PieNum Language Reference Manual

October 2017

Hadiah Venner (hkv2001)
Hana Fusman (hbf2113)
Ogochukwu Nwodoh(ocn2000)

Index

Introduction

Our motivation for our language is to use some elements from the NumPy library in the Python programming language to make image processing more accessible. NumPy adds support for large, multidimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. We want to create a static language that has some of the array manipulation power of NumPy. This would then allow us to write programs that involve manipulating arrays and matrices and doing complex mathematical calculations on them. Our vision is to create a function based language that will include built in functions for image processing, while also allowing the user to create his or her own functions.

1. Lexical Conventions

1.1 Comments

Comments begin with a # symbol and end with a # symbol. This convention should be used for both single line and multiline comments.

1.2 Identifiers

Identifiers are entities in our language such as variables, methods and data types. Valid identifier in PieNum are characters include ASCII letters and decimal digits. The first character of an identifier cannot be a digit. Identifiers cannot be the same sequence of characters as keywords.

1.3 Keywords

The following identifiers are reserved and cannot be used otherwise. They are case sensitive:

int	return

float	boolean
if	while
array	print
else	for
void	string
true	false
null	main
rotate_matrix	global

1.4 Literals

PieNum literals can be integers , booleans, floats, and strings.

2. Data Types

2.1 Primitive Types

2.1.1 Integers

Int

An integer is a whole value between -2^{31} and $2^{31} - 1$. The default value is 0.

2.1.2 Boolean

Boolean

A single byte that can have the value true or false. The default value is false.

2.1.3 Float

float

A float is an integer followed a decimal part (some fractional value). The default value is 0.0.

2.1.4 String

String

Strings are a sequence of zero or more ASCII characters, numbers, or spaces. Strings in PieNum must be enclosed in double quotation marks. The default value is the empty string is null. In PieNum, a single ASCII character is a string.

Example 1:
```
     “This is a string”
```
Example 2:
```
     “a”
```

2.1.5 Void

void

Use the void type to signify a function that has no return value.

2.2 Non-Primitive Types

2.2.1 Arrays

array

An array is a container that holds a number of values of a single type. The array size can be specified at creation, but in PieNum arrays are dynamically resizeable. For an array holding integers, the default value is 0. For an array holding strings, the default value is null. For an array holding floats, the default value is 0.0. For an array holding booleans, the default value is false. Arrays in PieNum are zero indexed.

3. Operators

3.1 Operators for Primitive Types

3.1.1 Assignment Operator

The assignment operators assign values from the right hand operand to the left side operand.

Examples:

```
     int x = 8;
     int y = 6 + 7;
     int z = true;
```

3.1.2 Arithmetic Operations

The arithmetic operators include + (addition), - (subtraction), * (multiplication), / (division) and negation. These operations are not defined for boolean.

Addition: `int x = 5 + 2;`
Subtraction: `int x = 3 - 2;`
Exponent: `int x = 4^3;`
Multiplication: `int x = 1 * 2;`
Division: `int x = 10 / 2;`
Negation: `int x = -4;`

Increment and Decrement:

```
#increment the variable i by one and then store the result
in i#
i++;

#decrement the variable i by one and then store the result
in i#
i--;
```

3.1.3 Precedence of Arithmetic Operations

The precedence of arithmetic operations and assignment is as follows:

(Highest)	Assignment operator **=**
	Increment/Decrement **++/--**
	Parentheses for grouping of operations **()**
	Exponent **^**
	Multiplication operator *****
	Division operator **/**

	Addition operator **+**
(Lowest)	Subtraction or negation operator **-**

Example:

```
     int y = 3 * (4 - 7)^3 ;
     # y is assigned the value -81
```

3.1.4 Relational Operators

```
value < value
value > value
value <= value
value >= value
```

The operators are < (less than), >(greater than), <= (less than or equal to) and >=(greater than or equal to). The relational operators group left to right.

3.1.5 Equality Operators

```
value == value
value != value
```

The == (equal to) and != (not equal to) operators evaluate the expression to determine if the two expressions are equal or not equal.

3.1.6 Logical Operators

```
boolean_value && boolean_value
boolean_value || boolean_value
```

The && (logical AND) returns true if both expressions are met and false otherwise. The || (logical OR) returns true if at least one expression is true and false if no expressions are met.

3.2 Array Operations

The array operations include **+** (addition), **-** (subtraction), ***** (multiplication), ****** (Cross Product), **~**(inverse). These operations are not defined for boolean.

3.2.1. Addition

On two 1D arrays this creates an array with the elements of both arrays.

Example:
```
     int[] array1 = {1, 2, 3, 4};
     int[] array2 = {4, 5, 7, 8};
     int[] array3 = array1 + array2;
     #array3 contains the elements of both array1 and
     array2#
```

On two matrices with the same dimensions this creates a matrix with the elements of both matrices. This throws an error if done between matrices of different dimensions. This throws an error if done between a primitive data type and array.

Example:
```
     int[] matrix1 = { {1, 2}, {3, 4 }};
     int[] matrix2 = { {7, 8}, {9, 10}};
     int[] matrix3 = matrix1+ matrix2;

     # the contents of matrix 3 are:
     [ 8, 10]
     [12, 14]
     #
```

3.2.2 Multiplication

Between an array (1D or 2D) and a non-boolean primitive data type, this multiplies all values of the matrix by the data type(scalar multiplication).. The type of the data type and the matrix data type must be the same.

Between two 1D arrays this calculates the dot product.

Between two matrices this performs matrix multiplication. If matrix A’s width is not equal to matrix B’s height, then an error is thrown.
Example:
```
     int[] matrix1 = { {1, 2}, {3, 4 }};
     int[] matrix2 = { {7, 8}, {9, 10}};
     int[] matrix3 = matrix1 * matrix2;

     # the contents of matrix3 are:
     [ 25, 28]
     [57, 64]
     #

     0.5 * matrix3;
     #The above is an invalid operation, matrix of int
     cannot be multiplied by float value#

     matrix3 =* 2; #multiplies all values of matrix3 by 2#
     int[] array1 = {1, 2, 3, 4};
     int[] array2 = {4, 5, 7, 8};

     # the value of x is 67 #
     int x = array1 * array2;
```

3.2.3 Subtraction

On two 1D arrays this creates an array with the elements of the matrix on the left hand side of the operator minus the elements on the right hand side of the operator. This throws an error if done between two matrices.

Example:

```
     int[] matrix1 = { {1, 2}, {3, 4 }};
     int[] matrix2 = { {7, 8}, {9, 10}};
     int[] matrix3 = matrix1 - matrix2;
     # the contents of matrix 3 are:
     [ -6, -6]
     [-6, -6]
     #
```

3.2.4 Cross Product

Between two 1D arrays of the same dimensions this calculates the cross product. This throws an error if done between two matrices or 1D arrays that don’t have the same dimensions.

Example:

```
     int[] matrix1 = {2,3,4};
     Int[] matrix2 = {5,6,7};
```

```
Int[] cross_product = matrix1 ** matrix2

#the content of cross_product
[-3,6,-3]
```

3.2.5 Inversing

Creates a matrix that is the inverse of the original matrix by multiplying it by its identity matrix. Throws an error if done on a primitive data type, 1D array, or matrix that is not square. If a matrix is composed of int values and inverting yields floating point values, the floating point component of the number will be thrown out.

Example:

```
int[] matrix1 = { {1, 2}, {3, 4}};
int [] inverse = ~matrix1;
#the content of inverse is
[-2, 1]
[1, 0]
#
```

3.2.6 Return the length of the array

This returns the length of a one-dimensional array.

Example:

```
int[] array = [2,3,4];
#size = 3#
int size = array.length()
```

4. Statements

4.1 Expression Statements

Expression statements are in the form: statement ;
Usually expression statements are assignments or function calls.

Example:

```
int value;
int value = 14;
```

4.2 if statement

The two forms of conditional statements are:

```
1. if(expression) {statement}
2. if (expression) {statement1}
   else
        {statement2}
```

The expression is evaluated in both cases and if it is true then the first statement is executed, if it evaluates to false statement2 is executed.

4.3 while statement

The while statement has the form:

```
while (expression) {statement}
```

The statement is executed repeatedly as long as the expression evaluates to true.

4.4 for loops

The for loop has the form:

```
for(int i = j; i < k; i++){

     statements;
}
```

The statement is executed repeatedly as long as the condition is still in the range.

4.5 return statement

The return statement has the form:

```
1. return null;
2. return (expression);
```

In the first case nothing is returned to the caller of the function, in the second case the expression is returned.

5. Methods

5.1 Method Basics

A method is a program procedure that is defined as part of a class. It is collection of statements that are grouped together to perform an operation. A void method returns nothing when called. If the void keyword is not present in the method declaration then the method must return another datatype. A method may or may not take in parameters. The data type of the parameters must be declared.

There is no method overloading in this language.

An example of a method declaration:

```
datatype methodName(datatype param1 ) {
#group of statements that do something#
return datatype;
}

#this method does not take in any parameters or return
anything#

void methodName(){
     #group of statements that do something#
}
```

5.2 Main Method

The main method is a method that calls other methods in other files or the methods in the same file it is defined in. There can only be one main method in a file. The parameters for the main method is always a String array called args. This String array are command line arguments that are space separated. The main method always returns void.

Main method declaration:

```
void main(String[] args)
```

```
{
     # do something #
}
```

6. Scope

Scope refers to the lifetime and accessibility of a variable. The scope of the variable depends on where it is declared.

6.1 Local Variables

Local variables are those declared within designated brackets within a method, conditional statements, etc. Local variables can only be used within the method they are defined in. The variable is created when the method is entered or conditional begins and are garbage-collected once the method is exited or conditional ends.

Example:

```
int doStuff(int a) {

     # a and i only exists within this method and once the
     method returns a is garbage collected #

     int i = 0;
     while(i < 100) {
          int b = a+2;
          a = a+b;
          i++;
          # b only exists within this loop, once the loop
          exists, b is garbage collected#
     }
     return a;
}
```

6.2 Global Variables

Global variables are declared outside of all functions, and therefore they are available to be used throughout the entire program. Global variables are declared with the keyword global.

Example:

```
#a is a global variable#
global int a = 2;

if (expression)
{
     a = a + 3;
}
```

7. File I/O

Since this is a matrix-oriented language file I/O will be be for reading in files in portable pixmap format (PPM) and outputting files in portable pixmap format.

7.1 Reading in a File

The readImage function takes in a String of a PPM file and outputs a matrix corresponding to the matrix of the image.

Example:

```
int[][] matrix = readImage("image.ppm");
```

7.2 Output Image File

The postImage function takes in a matrix and outputs a PPM file corresponding to that matrix of the given name. The file will be outputted in the current directory of the program.

Example:

```
postImage(int[][] matrix, "outputFile.ppm");
```

8. Sample Program (For DEMO)

8.1 Image Rotation

In PieNum, the image rotation function rotates a matrix around the X-axis by a specified angle. This is generally used for image rotation.

Example:

```
# ‘-1’ rotates the image counterclockwise 90 degrees and ‘1’
rotates the image clockwise 90 degrees. This returns a new
rotated matrix#

int rotate_matrix(int[][] matrix, -1 );
int rotate_matrix(int[][] matrix, 1 );
```

8.2 Image Resizing

In PieNum, the resize matrix function takes in two parameters, the first parameter is the size and name of the original matrix, and the second parameter is the new size of the matrix. It returns the newly resized matrix.

Example:

```
#This resize function takes a 4 by 4 matrix and resizes it
to a 2 by 1 matrix#

int resize_matrix(int[4][4] matrix, int [2][1] new_size );
```

8.3 Image Contrast

Image contrasting in PieNum, offset the r, g, and b components of colors in an image to contrast the color of the image.

Example:

```
image_contrast(offset_r, offset_g, offset_b);
```

8.4 Image Morphing

A Transformation is a function that maps one set to another set after performing some operations.The startPicture parameter is the original image, the endPicture parameter is the image the picture is blended with, and the degreeOfMorphing parameter is the degree the morphing is done to. degreeOfMorphing is a float ranging from 0.0 to 1.0.

Example:

```
image_morph(int[][] startPicture, int[][] endPicture, float
degreeOfMorphing);
```

8.5 Image Convolution

The image convolution method(image_convolute) takes in the original image as a matrix and a 3 X 3 matrix which represents a kernel. The kernel is then applied to each pixel to produce different effects like Gaussian blurring, image sharpening and edge detection(each effect requires a unique kernel).

```
image_convolute(int[][] kernel, int[][] original_image);
```