PickleBOL Implementation

Two and a Half Caleb’s – Caleb Scott, Caleb Evans and Luke DeGoes

# Introduction

This document outlines our team’s implementation of the pickle programming language. Each section includes a description of our implementation, important decisions we made designing the implementation, examples of pickle code, and information on included test files. Output from example code is commented in line with each print statement to describe the expected output in order to save space in the document.

[1 Introduction](#_Toc70788865)

[2 Data Types](#_Toc70788866)

[2.1 Numeric](#_Toc70788867)

[2.1.1 Int](#_Toc70788868)

[2.1.2 Float](#_Toc70788869)

[2.2 Bool](#_Toc70788870)

[2.3 String](#_Toc70788871)

[2.4 Date](#_Toc70788872)

[3 Arrays](#_Toc70788873)

[3.1 Array-to-Array Assignment](#_Toc70788874)

[4 Operations](#_Toc70788875)

[4.1 Generic Operations](#_Toc70788876)

[4.1.1 Equal To (==)](#_Toc70788877)

[4.1.2 Not Equal To (!=)](#_Toc70788878)

[4.1.3 Less Than (<)](#_Toc70788879)

[4.1.4 Greater Than (>)](#_Toc70788880)

[4.1.5 Less Than or Equal To (<=)](#_Toc70788881)

[4.1.6 Greater Than or Equal To (>=)](#_Toc70788882)

[4.2 Numeric Operations](#_Toc70788883)

[4.2.1 Binary Plus (+)](#_Toc70788884)

[4.2.2 Binary Minus (-)](#_Toc70788885)

[4.2.3 Unary Minus (-)](#_Toc70788886)

[4.2.4 Multiplication (\*)](#_Toc70788887)

[4.2.5 Division (/)](#_Toc70788888)

[4.2.6 Exponentiation (^)](#_Toc70788889)

[4.3 String Operations](#_Toc70788890)

[4.3.1 Concatenation (#)](#_Toc70788891)

[4.3.2 Subscripting (*string*[i])](#_Toc70788892)

[4.4 Array Operations](#_Toc70788893)

[4.4.1 Subscripting (*array*[i])](#_Toc70788894)

[4.5 Logical Operations](#_Toc70788895)

[4.5.1 Logical And (and)](#_Toc70788896)

[4.5.2 Logical Or (or)](#_Toc70788897)

[4.5.3 Logical Not (not)](#_Toc70788898)

[5 Slices](#_Toc70788899)

[5.1 Slicing Strings](#_Toc70788900)

[5.2 Slicing Arrays](#_Toc70788901)

[6 Flow Control](#_Toc70788902)

[6.1 break and continue](#_Toc70788903)

[6.2 for *cv* = *sv* to *endValue* by *incr*:](#_Toc70788904)

[6.3 for *char* in *string*:](#_Toc70788905)

[6.4 for *item* in *array*:](#_Toc70788906)

[6.5 for *stringCV* from *string* by *delimiter*:](#_Toc70788907)

[6.6 while *condition*:](#_Toc70788908)

[6.7 if *condition*:](#_Toc70788909)

[6.8 else](#_Toc70788910)

[6.9 select when default](#_Toc70788911)

[7 Functions](#_Toc70788912)

[7.1 Built-In Functions](#_Toc70788913)

[7.1.1 print()](#_Toc70788914)

[7.1.2 LENGTH(*string*)](#_Toc70788915)

[7.1.3 SPACES(*string*)](#_Toc70788916)

[7.1.4 ELEM(*array*)](#_Toc70788917)

[7.1.5 MAXELEM(array)](#_Toc70788918)

[7.1.6 dateDiff(*date1*, *date2*)](#_Toc70788919)

[7.1.7 dateAdj(*date*, *int*)](#_Toc70788920)

[7.1.8 dateAge(*date1*, *date2*)](#_Toc70788921)

[7.2 User Defined Functions](#_Toc70788922)

[8 Flexible Points Completed](#_Toc70788923)

# Data Types

## Numeric

There are two numeric data types: Int and Float. Both data types are implemented via a single Numeric Class. The Numeric Class contains attributes to store the String value, Integer value, Float value and Sub-Classification of the Numeric Object. The Class also contains methods to create, parse, and print Numeric objects.

Int and Float data types are outlined in the following sections.

### Int

The Int data type represents an integer value as a four-byte integer.

As outlined in [Section 2.1](#_Numeric), our implementation of the Int data type uses a Numeric Class to handle creating, parsing, and printing Int objects. Both positive and negative integers are supported. Integers are parsed from Strings by utilizing the built in Java functions of the String Class (String.matches()).

The Int data type can be used in conjunction with the following operations as defined in [Section 4](#_Operations): addition, subtraction, multiplication, division, exponentiation, equal to, not equal to, less than, greater than, less than or equal to and greater than or equal to.

**Example Code:**

Int radius;

radius = 8;

Int radius2 = radius ^ 2;

print(radius); // 8

print(radius2); // 64

**Test Cases:**

### Float

The Float data type represents a floating-point value as an eight-byte floating-point (double).

As outlined in [Section 2.1](#_Numeric), our implementation of the Float data type uses a Numeric Class to handle creating, parsing, and printing Float objects. Both positive and negative floating-point values are supported. Floating-point values are parsed from Strings by utilizing the built in Java functions of the String Class (String.matches()).

Float objects are always printed with a minimum of two decimal places even though they may contain a less precise value. A Float value can be assigned to an Int and the Float value will be truncated during the assignment.

The Float data type can be used in conjunction with the following operations as defined in [Section 4](#_Operations): addition, subtraction, multiplication, division, exponentiation, equal to, not equal to, less than, greater than, less than or equal to and greater than or equal to.

**Example Code:**

Float pi = 3.14;

Float area;

Float radius2;

Int radius = 8;

Int truncated = pi;

radius2 = radius ^ 2;

area = pi \* radius2;

print(pi); // 3.14

print(area); // 200.96

print(truncated); // 3

**Test Cases:**

## Bool

The Bool data type represents a Boolean value as a single byte character ‘T’ or ‘F’.

Our implementation of the Bool data type is done via a Bool Class. The Bool Class contains attributes to store the Boolean value, String value, Character value and Sub-Classification of the Bool object. The Class also contains methods to create, parse, and print Bool objects. Boolean values are parsed from Strings representing the single characters ‘T’ or ‘F’.

**Example Code:**

Bool bFlag;

bFlag = T;

print(bFlag); // T

**Test Cases:**

## String

The String data type represents a string of characters. String objects are mutable, and each character is a single byte.

Our implementation of the String data type is done via a ResultValue Class. The ResultValue Class contains attributes to store the String value and Sub-Classification of the String object. We did not implement an additional String Class because the existing ResultValue Class contained all the necessary parts to implement the String data type.

The String data type can be used in conjunction with the concatenation, subscripting, and generic operations as defined in [Section 4](#_Operations). The data type can also be used with the built-in functions LENGTH and SPACES as defined in [Section 7](#_Functions).

Slicing of String objects is supported as defined in [Section 5](#_Slices).

**Example Code:**

String str;

str = "My name";

print(str); // My Name

print("Jeff"); // Jeff

**Test Cases:**

## Date

The Date data type represents a date as a string of 10 characters in the YYYY-MM-DD format.

Our implementation of the Date data type is supported via a Date Class. The Date Class contains static attributes and methods to parse dates, convert Julian date format strings into YYYY-MM-DD strings, calculate differences in days, adjust dates by day counts and calculate differences in years. Calculations on dates is done via conversion of the YYYY-MM-DD date String into Julian date format integer to simplify calculations.

Because Dates are implemented as strings, they support all the same operations of the String data type as defined in [Section 2.3](#_String_1).

The Date data type is extended by the functions dateDiff, dateAdj and dateAge. Each are defined in [Section 7](#_Functions).

**Example Code:**

Date date = "1999-02-27";

print(date); // 1999-02-27

print(dateDiff("2021-02-27", date), "days"); // 8036 days

print(dateAge("2021-02-27", date), "years"); // 22 years

print(dateAdj(date, 10)); // 1999-03-09

**Test Cases:**

# Arrays

Arrays in pickles are homogeneous lists of the Int, Float, Bool or String data types. They can be either bounded or unbounded and can be declared and assigned in multiple ways. Arrays are indexed starting from 0, where 0 is the first element of the array.

Our implementation of arrays is done via a ResultList Class. The ResultList Class contains attributes to store the list of objects, the capacity of the array, the allocated size of the array and whether the array is unbounded or bounded.

A bounded array has a set capacity, which limits the maximum number of elements that can be present in the list. An array’s allocated size is the index of the lists’ highest populated element plus one. An unbounded array has a dynamic capacity, where the maximum number of elements increases/decreases based on the allocated size of the array.

An array in pickle is defined using the ‘[]’ characters after the variable name of a variable declaration. Arrays can be unbounded (dynamically grow and shrink in size). An array can be defined in the following ways:

|  |  |
| --- | --- |
| *DataType* *name*[] = *value1*, *value2*, *value3, …, valueN*; | Declare an Array of type *DataType*.  Initialize the Array with *values*. |
| *DataType name*[*N*]; | Declare an Array of type *DataType* with *N* capacity. |
| *DataType name*[*N*] = *value*; | Declare an Array of type *DataType* with *N* capacity, where all *N* values are *value*. (Scalar Assignment) |
| *DataType name*[*N*] = *value1*, *value2*, *value3, …, valueN*; | Declare an Array of type *DataType* with *N* capacity.  Initialize the Array with *values*. |
| *DataType* *name*[unbound]; | Declare an unbounded Array of type *DataType*. |
| *DataType* *name*[unbound] *value1*, *value2*, *value3, …, valueN*; | Declare an unbounded Array of type *DataType*.  Initialize the Array with *values*. |

Arrays in pickle can be used in conjunction with the subscripting operation (see [Section 4.4.1](#_Subscripting_(array[i]))) , the built-in functions ELEM and MAXELEM (see [Section 7.1](#_Built-In_Functions)) and slicing operations (see [Section 5.2](#_Slicing_Arrays)).

## Array-to-Array Assignment

Arrays support array-to-array assignment, where an array can be assigned into another array. Array-to-Array assignment is based on the following rules:

* If a larger array is assigned into a smaller array, the larger array is copied into the smaller array until the smaller array is full.
* If a smaller array is assigned into a larger array, the smaller array is copied into the larger array until the smaller array is out of elements, then the rest of the larger array is filled with empty values up to its capacity.
* If an array is assigned into an array of the same size (or into an unbounded array) all elements will be copied to the target array.

**Example Code:**

// array with 4 elements

Int iArray1[] = 10, 20, 30, 40;

print(iArray1); // 10 20 30 40

// array of max size 3

Int iArray2[3];

iArray2[0] = 30;

iArray2[1] = 20;

iArray2[2] = 10;

print(iArray2); // 30 20 10

// scalar assignment of array with max size 6

Int iArray3[6] = 3;

print(iArray3); // 3 3 3 3 3 3

// Assigning larger array into smaller array

iArray2 = iArray1;

print(iArray2); // 10 20 30

// Assigning smaller array into larger array

iArray3 = iArray1;

print(iArray3); // 10 20 30 40

// Unbounded Array with 3 elements

Int unboundedArray1[unbound] = 20, 30, 40;

print(unboundedArray1);

// Unbounded Array declaration

Int unboundedArray2[unbound];

unboundedArray2[6] = 60;

print(unboundedArray2);

**Test Cases:**

# Operations

## Generic Operations

The behavior of generic operations is dependent on the left operand. If the left operand is a String, then the entire operation is treated as a String comparison, and the second operand will be coerced into a String if necessary. If the left operand is a Numeric the operation is treated as a Numeric comparison and the second operand must be a Numeric, else an error will occur.

### Equal To (==)

The equal to operation returns a Bool object. The returned Bool object will be ‘T’ if both operands are equal, otherwise ‘F’ is returned.

The equal to operation is implemented via the Utility Class as a static method.

**Example Code:**

print("1" == 1); // T

print("fun" == "fun"); // T

print("fun" == "nuf"); // F

print(1 == 1); // T

print(1 == 2); // F

**Test Cases:**

### Not Equal To (!=)

The not equal to operation returns a Bool object. The returned Bool object will be ‘T’ if both operands are not equal, otherwise ‘F’ is returned.

The not equal to operation is implemented via the Utility Class as a static method.

**Example Code:**

print("1" != 1); // F

print("fun" != "fun"); // F

print("fun" != "nuf"); // T

print(1 != 1); // F

print(1 != 2); // T

**Test Cases:**

### Less Than (<)

The less than operation returns a Bool object. The returned Bool object will be ‘T’ if the first operand is less than the second operand, otherwise ‘F’ is returned.

The less than operation is implemented via the Utility Class as a static method.

**Example Code:**

print(1 < 2); // T

print(1 < 0); // F

print(1 < 1); // F

print("1" < 1); // F

print("1" < "2"); // T

print("1" < "0"); // F

**Test Cases:**

### Greater Than (>)

The greater than operation returns a Bool object. The returned Bool object will be ‘T’ if the first operand is greater than the second operand, otherwise ‘F’ is returned.

The greater than operation is implemented via the Utility Class as a static method.

**Example Code:**

print(1 > 2); // F

print(1 > 0); // T

print(1 > 1); // F

print("1" > 1); // F

print("1" > "2"); // F

print("1" > "0"); // T

**Test Cases:**

### Less Than or Equal To (<=)

The less than or equal to operation returns a Bool object. The returned Bool object will be ‘T’ if the first operand is less than the second operand or both operands are equal. Otherwise ‘F’ is returned.

The less than or equal to operation is implemented via the Utility Class as a static method.

**Example Code:**

print(1 <= 2); // T

print(1 <= 0); // F

print(1 <= 1); // T

print("1" <= 1); // T

print("1" <= "2"); // T

print("1" <= "0"); // F

**Test Cases:**

### Greater Than or Equal To (>=)

The greater than or equal to operation returns a Bool object. The returned Bool object will be ‘T’ if the first operand is greater than the second operand or both operands are equal. Otherwise ‘F’ is returned.

The greater than or equal to operation is implemented via the Utility Class as a static method.

**Example Code:**

print(1 >= 2); // F

print(1 >= 0); // T

print(1 >= 1); // T

print("1" >= 1); // T

print("1" >= "2"); // F

print("1" >= "0"); // T

**Test Cases:**

## Numeric Operations

Numeric operations are operations that apply to Numeric data types (see [Section 2.1](#_Numeric)).

The behavior of numeric operations is dependent on the left operand. If the left operand is an Int, then the second operand (for binary operations) will be coerced into an Int if it is a Float. If the left operand is a Float, then the second operand (for binary operations) will be coerced into a Float if it is an Int.

### Binary Plus (+)

The binary plus operation takes two numeric values (Int or Float data type) and returns the result of adding the second numeric to the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The binary plus operation is implemented via the Utility Class as a static method.

**Example Code:**

Int iVal = 10;

Float fVal = 1.25;

print(iVal + 8); // 18

print(iVal + 2.75); // 12

print(fVal + 3); // 4.25

print(fVal + 0.25); // 1.5

**Test Cases:**

### Binary Minus (-)

The binary minus operation takes two numeric values (Int or Float data type) and returns the result of subtracting the second numeric from the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The binary minus operation is implemented via the Utility Class as a static method.

**Example Code:**

Int iVal = 10;

Float fVal = 1.25;

print(iVal - 8); // 2

print(iVal - 2.75); // 8

print(fVal - 3); // -1.75

print(fVal - 0.25); // 1.0

**Test Cases:**

### Unary Minus (-)

The unary minus operation takes a single numeric value and returns the result of negating the value of the numeric operand. The result will have the same data type as the operand provided.

The unary minus operation is implemented via the Utility Class as a static method.

**Example Code:**

Int iVal = -25;

Float fVal = -0.25;

print(-iVal); // 25

print(-12); // -12

print(-fVal); // 0.25

print(-0.33); // -0.33

**Test Cases:**

### Multiplication (\*)

The multiplication operation takes two numeric values (Int or Float data type) and returns the result of multiplying the second numeric by the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The multiplication operation is implemented via the Utility Class as a static method.

**Example Code:**

Int iVal = 25;

Float fVal = 0.25;

print(iVal \* 5); // 125

print(iVal \* 3.9); // 75

print(fVal \* 2); // 0.5

print(fVal \* 0.25); // 0.0625

**Test Cases:**

### Division (/)

The division operation takes two numeric values (Int or Float data type) and returns the result of dividing the second numeric from the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The division operation is implemented via the Utility Class as a static method.

**Example Code:**

Int iVal = 125;

Float fVal = 15.25;

print(iVal / 25); // 5

print(iVal / 15.9); // 8

print(fVal / 2); // 7.625

print(fVal / 0.25); // 61.0

**Test Cases:**

### Exponentiation (^)

The exponentiation operation takes two numeric values (Int or Float data type) and returns the result of raising the first numeric to the power of the second numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The exponentiation operation is implemented via the Utility Class as a static method.

**Example Code:**

Int iVal = 10;

Float fVal = 0.25;

print(iVal ^ 2); // 100

print(iVal ^ 0.1); // 1

print(fVal ^ 2); // 0.0625

print(fVal ^ 3.0); // 0.015625

**Test Cases:**

## String Operations

### Concatenation (#)

The String concatenation operator concatenates two Strings into a single String, where the second String is concatenated to the first String immediately after the last character in the first String.

Both String objects and constants can be concatenated.

The String concatenation operation is implemented via the Utility Class as a static method.

**Example Code:**

String str1 = "King";

print("Burger " # str1); // Burger King

print("School " # "Bus"); // School Bus

**Test Cases:**

### Subscripting (*string*[i])

String subscripting provides a way to index individual characters within a string. String characters are indexed starting from 0, where index 0 is the first character. Our implementation of subscripting allows negative indexing of strings starting from the last character, where -1 is the last character in the string.

The string subscripting operation is implemented via the Utility Class as a static method.

**Example Code:**

String str = "abcdefg";

print(str[0]); // a

print(str[1]); // b

print(str[-1]); // g

print(str[-2]); // f

**Test Cases:**

## Array Operations

### Subscripting (*array*[i])

Array subscripting provides a way to index individual elements within an array. Array elements are numbered starting from 0, where index 0 is the first element in the array. Our implementation of subscripting allows negative indexing of arrays starting from the last element, where -1 is the last element in the array.

The array subscripting operation is implemented via the Utility Class as a static method.

**Example Code:**

Int array[] = 10, 20, 30, 40, 50, 60;

print(array[0]); // 10

print(array[1]); // 20

print(array[-1]); // 60

print(array[-2]); // 50

**Test Cases:**

## Logical Operations

### Logical And (and)

The logical and operation requires two Bool operands and returns a Bool object. The returned Bool object’s value is ‘T’ if and only if both operands are ‘T’. Otherwise ‘F’ is returned.

The logical and operation is implemented via the Utility Class as a static method.

**Example Code:**

print(T and T); // T

print(T and F); // F

print(F and T); // F

print(F and F); // F

**Test Cases:**

### Logical Or (or)

The logical or operation requires two Bool operands and returns a Bool object. The returned Bool object’s value is ‘T’ either or both operands are ‘T’. Otherwise ‘F’ is returned.

The logical or operation is implemented via the Utility Class as a static method.

**Example Code:**

print(T or T); // T

print(T or F); // T

print(F or T); // T

print(F or F); // F

**Test Cases:**

### Logical Not (not)

The logical or operation requires a Bool operand and returns a Bool object. The returned Bool object’s value is ‘T’ if the Bool operand is ‘F’, or ‘F’ if the Bool operand is ‘T’.

The logical not operation is implemented via the Utility Class as a static method.

**Example Code:**

print(not F); // T

print(not T); // F

**Test Cases:**

# Slices

Our implementation of pickle supports the slicing of Strings and Arrays. Slicing is the process of creating substrings or sub-arrays from Strings or Arrays from indices.

## Slicing Strings

Strings can be sliced by combining the subscript operation ‘[]’ and the slicing operation ‘~’. A String is sliced from an inclusive lower-bound index to an exclusive upper-bound index. The indices provided must be within the bounds of the String (upper-bound can be max+1 since it is exclusive). Index values start at 0, which represents the index of the first character in the String. Negative indexing is supported where -1 is the index of the last character in the String.

It is not necessary to provide both the lower-bound and upper-bound indices if you are slicing from the beginning or end of a String. Either the lower-bound or the upper-bound index can be blank, where a blank lower-bound represents the index 0 and a blank upper-bound represents the index -1.

Strings can be assigned into slices of Strings, which will grow or shrink the length of the target String based on the size of the slice and the length of the source String. Assignment uses the following rules:

* When the source String is larger than the length of the target slice, the target String will grow to fit the entire source String within the slice.
* When the source String is smaller than the length of the target slice, the target String will shrink.

**Example Code:**

String str;

str = "goodbye";

print(str[0~4]); // good

print(str[~4]); // good

print(str[4~]); // bye

str = "tacobell";

str[~4] = "school";

print(str); // schoolbell

str[0~6] = "";

print(str); // bell

**Test Cases:**

## Slicing Arrays

Arrays can be sliced by combining the subscript operation ‘[]’ and the slicing operation ‘~’. An array is sliced from an inclusive lower-bound index to an exclusive upper-bound index. The indices provided must be within the bounds of the array (upper-bound can be max+1 since it is exclusive). Index values start at 0, which is the index of the first element in the array. Negative indexing is supported where -1 is the index of the last element in the array. Unbounded arrays can be sliced.

It is not necessary to provide both the lower-bound and upper-bound indices if you are slicing from the beginning or end of an array. Either the lower-bound or the upper-bound index can be blank, where a blank lower-bound represents the index 0 and a blank upper-bound represents the index -1.

Arrays cannot be assigned into slices of Arrays.

**Example Code:**

Float gradeM [] = 90.5, 50.0, 60.0, 85.5;

Float myGradeM[5];

print(gradeM[2~3]); // 60.0

print(gradeM[~3]); // 90.5 50.0 60.0

print(gradeM[2~]); // 60.0 85.5

// The following assignment would cause

// myGradeM to contain 60.0 and 85.5

myGradeM = gradeM[2~];

print(myGradeM); // 60.0 85.5

**Test Cases:**

# Flow Control

## break and continue

**Example Code:**

**Test Cases:**

## for *cv* = *sv* to *endValue* by *incr*:

**Example Code:**

**Test Cases:**

## for *char* in *string*:

**Example Code:**

**Test Cases:**

## for *item* in *array*:

**Example Code:**

**Test Cases:**

## for *stringCV* from *string* by *delimiter*:

**Example Code:**

**Test Cases:**

## while *condition*:

**Example Code:**

**Test Cases:**

## if *condition*:

**Example Code:**

**Test Cases:**

## else

**Example Code:**

**Test Cases:**

## select when default

**Example Code:**

**Test Cases:**

# Functions

## Built-In Functions

### print()

**Example Code:**

**Test Cases:**

### LENGTH(*string*)

**Example Code:**

**Test Cases:**

### SPACES(*string*)

**Example Code:**

**Test Cases:**

### ELEM(*array*)

**Example Code:**

**Test Cases:**

### MAXELEM(array)

**Example Code:**

**Test Cases:**

### dateDiff(*date1*, *date2*)

**Example Code:**

**Test Cases:**

### dateAdj(*date*, *int*)

**Example Code:**

**Test Cases:**

### dateAge(*date1*, *date2*)

**Example Code:**

**Test Cases:**

## User Defined Functions

# Flexible Points Completed

|  |  |  |  |
| --- | --- | --- | --- |
| **Flexible Feature** | **Points** | **Completed (yes, -)** | **Information** |
| **Date** | **10** |  | **required of everyone** |
| **break and continue** | **10** |  | **required of everyone** |
| Infix Expressions | 25 |  |  |
| Unbounded Arrays | 5 |  |  |
| Additional Numeric Assignment | 5 |  | +=, -= |
| IN, NOTIN | 10 |  |  |
| slices | 30 |  |  |
| +5 if doing infix and slices | 5 |  | test cases must show many uses of this |
| for tokenizing | 10 |  |  |
| select when default | 10 |  | including break and continue |
| programmer-defined functions | 80 |  |  |
| functions by value | 5 |  | in addition to by reference |
| functions variable number of args | 10 |  |  |
| functions return arrays | 5 |  | if slices were done, also returning slices |
| **Excellent test Cases** | **30** |  | **You must create test cases which completely test your interpreter. This includes testing for error cases. If your cases are inadequate, you can lose an additional 150 points.** |
| Total Completed (out of 250) |  |  |  |