Methods

Note

This page contains advanced material. If you are a beginning Pine Script™ programmer, we recommend that you become familiar with other, more accessible Pine Script™ features before you venture here.

Introduction

Pine Script™ methods are specialized functions associated with specific instances of built-in or user-defined types. They are essentially the same as regular functions in most regards but offer a shorter, more convenient syntax. Users can access methods using dot notation on variables directly, just like accessing the fields of a Pine Script™ object.

Built-in methods

Pine Script™ includes built-in methods for array, matrix, line, linefill, label, box, and table types. These methods provide users with a more concise way to call specialized routines for these types within their scripts.

When using these special types, the expressions

```
<namespace>.<functionName>([paramName =] <objectName>, …)
```

and

```
<objectName>.<functionName>(…)
```

are equivalent. For example, rather than using

```
array.get(id, index)
```

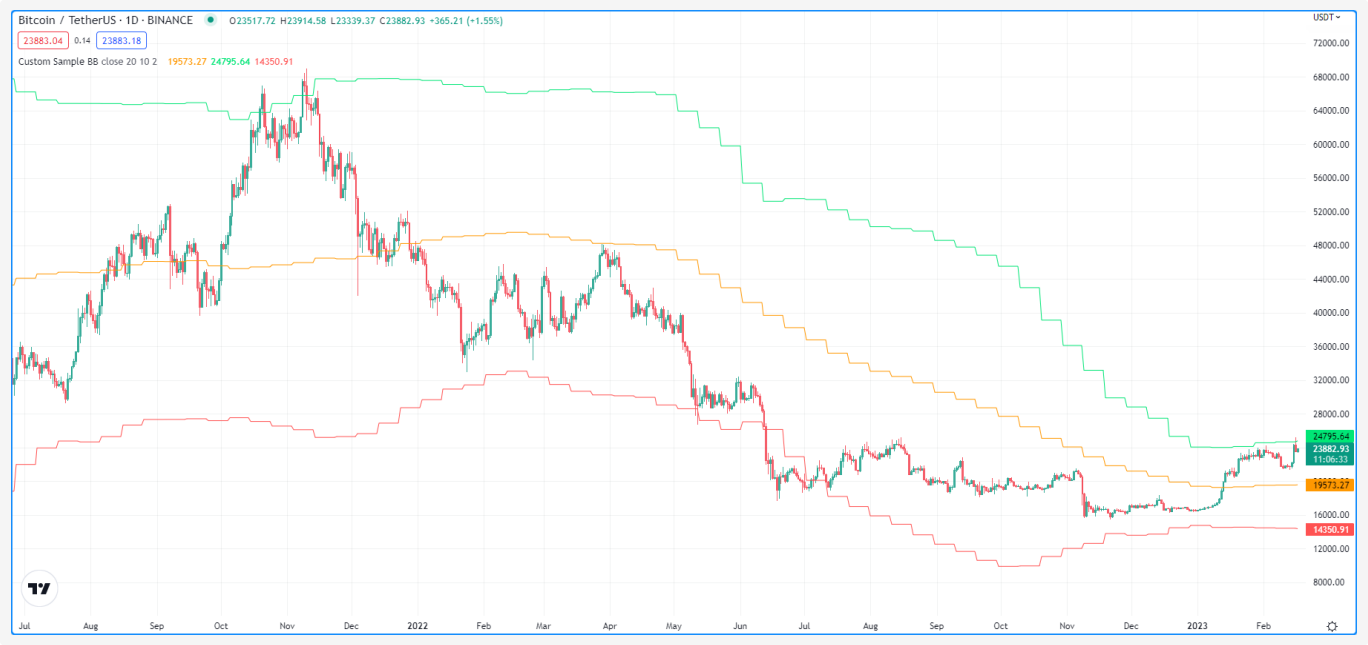
to get the value from an array `id` at the specified `index`, we can simply use

```
id.get(index)
```

to achieve the same effect. This notation eliminates the need for users to reference the function's namespace, as get() is a method of `id` in this context.

Written below is a practical example to demonstrate the usage of built-in methods in place of functions.

The following script computes Bollinger Bands over a specified number of prices sampled once every `n` bars. It calls array.push() and array.shift() to queue `sourceInput` values through the `sourceArray`, then array.avg() and array.stdev() to compute the `sampleMean` and `sampleDev`. The script then uses these values to calculate the `highBand` and `lowBand`, which it plots on the chart along with the `sampleMean`:



```
//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput  = input.source(close, "Source")
int   samplesInput = input.int(20, "Samples")
int   n            = input.int(10, "Bars")
float multiplier   = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)
var float        sampleMean  = na
var float        sampleDev   = na

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    array.push(sourceArray, sourceInput)
    array.shift(sourceArray)
    // Update the mean and standard deviaiton values.
    sampleMean := array.avg(sourceArray)
    sampleDev  := array.stdev(sourceArray) * multiplier

// Calculate bands.
float highBand = sampleMean + sampleDev
float lowBand  = sampleMean - sampleDev

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)
```

Let's rewrite this code to utilize methods rather than built-in functions. In this version, we have replaced all built-in array.* functions in the script with equivalent methods:

```
//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput  = input.source(close, "Source")
int   samplesInput = input.int(20, "Samples")
int   n            = input.int(10, "Bars")
float multiplier   = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)
var float        sampleMean  = na
var float        sampleDev   = na

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.push(sourceInput)
    sourceArray.shift()
    // Update the mean and standard deviaiton values.
    sampleMean := sourceArray.avg()
    sampleDev  := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand = sampleMean + sampleDev
float lowBand  = sampleMean - sampleDev

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)
```

Note that:

- We call the array methods using `sourceArray.*` rather than referencing the array namespace.
- We do not include `sourceArray` as a parameter when we call the methods since they already reference the object.

User-defined methods

Pine Script™ allows users to define custom methods for use with objects of any built-in or user-defined type. Defining a method is essentially the same as defining a function, but with two key differences:

- The method keyword must be included before the function name.
- The type of the first parameter in the signature must be explicitly declared, as it represents the type of object that the method will be associated with.

```
[export] method <functionName>(<paramType> <paramName> [= <defaultValue>], …) =>
    <functionBlock>
```

Let's apply user-defined methods to our previous Bollinger Bands example to encapsulate operations from the global scope, which will simplify the code and promote reusability. See this portion from the example:

```
// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.push(sourceInput)
    sourceArray.shift()
    // Update the mean and standard deviaiton values.
    sampleMean := sourceArray.avg()
    sampleDev  := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand = sampleMean + sampleDev
float lowBand  = sampleMean - sampleDev
```

We will start by defining a simple method to queue values through an array in a single call.

This `maintainQueue()` method invokes the push() and shift() methods on a `srcArray` when `takeSample` is true and returns the object:

```
// @function         Maintains a queue of the size of `srcArray`.
//                   It appends a `value` to the array and removes its oldest element a
// @param srcArray   (array<float>) The array where the queue is maintained.
// @param value      (float) The new value to be added to the queue.
//                   The queue's oldest value is also removed, so its size is constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this is true
// @returns          (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray
```

Note that:

- Just as with user-defined functions, we use the `@function` compiler annotation to document method descriptions.

Now we can replace `sourceArray.push()` and `sourceArray.shift()` with `sourceArray.maintainQueue()` in our example:

```
// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.maintainQueue(sourceInput)
    // Update the mean and standard deviaiton values.
    sampleMean  := sourceArray.avg()
    sampleDev   := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand  = sampleMean + sampleDev
float lowBand   = sampleMean - sampleDev
```

From here, we will further simplify our code by defining a method that handles all Bollinger Band calculations within its scope.

This `calcBB()` method invokes the avg() and stdev() methods on a `srcArray` to update `mean` and `dev` values when `calculate` is true. The method uses these values to return a tuple containing the basis, upper band, and lower band values respectively:

```
// @function         Computes Bollinger Band values from an array of data.
// @param srcArray   (array<float>) The array where the queue is maintained.
// @param multiplier (float) Standard deviaiton multiplier.
// @param calcuate   (bool) The method will only calculate new values when this is true
// @returns          A tuple containing the basis, upper band, and lower band respectiv
method calcBB(array<float> srcArray, float mult, bool calculate = true) =>
    var float mean = na
    var float dev  = na
    if calculate
        // Compute the mean and standard deviation of the array.
        mean := srcArray.avg()
        dev  := srcArray.stdev() * mult
    [mean, mean + dev, mean - dev]
```

With this method, we can now remove Bollinger Band calculations from the global scope and improve code readability:

```
// Identify if `n` bars have passed.
bool newSample = bar_index % n == 0

// Update the queue and compute new BB values on each new sample.
[sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput, newSample).cal
```

Note that:

- Rather than using an `if` block in the global scope, we have defined a `newSample` variable that is only true once every `n` bars. The `maintainQueue()` and `calcBB()` methods use this value for their respective `takeSample` and `calculate` parameters.
- Since the `maintainQueue()` method returns the object that it references, we're able to call `calcBB()` from the same line of code, as both methods apply to `array<float>` instances.

Here is how the full script example looks now that we've applied our user-defined methods:

```
//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput  = input.source(close, "Source")
int   samplesInput = input.int(20, "Samples")
int   n            = input.int(10, "Bars")
float multiplier   = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)

// @function         Maintains a queue of the size of `srcArray`.
//                   It appends a `value` to the array and removes its oldest element a
// @param srcArray   (array<float>) The array where the queue is maintained.
// @param value      (float) The new value to be added to the queue.
//                   The queue's oldest value is also removed, so its size is constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this is true
// @returns          (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray

// @function         Computes Bollinger Band values from an array of data.
// @param srcArray   (array<float>) The array where the queue is maintained.
// @param multiplier (float) Standard deviaiton multiplier.
// @param calcuate   (bool) The method will only calculate new values when this is true
// @returns          A tuple containing the basis, upper band, and lower band respectiv
method calcBB(array<float> srcArray, float mult, bool calculate = true) =>
    var float mean = na
    var float dev  = na
    if calculate
        // Compute the mean and standard deviation of the array.
        mean := srcArray.avg()
        dev  := srcArray.stdev() * mult
    [mean, mean + dev, mean - dev]

// Identify if `n` bars have passed.
bool newSample = bar_index % n == 0

// Update the queue and compute new BB values on each new sample.
[sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput, newSample).cal

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)
```

Method overloading

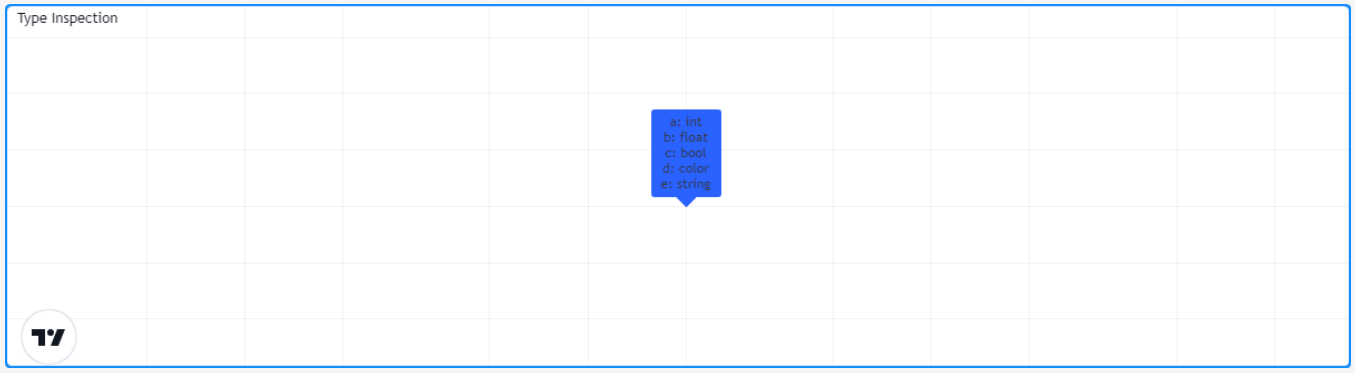
User-defined methods can override and overload existing built-in and user-defined methods with the same identifier. This capability allows users to define multiple routines associated with different parameter signatures under the same method name.

As a simple example, suppose we want to define a method to identify a variable's type. Since we must explicitly specify the type of object associated with a user-defined method, we will need to define overloads for each type that we want it to recognize.

Below, we have defined a `getType()` method that returns a string representation of a variable's type with overloads for the five primitive types:

```
// @function   Identifies an object's type.
// @param this Object to inspect.
// @returns    (string) A string representation of the type.
method getType(int this) =>
    na(this) ? "int(na)" : "int"

method getType(float this) =>
    na(this) ? "float(na)" : "float"

method getType(bool this) =>
    na(this) ? "bool(na)" : "bool"

method getType(color this) =>
    na(this) ? "color(na)" : "color"

method getType(string this) =>
    na(this) ? "string(na)" : "string"
```

Now we can use these overloads to inspect some variables. This script uses str.format() to format the results from calling the `getType()` method on five different variables into a single `results` string, then displays the string in the `lbl` label using the built-in set_text() method:



```
//@version=5
indicator("Type Inspection")

// @function   Identifies an object's type.
// @param this Object to inspect.
// @returns    (string) A string representation of the type.
method getType(int this) =>
    na(this) ? "int(na)" : "int"

method getType(float this) =>
    na(this) ? "float(na)" : "float"

method getType(bool this) =>
    na(this) ? "bool(na)" : "bool"

method getType(color this) =>
    na(this) ? "color(na)" : "color"

method getType(string this) =>
    na(this) ? "string(na)" : "string"

a = 1
b = 1.0
c = true
d = color.white
e = "1"

// Inspect variables and format results.
results = str.format(
 "a: {0}\nb: {1}\nc: {2}\nd: {3}\ne: {4}",
 a.getType(), b.getType(), c.getType(), d.getType(), e.getType()
 )

var label lbl = label.new(0, 0)
lbl.set_x(bar_index)
lbl.set_text(results)
```

Note that:

- The underlying type of each variable determines which overload of `getType()` the compiler will use.
- The method will append “(na)” to the output string when a variable is `na` to demarcate that it is empty.

Advanced example

Let’s apply what we’ve learned to construct a script that estimates the cumulative distribution of elements in an array, meaning the fraction of elements in the array that are less than or equal to any given value.

There are many ways in which we could choose to tackle this objective. For this example, we will start by defining a method to replace elements of an array, which will help us count the occurrences of elements within a range of values.

Written below is an overload of the built-in fill() method for `array<float>` instances. This overload replaces elements in a `srcArray` within the range between the `lowerBound` and `upperBound` with an `innerValue`, and replaces all elements outside the range with an `outerValue`:

```
// @function          Replaces elements in a `srcArray` between `lowerBound` and `upper
//                    and replaces elements outside the range with an `outerValue`.
// @param srcArray    (array<float>) Array to modify.
// @param innerValue  (float) Value to replace elements within the range with.
// @param outerValue  (float) Value to replace elements outside the range with.
// @param lowerBound  (float) Lowest value to replace with `innerValue`.
// @param upperBound  (float) Highest value to replace with `innerValue`.
// @returns           (array<float>) `srcArray` object.
method fill(array<float> srcArray, float innerValue, float outerValue, float lowerBound
    for [i, element] in srcArray
        if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or na(u
            srcArray.set(i, innerValue)
        else
            srcArray.set(i, outerValue)
    srcArray
```

With this method, we can filter an array by value ranges to produce an array of occurrences. For example, the expression

```
srcArray.copy().fill(1.0, 0.0, min, val)
```

copies the `srcArray` object, replaces all elements between `min` and `val` with 1.0, then replaces all elements above `val` with 0.0. From here, it's easy to estimate the output of the cumulative distribution function at the `val`, as it's simply the average of the resulting array:

```
srcArray.copy().fill(1.0, 0.0, min, val).avg()
```

Note that:

- The compiler will only use this `fill()` overload instead of the built-in when the user provides `innerValue`, `outerValue`, `lowerBound`, and `upperBound` arguments in the call.
- If either `lowerBound` or `upperBound` is `na`, its value is ignored while filtering the fill range.
- We are able to call `copy()`, `fill()`, and `avg()` successively on the same line of code because the first two methods return an `array<float>` instance.

We can now use this to define a method that will calculate our empirical distribution values. The following `eCDF()` method estimates a number of evenly spaced ascending `steps` from the cumulative distribution function of a `srcArray` and pushes the results into a `cdfArray`:

```
// @function       Estimates the empirical CDF of a `srcArray`.
// @param srcArray (array<float>) Array to calculate on.
// @param steps    (int) Number of steps in the estimation.
// @returns        (array<float>) Array of estimated CDF ratios.
method eCDF(array<float> srcArray, int steps) =>
    float min = srcArray.min()
    float rng = srcArray.range() / steps
    array<float> cdfArray = array.new<float>()
    // Add averages of `srcArray` filtered by value region to the `cdfArray`.
    float val = min
    for i = 1 to steps
        val += rng
        cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg())
    cdfArray
```

Lastly, to ensure that our `eCDF()` method functions properly for arrays containing small and large values, we will define a method to normalize our arrays.

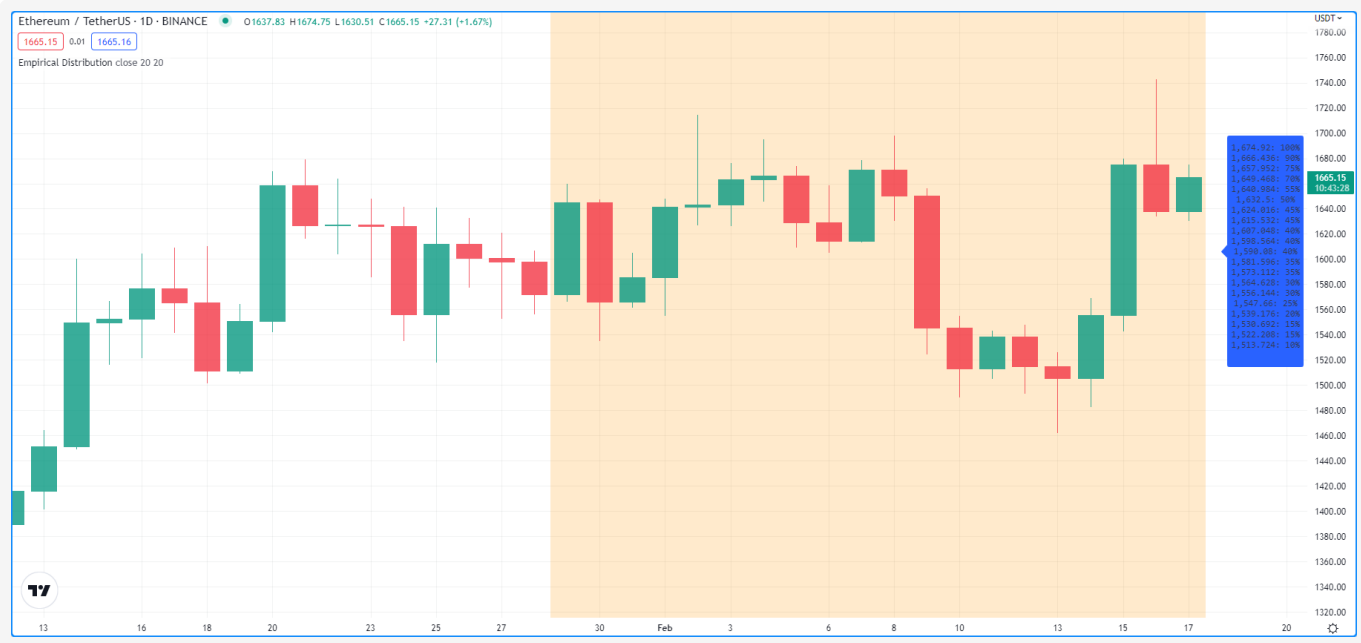
This `featureScale()` method uses array min() and range() methods to produce a rescaled copy of a `srcArray`. We will use this to normalize our arrays prior to invoking the `eCDF()` method:

```
// @function        Rescales the elements within a `srcArray` to the interval [0, 1].
// @param srcArray  (array<float>) Array to normalize.
// @returns         (array<float>) Normalized copy of the `srcArray`.
method featureScale(array<float> srcArray) =>
    float min = srcArray.min()
    float rng = srcArray.range()
    array<float> scaledArray = array.new<float>()
    // Push normalized `element` values into the `scaledArray`.
    for element in srcArray
        scaledArray.push((element - min) / rng)
    scaledArray
```

Note that:

- This method does not include special handling for divide by zero conditions. If `rng` is 0, the value of the array element will be `na`.

The full example below queues a `sourceArray` of size `length` with `sourceInput` values using our previous `maintainQueue()` method, normalizes the array's elements using the `featureScale()` method, then calls the `eCDF()` method to get an array of estimates for `n` evenly spaced steps on the distribution. The script then calls a user-defined `makeLabel()` function to display the estimates and prices in a label on the right side of the chart:




```
//@version=5
indicator("Empirical Distribution", overlay = true)

float sourceInput = input.source(close, "Source")
int length        = input.int(20, "Length")
int n             = input.int(20, "Steps")

// @function         Maintains a queue of the size of `srcArray`.
//                   It appends a `value` to the array and removes its oldest element a
// @param srcArray   (array<float>) The array where the queue is maintained.
// @param value      (float) The new value to be added to the queue.
//                   The queue's oldest value is also removed, so its size is constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this is true
// @returns          (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
```

```
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray

// @function          Replaces elements in a `srcArray` between `lowerBound` and `upper
//                    and replaces elements outside the range with an `outerValue`.
// @param srcArray    (array<float>) Array to modify.
// @param innerValue  (float) Value to replace elements within the range with.
// @param outerValue  (float) Value to replace elements outside the range with.
// @param lowerBound  (float) Lowest value to replace with `innerValue`.
// @param upperBound  (float) Highest value to replace with `innerValue`.
// @returns           (array<float>) `srcArray` object.
method fill(array<float> srcArray, float innerValue, float outerValue, float lowerBound
    for [i, element] in srcArray
        if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or na(u
            srcArray.set(i, innerValue)
        else
            srcArray.set(i, outerValue)
    srcArray

// @function       Estimates the empirical CDF of a `srcArray`.
// @param srcArray (array<float>) Array to calculate on.
// @param steps    (int) Number of steps in the estimation.
// @returns        (array<float>) Array of estimated CDF ratios.
method eCDF(array<float> srcArray, int steps) =>
    float min = srcArray.min()
    float rng = srcArray.range() / steps
    array<float> cdfArray = array.new<float>()
    // Add averages of `srcArray` filtered by value region to the `cdfArray`.
    float val = min
    for i = 1 to steps
        val += rng
        cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg())
    cdfArray

// @function        Rescales the elements within a `srcArray` to the interval [0, 1].
// @param srcArray  (array<float>) Array to normalize.
// @returns         (array<float>) Normalized copy of the `srcArray`.
method featureScale(array<float> srcArray) =>
    float min = srcArray.min()
    float rng = srcArray.range()
    array<float> scaledArray = array.new<float>()
    // Push normalized `element` values into the `scaledArray`.
    for element in srcArray
        scaledArray.push((element - min) / rng)
    scaledArray

// @function        Draws a label containing eCDF estimates in the format "{price}: {pe
// @param srcArray  (array<float>) Array of source values.
// @param cdfArray  (array<float>) Array of CDF estimates.
// @returns         (void)
makeLabel(array<float> srcArray, array<float> cdfArray) =>
    float max      = srcArray.max()
    float rng      = srcArray.range() / cdfArray.size()
    string results = ""
    var label lbl  = label.new(0, 0, "", style = label.style_label_left, text_font_fami
    // Add percentage strings to `results` starting from the `max`.
    cdfArray.reverse()
    for [i, element] in cdfArray
        results += str.format("{0}: {1}%\n", max - i * rng, element * 100)
    // Update `lbl` attributes.
    lbl.set_xy(bar_index + 1, srcArray.avg())
    lbl.set_text(results)

var array<float> sourceArray = array.new<float>(length)
```

```
var array<float> sourceArray = array.new<float>(length)

// Add background color for the last `length` bars.
bgcolor(bar_index > last_bar_index - length ? color.new(color.orange, 80) : na)

// Queue `sourceArray`, feature scale, then estimate the distribution over `n` steps.
array<float> distArray = sourceArray.maintainQueue(sourceInput).featureScale().eCDF(n)
// Draw label.
makeLabel(sourceArray, distArray)
```

Objects

Concepts

Options v: v5