StockX

Jesse Van Marter jjv2121
Ravie Lakshmanan rl2857
Ricardo Martinez rmm2222
Sophie Lucy sjl2185

Language Reference Manual

Contents

Introduction

The financial industry has been a perfect space for the use of programming languages because of the heavy reliance on mathematical calculation and accuracy involved in gaining successful returns. Technology has allowed people to be more precise in their trades and experimental in their strategies. Thanks to the amount of historical as well as real time stock market data publically available, there are infinite possibilities to the application of a financial programming language. By using such resources, we intend to build a language that make it easy for the user to implement complex algorithms to perform sophisticated financial analysis.

Lexical Conventions

Tokens

Tokens are divided into identifiers, operators, separators, whitespace and reserved words.

Identifiers

Identifiers indicate function or variable name. StockX identifiers are case-sensitive.

Comments

Both single and multi-line comments are supported in StockX

e.g. `//this is a single line comment`

```
/*this is a
multi-line comment/*
```

Whitespace

Whitespace is ignored in StockX.

Separators

`;`	statement delimiter
`{}`	function body separator
`[]`	indication of array
`()`	indication of list of argument(s)

Reserved Words and Symbols

Data Types

`int`
`float`
`bool`
`null`
`string`
`stock`

`order`
`portfolio`
`struct`
`array`
`true`
`false`

Boolean Logic Operators

`and`
`or`
`not`

Branch Control and Loops

`if`
`else`
`for`
`while`
`return`

Functions

`function`
`void`
`return`

Built-in Functions

`delta`
`stddev`
`correlation`
`covariance`
`regression`
`ewma`
`lma`

Data Types

`int`

- a string of numeric characters without a decimal point, and an optional sign character

`float`

- a string of numeric characters that can be before and/or after a decimal point, with an optional sign character

`bool`

- a binary variable where value can be either `true` or `false`

`null`

- the value of an uninitialized object

`string`

- a finite sequence of ASCII characters, enclosed in double quotes

e.g. `"we are stockx"`

`struct`

A struct is a structure that allows the user to create their own custom data-type. A struct may hold variables and/or functions. The syntax for creating a struct is as follows:

```
struct studentStruct = {
      name  : string,
      age   : int,
      func : myfunc //where myfunc is a built-in or user-defined
                    //function
}
```

To access struct fields, use a dot in between the struct and its fields.

```
e.g.  int myAge = studentStruct.age;
      studentStruct.func();
```

`array`

- arrays only hold elements of the same type

e.g. `array string whoweare = ["we", "are", "stoc kx"];`

Add
- <identifier>.add(<element>)

Remove
- <identifier>.remove(<index>)

Access
- <identifier>.at(<index>)

Minimum
- <identifier>.min()

Maximum
- <identifier>.max()

Average
- <identifier>.avg()

`stock`

A type that contains the stock symbol and its associated attributes like stock price at any given time passed as an array of one or more elements.

```
e.g. stock goog = {
 name  : "GOOG",
 prices: [ 620.05, 625.50, 630.50 ]
}
```

Individual fields can be accessed by prefixing the field with the stock type and a dot.

`e.g. goog.name, goog.price`

`delta()` function - Given a stock and an array of stock prices at different times, the delta() function calculates the price variations in the stock. So for an array of stock prices length n, delta() would return n-1 elements.

`e.g. delta(google.prices)` calculates the difference of the two price values passed to return the delta list [5.45, 5.50].

`Order`

A type that indicates a buy or sell order on a stock. The type has a `.stddev()` function that calculates the standard deviation of an investment. The larger the value, the riskier the investment is deemed, and implies the stock needs to be sold. On the other hand, if the standard deviation value is low, it can be be bought.
It contains

- <identifier>.shares: the number of shares to be bought or sold
- <identifier>.stock: a stock type

- <identifier>.price: the price of the stock at the time the order is placed
- <identifier>.time: the timestamp at which the order is executed

`portfolio`

A type that holds the order history and thus the list of stocks owned.

Branch Control

`if`

- conditional `if` statement are followed by a boolean expression

e.g. `if(x==2){return true}`

`else`

- An `else` statement may follow an `if` statement
- A statement list then follows

e.g. `if(x==2) {retu rn true} else{return false}`

`for`

The `for` statement allows for looping over a range of values. The format is as follows:

e.g. `for (initiali zation; termination; update) { stmt }`

The initialization begins the for statement and is executed only once (before the loop begins).

The termination is a boolean expression that is checked for before each loop. When it returns false, the loop terminates.

The update is an expression that occurs once after each loop, and should modify the variable(s) being checked for in the termination.

`while`

The `while` statement is used for looping so long as a boolean expression inside of the while statement evaluates to true. The syntax is as follows:

```
e.g. while (expression) { stmt }
```

`return`

The return keyword is used both in function declarations, and inside of functions to return a value. The syntax is as follows:

```
return expr;
```

The return keyword may also be used by itself, indicating a return type of void. This looks like:

```
return;
```

Expressions

Declaration and Assignment

The general syntax is as follows:

- <type> <identifier>;
- examples: `int x; float number; string name;`
- Arrays can be declared as follows - array <type> <identifier>;
  e.g. `array string whoweare = ["we", "are", "stockx"];`

Arithmetic Operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus
`+=`	increment
`-=`	decrement
`*=`	multiply and assignment
`/=`	division and assignment
`%=`	modulus and assignment
`^`	power

Relational Operators

`=`	equal to
`==`	logical equal to
`!=`	not equal to
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to

Logic Operators

`and`

- logical intersection of two expressions

e.g. 0 and 1 evaluates to 0(false)

`or`

- logical union of two expressions

e.g. 1 or 0 evaluates to 1(true)

`not`

- logical negation of an expression

e.g. not 1 evaluates to 0

Functions

`function`
- establishes a user-defined function that will return a value

`void`
- when the function does not return a value

`return`
- caller of the function

Built-in Functions

`map`
- A function that takes two arguments - a function and an array, and applies the function to each element in the array.
- e.g. `map(delta(), [[80,90,100],[20, 30]]);`

`delta`
- Calculates and returns the difference in a time series array.
- `delta([20,30,50])` would return [10,20]

`stddev`
- Calculates and returns the standard deviation of an array.
- `stddev(x)` where x is array

`correlation`
- Given two different arrays, `correlation()` function calculates the correlation coefficient. The value of correlation coefficient is between -1 and 1. A positive correlation indicates both arrays move in the same direction, whereas a negative correlation corresponds to arrays moving in opposite direction.
- `correlation(x,y)` where x and y are arrays

`covariance`
- Given two different arrays, `covariance()` function calculates the covariance between two stocks. A positive value indicates a positive correlation and vice-versa.
- `covariance(x,y)` where x and y are arrays

`regression`

- Given an explanatory variable array x and dependent variable array y and a number n, `regression()` function computes the expected dependent y-value at x=n using linear regression
- `regression(x,y, 11)` where x and y are arrays will compute the linearly regressed "expected" value of y at x=11.

`ewma`

- Calculates exponentially weighted moving average of time series array. Takes in array of a time series and a center of mass. Center_of_mass= (length_of_ema-1)/2.
  $\alpha = 1/(1 + com)$
- `ewma(goog.prices, 9.5)` for a 20day ewma

`lma`

- Calculates linear moving average of time series array. Takes in time series array, length of average, and boolean for weighted moving average. If True, will return weighted moving average. If false will return simple moving average.
- `lma(goog.prices, 10, true)` will return a 10 day weighted moving average of google prices.

Function Definition

Functions are defined in the following format:

```
function <identifier> (<argument list>) return (<data type>){
      <statement list>
}
```

e.g.
```
function max( float a, float b) return (flo at) {
      //Statement list
      if(a > b ) {return a;} else{return b;};
}
```