

PlOtter

Language Reference Manual

Ibrahima Niang	in2190
Ranjith Kumar Shanmuga Visvanathan	rs3579
Sania Arif	sa3311

7th March, 2016.

Contents

1	Introduction	3
2	Lexical Conventions	3
2.1	Identifiers	3
2.2	Keywords	3
2.3	Comments	3
2.4	Constants	4
2.5	White space	4
3	Types	4
3.1	Operators	5
4	Expressions	5
4.1	<i>num</i>	5
4.2	<i>bool</i>	5
4.3	<i>string</i>	5
4.4	<i>list</i>	5
4.5	<i>hash</i>	7
4.6	<i>point</i>	7
4.7	Arithmetic operators:	7
4.8	Relational and logical operators:	7
5	Statements	8
5.1	Declaration & Assignment	8
5.2	Break	8
5.3	Continue	8
5.4	Return	9
6	Control Statements	9
6.1	Conditions	9
6.2	Loops	9
7	Scope	11
8	Functions	11
8.1	Function Declaration & Definition	11
8.2	Function Call	12
8.3	Built-in functions	12
9	Sample Program	13

1 Introduction

plOtter is a data manipulation language built on the principles and appreciation of a minimalist aesthetic. The goal of plOtter is to provide a means for users to design and implement their own plots to visualize data, that would otherwise be done by rigid, automated software (such as Microsoft Excel). Our language will simplify charting maps through domain-specific types and operations. The users can build their custom graph templates using our language and can reuse the template to visualize different data points.

Very primitive examples could be building a bar-graph, histogram etc. Higher level examples could be drawing small album art, or other complex examples could be to visualize/plot a binary tree, Gantt chart, etc.

Following is a manual that provides reference information for using plOtter. It describes the lexical conventions, basic types scoping rules, built-in functions and also displays a sample program and output.

2 Lexical Conventions

The lexical structure consists of the set of basic rules that define how to write programs in plOtter. The normal token types are identifiers, keywords, operators, delimiters, and literals, as covered in the following sections.

2.1 Identifiers

An identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter (A to Z or a to z) or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9). plOtter is case sensitive, lowercase and uppercase letters are distinct.

2.2 Keywords

num	string	bool	point
true	false	while	for
if	else	then	in
fn	end	break	print
list	and	or	return

2.3 Comments

Both single line and multi-line comments are supported.

Single Line	#
Multi Line	/* */

The pound or hash-tag symbol # is used for comment. Everything from the # to the end of the line is ignored. It can be used for either single line comment multiple line comments as follow:

```
#This is a single line comment
```

`/* ... */` is used for multi line comments. Multi-line comments cannot be nested.

```
/* this is
   an example of
   multi-line comment */
```

2.4 Constants

false: The false value of the bool type.

true: The true value of the bool type.

none: The sole value of types none. It is used to represent the absence of a value.

2.5 White space

- Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated).

- Line statement

In plOtter, end of a statement is marked by a newline character

- Indentations

plOtter provides no braces to indicate blocks of code for function definitions or flow control. Blocks of code are denoted by line indentation, which must be ended by end.

For example:

```
if true:
    print "True"
end
else:
    print "False"
end
```

3 Types

Our language supports the following data-types:

1. Primitive types

<i>num</i>	32 bit floating point number
<i>bool</i>	Boolean value

2. Non-Primitive or object types

<i>point</i>	A bi-element tuple of type <i>num</i>
<i>list</i>	List of <i>num</i> or <i>string</i> elements
<i>hash</i>	Dictionary with <i>num</i> or <i>string</i> keys
<i>string</i>	Chain of characters enclosed by double quotes

Note: We do not have a primitive type ‘char’, and an index of a string points to a string of length one.

3.1 Operators

Operators are listed in order of precedence. All operators associate left to right, except for assignment, which associates right to left.

<i>Arithmetic</i>	+, -, *, /, %, **
<i>Comparison</i>	==, !=, <, >, <=, >=, and, or
<i>Assignment</i>	=, +=, -=, *=, /=, %=, *=
<i>Membership</i>	in, not in

4 Expressions

An expression is a combination of values, variables, and operators.

4.1 *num*

```
num a
a = 10
num b = 10
```

4.2 *bool*

```
bool a
a = false
bool b = false
```

4.3 *string*

```
string a
a = "plOtter"
string b = "plOtter"
```

4.4 *list*

Lists support both lists of num and lists of string. the list is written as a list of comma separated elements inside square brackets. Each element of the list has an index with which it can be accessed directly. The list supports access, appending, deleting. The size can

change dynamically during the course of the program with appending or deletion of an element in the list. Some examples of list initializations are given below.

```
list a = [ 1, 2, 3, 4, 5]
list b = [ 1.0, 2.0, 3.0, 4.0, 5.0]
list c = [ "string 1", "String 2", "String 3"]
list a = []
```

Operations

`append(x)` : Appends the given element, x, to the end of the list. x should be of the same type as the existing elements in list. If this is the first element then that is set as the type for that list. And once this type is fixed it cant be changed after that. i.e After filling a list with num type, we cant pop all elements and insert string. That would throw an error.

```
a.append(6)
will make a to [ 1, 2, 3, 4, 5, 6 ]
```

`pop()` : this operations pops the last element of the list. If there are no elements present, it doesnt do anything.

```
a.pop()
will make a to [ 1, 2, 3, 4, 5]
```

`remove(x)` : This operation is to remove the element at the x'th position. And the index starts with 0.

```
a.remove(3)
will remove index 3 element '4' and make a to [ 1, 2, 3, 5 ]
```

`at(x)` or `"[x]"` : The elements in the list can be accessed by their index using 'at' and providing the index, or by directly providing the index at square brackets. This with return the element at the index x.

```
a.at(2) or a[2]
will return 3
```

`length()` : This returns the length of the list at that point.

```
a.length() with return 4
```

Also the type of the elements in the list need not be specified during the initialization. The language automatically determines the type. However all the elements in a single list variable should be of same type. It should eiether be all string or all num.

4.5 *hash*

This is the key value pair supported by the plotter. The key and value are separated by colon(:) and the key-value pairs are separated by comma(,).

The key should be of type string, and only *num*, *string* and *bool* are supported as values.

```
hash h = {}  
hash options = { "xAxis": true, "yAxis": true, "status":0}  
hash ran = { "boolType": true, "stringType": "hi", "numType":5.0}
```

4.6 *point*

```
point pMax = (10, 10)
```

4.7 Arithmetic operators:

```
2 + 3  
5  
num a = 5.0  
num b = 3.0  
a * b  
15.0  
14 % 3  
2  
22.0 12.0  
1.8333333333333333
```

4.8 Relational and logical operators:

```
True or False  
True
```

```
True and False  
False
```

```
1 < 2  
True
```

```
23 == 45  
False
```

```
34 != 323  
True
```

5 Statements

5.1 Declaration & Assignment

Declaration statements are to declare variables to use in the program. Assignment statements are used to declare variables and initialize a value to a variable, or assign a value to an initialized variable. They can be expressed in the following ways:

1. *Type Name Variable Name = Value*

For example:

```
num x = 3
string s = "hello"
```

Note: Strings values are expressed by enclosing characters in double quotes.

2. *Type Name Variable Name = Value1*
Variable Name = Value2

For example:

```
num x = 3
x = 4.5
```

Note: An un-initialized variable will be initialized to the following:

<i>num</i>	0
<i>string</i>	null
<i>point</i>	(0,0)
<i>list</i>	[]
<i>hash</i>	{ }

5.2 Break

The break statement causes termination of a looping control statement, and causes control to be passed on to the successive statement after the loop ends. In case of nested loops, the current loop (in which the break statement exists) is terminated, and the other loops are still intact.

It is expressed simply as:

break

5.3 Continue

The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. In other words, it returns the control to the beginning of the loop. It is expressed simply as:

continue

5.4 Return

The return statement is used to exit a function, and it may optionally pass back an expression to the caller. If required, the expression is converted, by assignment, to the type of the function in which it appears. A return statement with no arguments is the same as return None. It can be expressed as:

1. *return expression*
2. *return*

6 Control Statements

6.1 Conditions

The 2 forms of conditional statements are:

- *if- else*
This is of the form:

```
if expression
    statements
else
    statements
end
```

Note: The end indicates the end of the conditional block.

- *if- else if - else* This is of the form:

```
if expression
    statements
else if expression
    statements
else
    statements
end
```

Note: The end indicates the end of the conditional block.

In both cases, the expression is evaluated, and if the condition is satisfied, the set of statements associated in the block is executed. If none of the expressions' result is satisfied, the alternate set of statements associated with the *else* block are executed.

6.2 Loops

- *while*

This has the form:

```
while expression
    statements
end
```

The expression is evaluated, and if the condition is satisfied, the set of statements associated in the block are executed. The test takes place before each iteration of execution of the statements, and the statements are executed repeatedly, as long as the value of the expression remains satisfied.

– *for*

This has the following forms:

```
* for Initialize ; Condition ; Increment/ Decrement :
    Statements
end
```

Initialization specifies the initialization for the loop. Condition specifies an expression to test if the loop should continue. The loop is exited when the expression is satisfied. The final part is typically an increment or decrement operation to be made on the loop variable.

```
* for Type name = start ; end :
    Statements
end
```

Here, start specifies the value where from to begin loop iteration. End is the final part, i.e. till when the loop should continue. The value of the loop variable will increment or decrement by 1, till 'end' is reached.

```
* for start : end :
    Statements
end
```

Here, start specifies the value where from to begin loop iteration. End is the final part, i.e. till when the loop should continue. The value of the loop variable will increment or decrement by 1, till 'end' is reached.

The difference between this representation and the previous one, is that here, there is no variable to use inside the loop block, unlike *name* in the previous declaration.

```
* for variable in list :
    Statements
end
```

Here, the loop iterates over a list, and goes through each variable present in the list.

7 Scope

We have broadly 3 categories for scope of variables:

- Global Scope
These are for variables defined outside a function or control statement block, that are accessible throughout the program. They exist throughout the lifetime of the program and can be accessed inside any function or control statement block.
- Local Scope
These are variables that are declared inside a function or a control statement block. They can be used only within the function or control statement block and are prone to cause errors when accessed outside the block.
We define the end of a block using the *end* keyword.
- Function parameters
Formal parameters of a function are treated as local variables within the function and take preference over the global variables inside the function.

8 Functions

Functions allow structure programs as segments of code to perform tasks that also can be reused.

8.1 Function Declaration & Definition

Function declaration is done in the following format:

*fn name of the function (arguments):
end*

Here, arguments can be a set of arguments or no arguments. The function definition does not convey anything about the return type of the function. Some examples are given below.

Examples :

Function to convert Celsius to Fahrenheit. It takes one argument and returns Num type.

```
1  fn convertToCelcius(num f):  
2      return (f-32)/1.8  
3  end  
4
```

Function with no argument, returning type string

```
1  fn getWelcomeString():  
2      return "Hello World"  
3  end
```

8.2 Function Call

Functions are called in the following way. They are basically identified by their function name and the arguments it has, to be sent. Error will be thrown if the number of arguments during the call didn't match the function signature. However defaulting the argument values can be done, in such a case the passing arguments could be less.

A simple function call to the convert to Celsius function.

convertToCelcius(35.0)

Function with default argument values and calling them.

```
1  fn getWelcomeString(string welcome="Hello World") :
2      return welcome
3  end
4
5  #Calling them with params - returns 'Hi all'
6  print getWelcomeString("Hi all")
7
8  #Calling without arguments - uses argument default value 'Hello World'
9  print getWelcomeString()
```

8.3 Built-in functions

These are the functions our language provides. However, most of these can be written by using the primitive blocks in pLOtter.

Note : In all these functions and also the program all size/thickness units are *pixel* and all *color* is 6 byte hex string Eg "FFFAAF". By default color is black. Also if not specified in options default values are substituted.

line (*point p1, point p2, hash options*)

This function lets users draw a line from one point to another point in the SVG. It takes in the starting point, the ending point and at last options like color size etc. The options are thickness and color.

Eg. line((0,0),(10,10),"thickness":10)

printXY (*point p, string s, bool dir, hash options*)

This function lets users print string in the SVG. The printXY function, takes as argument point in the screen the point specifying the top right corner of the print, the text to be printed, boolean dir, representing whether the string should be horizontal(True) or vertical(False), and at last options like color size etc. The options are size and color.

Eg. printXY((10,10),"Hello World", true, "size":10)

rectangle (*point p, num width, num height, hash options*)

This draws a rectangle of width and height, starting from p, with options. The options we are considering now are simple like, thickness(of the border), color(of the border) and fill(boolean variable specifying fill should be done). If fill is true, additional option called fillColor is also required, if not specified default of black fill is used.

Eg. rectangle((10,10),5, 10,"thickness":1, "color":"FFFFFF","fill":false)

circle(*point center, num radius, hash options*)

This draws a circle with the given radius, with any given point as center. It takes additional options as thickness, color and fill. If the circle goes out of bounds, the out of bound parts are neglected. The options we are considering now are simple like, thickness(of the border), color(of the border) and fill(boolean variable specifying fill should be done). If fill is true, additional option called fillColor is also required, if not specified default of black fill is used.

Eg. circle((10,10),5, "thickness":1, "color":"FFFFFF", "fill":true, "fillColor":"00FF00")

barGraph(*list data, hash options*)

This draws a barGraph given the data and specified options. The options include, num max-width, num max-height, num color (of the bar), xaxis(boolean representing whether the axis should be there), bool yaxis, string xLabel, string yLabel, bool grid(to have/not have grid lines) and string title (For example: padding, max width, max height, etc.)

Eg.

hash options = { "xAxis": true, "yAxis": true, "xLabel": "Time", "yLabel": "Celcius", "title": "Temperature with Time", "grid": true }

barGraph([5,1,4,2,7], options)

We will have more functions, specific to each type of plot (For example: line plot, pie charts, etc.) similar to the barGraph function, for the user to call.

The purpose of providing these built-in functions is to demonstrate the power of our language, enabling the user to build his own custom functions, as desired. Since most of these built in functions are written in plotter

9 Sample Program

This is a sample program that gets data from a csv file, applies some options to the data and plots the data in a bar-graph. We also supply sample code for the bar-graph function, written in pLOtter, which will be provided as an in-built function to the user, to show the ability to build custom graphs/plots using our language.

```
1 # Function written by user
2 fn convertToCelcius(num f):
3     return (f-32)/1.8
4 end
5
6 # Importing data from CSV
7 list data = getDataFromAsListCSV("data.csv")
8
9 # Data manipulations Converting Farenheit to Celcius
10 num i
11 for(i=0;i<data.length();i++):
12     data[i]=convertToCelcius(data[i])
13 end
14
15 hash options = {"xAxis": true, "yAxis": true, "xLabel": "Time", "yLabel": "
    Celcius", "title": "Temperature with Time", "grid": true}
16
```

```

17 # Calling in-built function (which could alternatively be implemented by
    user)
18 barGraph(data, options)

```

Listing 1: Data-manipulation and Bar Graph Example

The output of the above code will be like something below.

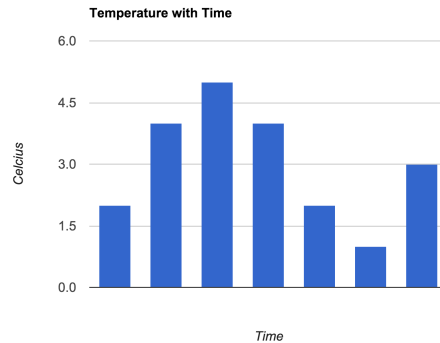


Figure 1: Simple Bar Graph

The function `barGraph` is built using primitive blocks and is provided as a built-in function. However, its shown here to emphasize the power user can have for creating his own graph.

```

1 #Bar Graph
2 fn barGraph(list data, hash options):
3     num mx
4     point pMax, pad = (10,10)
5
6     #Apply Options specified by user, if not specified, use defaults
7     options = applyOptions(options)
8
9     #Setting size set by the user or taking default
10    pMax[0] = if "xMax" in options then options.xMax else 1080
11    pMax[1] = if "yMax" in options then options.yMax else 1920
12
13
14    #Scaling data to Fit on display appropriately
15    mx = max(data)
16    options.gap = (pMax[0] - pad[1]) / (options.barWidth * len(data))
17
18    for i in data:
19        rectangle( (st[0], (pMax[1] - pad[1]) * i / mx), barWidth, barHeight,
20        options.barStyle)
21        pad[0] += barWidth + gap
22    end
23 end

```

Listing 2: Snippet of code for in-built `barGraph` function in `plotter`.