my $stu = shift;
            if (@_) {
                  @{ $stu->{COURSES} } = @_
            }
              return @{ $stu->{COURSES} };
      }

      sub grades {
            my $stu = shift;
              if (@_) {
                  @{ $stu->{GRADES} } = @_
            }
              return @{ $stu->{GRADES} };
      }

1; #so that "use" doesn't complain
```

Appendix A-1.2 – Perl Test Run

```
Student Manager version 1.0

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
1

Enter new student name:
Alastair
Enter course (x to stop)
CS4613
CS3113
x
Adding student Alastair with courses CS4613,CS3113

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
1

Enter new student name:
Chris
Enter course (x to stop)
EE1713
x
Adding student Chris with courses EE1713

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
2

1 - Alastair
2 - Chris
Enter # of student to update: 1
1 - CS4613  0
2 - CS3113  0

Enter # of course grade to update: 1
Enter new grade: 99

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
2
```

```
1 - Alastair
2 - Chris
Enter # of student to update: 2
1 - EE1713  0

Enter # of course grade to update: 1
Enter new grade: 76

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
3

Name: Alastair
Course  Grade
CS4613  99
CS3113  0

Average grade => 49.50
Name: Chris
Course  Grade
EE1713  76

Average grade => 76.00

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
2

1 - Alastair
2 - Chris
Enter # of student to update: 1
1 - CS4613  99
2 - CS3113  0

Enter # of course grade to update: 2
Enter new grade: 85

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
2

1 - Alastair
2 - Chris
Enter # of student to update: 2
1 - EE1713  76

Enter # of course grade to update: 1
```

```
Enter new grade: 80

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
3

Name: Alastair
Course  Grade
CS4613  99
CS3113  85

Average grade => 92.00
Name: Chris
Course  Grade
EE1713  80

Average grade => 80.00

--Main menu--
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
4

Exiting...
```

Appendix A-2.1 – Python

File: studentManagement.py

```
#!/usr/bin/python

# Created by David Hirtle.

# A system for managing (adding, updating, viewing) student #
      records, consisting of name, courses and grades

# Implemented as part of a project for CS4613 at
#     the University of New Brunswick

import sys
from string import join, split

from Student import Student # in Student.py

FILENAME = "students.dat"

def createStudent():
    """Creates a new student record at the end of the file."""

    print "" # skip a line

    # get student name
    done = False
    while not done:
        input = raw_input("Enter the student's name: ").strip()
        if input != "":
            done = True
            name = input

    print "" # skip a line
    courses = []
    grades = []

    # get list of courses student has taken
    done = False
    while not done:
        input = raw_input("Enter " + name
                        + "'s courses (x to stop): ").strip()
        if input == "":
            continue
        elif input != "x":
            courses.append(input)
            grades.append("0")
        else:
            done = True
              # create the student object
            newStudent = Student(name, courses, grades)

    courses = join(courses, ", ")
```

```
    grades = join(grades, ", ")

    # record the new student record in the list
    f = file(FILENAME, "a")
    f.write("%s|%s|%s\n" % (name, courses, grades))
    f.close()

    print "\nAdded student '%s' with the following course(s):\n%s\n" %
(name, courses)


def updateStudent():
    """Update a student's grades."""

    # read everything from list and store as objects
    students = []
    f = file(FILENAME, "r")
    for line in f:
        (name, courses, grades) = split(line.strip(), "|")
        courses = split(courses, ", ")
        grades = split(grades, ", ")
        # create the student object
        students.append(Student(name, courses, grades))
    f.close()

    print "" # skip a line
    # find out which student's grades are to be updated
    for count in range(len(students)):
        print "%d. %s" % (count+1, students[count].getName())

    print "" # skip a line
    done = False
    while not done:
        input = raw_input("Enter the number of the student to update:
").strip()
        if input in str(range(1,len(students)+1)) and input != "":
            done = True
            studentIndex = int(input)-1

    courses = students[studentIndex].getCourses()
    grades = students[studentIndex].getGrades()

    # find out which course's grades are to be updated
    for count in range(len(courses)):
        print "%d. %s (grade: %s)" % (count+1, courses[count],
grades[count])

    print "" # skip a line
    done = False
    while not done:
        input = raw_input("Enter the number of the course grade to
update: ").strip()
        if input in str(range(1,len(courses)+1)) and input != "":
            done = True
            index = int(input)-1

    updatedCourse = courses[index]
```

```
    updatedStudent = students[studentIndex].getName()

    # let the user input a new grade
    print "" # skip a line
    done = False
    while not done:
        input = raw_input("Enter a new grade for course "
                        + updatedCourse + ": ").strip()
        if input in str(range(101)) and input != "":
            done = True
            newGrade = input

    grades[index] = newGrade
    students[studentIndex].setGrades(grades)

    # update the student list
    f = file(FILENAME, "w")
    for count in range(len(students)):
        name = students[count].getName()
        courses = join(students[count].getCourses(), ", ")
        grades = join(students[count].getGrades(), ", ")
        f.write("%s|%s|%s\n" % (name, courses, grades))
    f.close()
    print "\nUpdated %s's grade for %s to %s.\n"
                  % (updatedStudent, updatedCourse, newGrade)


def viewStudent():
    """Displays all courses and grades for a student, and computes
average grade."""

    # read in everything from list and stores as objects
    students = []
    f = file(FILENAME, "r")
    for line in f:
        (name, courses, grades) = split(line.strip(), "|")
        courses = split(courses, ", ")
        grades = split(grades, ", ")
        # create the student object
        students.append(Student(name, courses, grades))
    f.close()

    print "" # skip a line
    # find out which student's grades are to be updated
    for count in range(len(students)):
        print "%d. %s"
                        % (count+1, students[count].getName())

    print "" # skip a line
    done = False
    while not done:
        input = raw_input("Enter the number of the student to view:
").strip()
        if input in str(range(1,len(students)+1)) and input != "":
            done = True
            studentIndex = int(input)-1
```

```
    courses = students[studentIndex].getCourses()
    grades = students[studentIndex].getGrades()
    name = students[studentIndex].getName()

     # display all courses and grades for selected student, # and
     calculate average grade
    totalGrade = 0
    print "\nCourse       Grade"
    print   "--------     -----"
    for count in range(len(courses)):
        print courses[count].ljust(8), "   ", grades[count].rjust(5)
        totalGrade = totalGrade + int(grades[count])

    average = totalGrade/(count+1)
    print "\n%s's average grade: %d" % (name, average)
    print "" # skip a line


def main():
    """Main menu of Student Management system."""

    print "Student Manager (version 1.0)\n"

    # get user's menu choice
    done = False
    while not done:

        print "Main menu"
        print "============================"
        print "1 - Create a new student"
        print "2 - Update a student's grade"
        print "3 - View a student's grades"
        print "4 - Exit\n"

        while True:
            input = raw_input('Select a menu option (1-4): ').strip()

            if input in str(range(1,5)) and input != "":

                # validates the input
                if input == "1":
                    createStudent()

                elif input == "2":
                    updateStudent()

                elif input == "3":
                    viewStudent()
                elif input == "4":
                    print "" # skip a line
                    done = True
                break

main() # show menu
```

File: Student.py

```
#!/usr/bin/python

# Created by David Hirtle.
# A class to represent students in a Student Management
#   system, a project for CS4613 at UNB

class Student:
    """Represents a student."""
    def __init__(self, name, courses, grades):
        self.name = name
        self.courses = courses
        self.grades = grades;

    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

    def getCourses(self):
        return self.courses

    def setCourses(self, courses):
        self.courses = courses

    def getGrades(self):
        return self.grades

    def setGrades(self, grades):
        self.grades = grades
```

Appendix A-2.2 – Python Test Run

```
Student Manager (version 1.0)

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): test
Select a menu option (1-4): 0
Select a menu option (1-4): 1

Enter the student's name: David

Enter David's courses (x to stop): LING3422
Enter David's courses (x to stop): CS6905
Enter David's courses (x to stop): x

Added student 'David' with the following course(s):
LING3422, CS6905

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 1

Enter the student's name: Lacey

Enter Lacey's courses (x to stop): CE3033
Enter Lacey's courses (x to stop): x

Added student 'Lacey' with the following course(s):
CE3033

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 2

1. David
2. Lacey

Enter the number of the student to update: 1
1. LING3422 (grade: 0)
```

```
2. CS6905 (grade: 0)

Enter the number of the course grade to update: 1

Enter a new grade for course LING3422: 98

Updated David's grade for LING3422 to 98.

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 2

1. David
2. Lacey

Enter the number of the student to update: 5
Enter the number of the student to update: 2
1. CE3033 (grade: 0)

Enter the number of the course grade to update: 1

Enter a new grade for course CE3033: 99

Updated Lacey's grade for CE3033 to 99.

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 3

1. David
2. Lacey

Enter the number of the student to view: 1

Course       Grade
--------     -----
LING3422        98
CS6905           0

David's average grade: 49

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit
```

```
Select a menu option (1-4): 3

1. David
2. Lacey

Enter the number of the student to view: 2

Course       Grade
--------     -----
CE3033          99

Lacey's average grade: 99

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 2

1. David
2. Lacey

Enter the number of the student to update: 1
1. LING3422 (grade: 98)
2. CS6905 (grade: 0)

Enter the number of the course grade to update: 2

Enter a new grade for course CS6905: 150
Enter a new grade for course CS6905: -10
Enter a new grade for course CS6905: 88

Updated David's grade for CS6905 to 88.

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 2

1. David
2. Lacey

Enter the number of the student to update: 2
1. CE3033 (grade: 99)

Enter the number of the course grade to update: 1

Enter a new grade for course CE3033: 100
```

```
Updated Lacey's grade for CE3033 to 100.

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 3

1. David
2. Lacey

Enter the number of the student to view: 1

Course       Grade
--------     -----
LING3422        98
CS6905          88

David's average grade: 93

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 3

1. David
2. Lacey

Enter the number of the student to view: 2

Course       Grade
--------     -----
CE3033         100

Lacey's average grade: 100

Main menu
============================
1 - Create a new student
2 - Update a student's grade
3 - View a student's grades
4 - Exit

Select a menu option (1-4): 4

>>>
```