Type system

- Introduction
  - Forms
  - Types
- Using forms and types
  - Forms
    - const
    - input
    - simple
    - series
  - Types
    - int
    - float
    - bool
    - color
    - string
    - plot and hline
    - line, linefill, label, box and table
    - Arrays and matrices
    - User-defined types
    - void
- `na` value
- Type templates
- Type casting
- Tuples

Introduction

Pine Script™'s type system is important because it determines what sort of values can be used when calling Pine Script™ functions, which is a requirement to do pretty much anything in Pine Script™. While it is possible to write very simple scripts without knowing anything about the type system, a reasonable understanding of it is necessary to achieve any degree of profiency with the language, and in-depth knowledge of its subtleties will allow you to exploit the full potential of Pine Script™.

The type system uses the *form type* pair to qualify the type of all values, be they literals, a variable, the result of an expression, the value returned by functions or the arguments supplied when calling a function.

The *form* expresses when a value is known.

The *type* denotes the nature of a value.

Note

We will often use “type” to refer to the “form type” pair.

The type system is intimately linked to Pine Script™’s execution model and time series concepts. Understanding all three is key to making the most of the power of Pine Script™.

Forms

Pine Script™ **forms** identify when a variable’s value is known. They are:

- “const” for values known at compile time (when adding an indicator to a chart or saving it in the Pine Script™ Editor)
- “input” for values known at input time (when values are changed in a script’s “Settings/Inputs” tab)
- “simple” for values known at bar zero (when the script begins execution on the chart’s first historical bar)
- “series” for values known on each bar (any time during the execution of a script on any bar)

Forms are organized in the following hierarchy: **const < input < simple < series**, where “const” is considered a *weaker* form than “input”, for example, and “series” *stronger* than “simple”. The form hierarchy translates into the rule that, whenever a given form is required, a weaker form is also allowed.

An expression’s result is always of the strongest form used in the expression’s calculation. Furthermore, once a variable acquires a stronger form, that state is irreversible; it can never be converted back to a weaker form. A variable of “series” form can thus never be converted back to a “simple” form, for use with a function that requires arguments of that form.

Note that of all these forms, only the “series” form allows values to change dynamically, bar to bar, during the script’s execution over each bar of the chart’s history. Such values include close or hlc3 or any variable calculated using values of “series” form. Variables of “const”, “input” or “simple” forms cannot change values once execution of the script has begun.

Types

Pine Script™ **types** identify the nature of a value. They are:

- The fundamental types: “int”, “float”, “bool”, “color” and “string”
- The special types: “plot”, “hline”, “line”, “linefill”, “label”, “box”, “table”, “array”, “matrix”
- User-defined types (UDTs)
- “void”

Each fundamental type refers to the nature of the value contained in a variable: `1` is of type “int”, `1.0` is of type “float”, `"AAPL"` is of type “string”, etc. Variables of special types contain an ID referring to an object of the type’s name. A variable of type “label” contains an ID (or *pointer*) referring to a label, and so on. The “void” type means no value is returned.

The Pine Script™ compiler can automatically convert some types into others when a value is not of the required type. The auto-casting rules are: **int** 🅰🅰 **float** 🅰🅰 **bool**. See the Type casting section of this page for more information on type casting.

Except for parameter definitions appearing in function signatures, Pine Script™ forms are implicit in code; they are never declared because they are always determined by the compiler. Types, however, can be specified when declaring variables, e.g.:

```
//@version=5
indicator("", "", true)
int periodInput = input.int(100, "Period", minval = 2)
float ma = ta.sma(close, periodInput)
bool xUp = ta.crossover(close, ma)
color maColor = close > ma ? color.lime : color.fuchsia
plot(ma, "MA", maColor)
plotchar(xUp, "Cross Up", "▲", location.top, size = size.tiny)
```

Using forms and types

Forms

const

Values of “const” form must be known at compile time, before your script has access to any information related to the symbol/timeframe information it is running on. Compilation occurs when you save a script in the Pine Script™ Editor, which doesn’t even require it to already be running on your chart. “const” variables cannot change during the execution of a script.

Variables of “const” form can be initialized using a *literal* value, or calculated from expressions using only literal values or other variables of “const” form. Our Style guide recommends using upper case SNAKE_CASE to name variables of “const” form. While it is not a requirement, “const” variables can be declared using the var keyword so they are only initialized on the first bar of the dataset. See the section on `var` for more information.

These are examples of literal values:

- *literal int*: `1`, `-1`, `42`
- *literal float*: `1.`, `1.0`, `3.14`, `6.02E-23`, `3e8`
- *literal bool*: `true`, `false`
- *literal string*: `"A text literal"`, `"Embedded single quotes 'text'"`, `'Embedded double quotes "text"'`
- *literal color*: `#FF55C6`, `#FF55C6ff`

> **Note**
> In Pine Script™, the built-in variables `open`, `high`, `low`, `close`, `volume`, `time`, `hl2`, `hlc3`, `ohlc4`, etc., are of “series” form because their values can change bar to bar.

The “const” form is a requirement for the arguments to the `title` and `shorttitle` parameters in indicator(), for example. All these are valid variables that can be used as arguments for those parameters when calling the function:

```
//@version=5
NAME1 = "My indicator"
var NAME2 = "My Indicator"
var NAME3 = "My" + "Indicator"
var NAME4 = NAME2 + " No. 2"
indicator(NAME4, "", true)
plot(close)
```

This will trigger a compilation error:

```
//@version=5
var NAME = "My indicator for " + syminfo.type
indicator(NAME, "", true)
plot(close)
```

The reason for the error is that the `NAME` variable's calculation depends on the value of syminfo.type which is a "simple string" (syminfo.type returns a string corresponding to the sector the chart's symbol belongs to, eg., `"crypto"`, `"forex"`, etc.).

Note that using the `:=` operator to assign a new value to a previously declared "const" variable will transform it into a "simple" variable, e.g., here with `name1`, for which we do not use an uppercase name because it is not of "const" form:

```
var name1 = "My Indicator "
var NAME2 = "No. 2"
name1 := name1 + NAME2
```

input

Values of "input" form are known when the values initialized through `input.*()` functions are determined. These functions determine the values that can be modified by script users in the script's "Settings/Inputs" tab. When these values are changed, this always triggers a re-execution of the script from the beginning of the chart's history (bar zero), so variables of "input" form are always known when the script begins execution, and they do not change during the script's execution.

> **Note**
> The input.source() function yields a value of "series" type — not "input". This is because built-in variables such as open, high, low, close, hl2, hlc3, ohlc4, etc., are of "series" form.

The script plots the moving average of a user-defined source and period from a symbol and timeframe also determined through inputs:

```
//@version=5
indicator("", "", true)
symbolInput = input.symbol("AAPL", "Symbol")
timeframeInput = input.timeframe("D", "Timeframe")
sourceInput = input.source(close, "Source")
periodInput = input(10, "Period")
v = request.security(symbolInput, timeframeInput, ta.sma(sourceInput, periodInput))
plot(v)
```

Note that:

- The `symbolInput`, `timeframeInput` and `periodInput` variables are of "input" form.
- The `sourceInput` variable is of "series" form because it is determined from a call to input.source().
- Our request.security() call is valid because its `symbol` and `timeframe` parameters require a "simple" argument and the "input" form we use is weaker than "simple". The function's `expression` parameter requires a "series" form argument, and that is what form our `sourceInput` variable is. Note that because a "series" form is required there, we could have used "const", "input" or "simple" forms as well.
- As per our style guide's recommendations, we use the "Input" suffix with our input variables to help readers of our code remember the origin of these variables.

Wherever an "input" form is required, a "const" form can also be used.

simple

Values of "simple" form are known only when a script begins execution on the first bar of a chart's history, and they never change during the execution of the script. Built-in variables of the `syminfo.*`, `timeframe.*` and `ticker.*` families, for example, all return results of "simple" form because their value depends on the chart's symbol, which can only be detected when the script executes on it.

A "simple" form argument is also required for the `length` argument of functions such as ta.ema() or ta.rma() which cannot work with dynamic lengths that could change during the script's execution.

Wherever a "simple" form is required, a "const" or "input" form can also be used.

series

Values of "series" form (also sometimes called *dynamic*) provide the most flexibility because they can change on any bar, or even multiples times during the same bar, in loops for example. Built-in variables such as open, close, high, time or volume are of "series" form, as would be the result of expressions calculated using them. Functions such as barssince() or crossover() yield a result of "series" form because it varies bar to bar, as does that of the `[]` history-referencing operator used to access past values of a time series. While the "series" form is the most common form used in Pine Script™, it is not always allowed as arguments to Pine Script™ built-in functions.

Suppose you want to display the value of pivots on your chart. This will require converting values into strings, so the string values your code will be using will be of "series string" type. Pine Script™'s label.new() function can be used to place such "series string" text on the chart because its `text` parameter accepts arguments of "series" form:

```
//@version=5
indicator("", "", true)
pivotBarsInput = input(3)
hiP = ta.pivothigh(high, pivotBarsInput, pivotBarsInput)
if not na(hiP)
    label.new(bar_index[pivotBarsInput], hiP, str.tostring(hiP, format.mintick),
     style = label.style_label_down,
     color = na,
     textcolor = color.silver)
plotchar(hiP, "hiP", "•", location.top, size = size.tiny)
```

Note that:

- The `str.tostring(hiP, format.mintick)` call we use to convert the pivot's value to a string yields a "series string" result, which will work with label.new().
- While prices appear at the pivot, the pivots actually require `pivotBarsInput` bars to have elapsed before they can be detected. Pivot prices only appear on the pivot because we plot them in the past after the pivot's detection, using `bar_index[pivotBarsInput]` (the bar_index's value, offset `pivotBarsInput` bars back). In real time, these prices would only appear `pivotBarsInput` bars after the actual pivot.
- We print a blue dot using plotchar() when a pivot is detected in our code.
- Pine Script™'s plotshape() can also be used to position text on the chart, but because its `text` parameter requires a "const string" argument, we could not have used it in place of label.new() in our script.

Wherever a "series" form is required, a "const", "input" or "simple" form can also be used.

Types

int

Integer literals must be written in decimal notation, e.g.:

```
1
-1
750
```

Built-in variables such as bar_index, time, timenow, time_close, or dayofmonth all return values of type “int”.

float

Floating-point literals contain a delimiter (the symbol `.`) and may also contain the symbol `e` or `E` (which means “multiply by 10 to the power of X”, where X is the number after the symbol `e`), e.g.:

```
3.14159    // Rounded value of Pi (π)
- 3.0
6.02e23    // 6.02 * 10^23 (a very large value)
1.6e-19    // 1.6 * 10^-19 (a very small value)
```

The internal precision of floats in Pine Script™ is 1e-10.

bool

There are only two literals representing *bool* values:

```
true    // true value
false   // false value
```

When an expression of type “bool” returns na and it is used to test a conditional statement or operator, the “false” branch is executed.

color

Color literals have the following format: `#RRGGBB` or `#RRGGBBAA`. The letter pairs represent `00` to `FF` hexadecimal values (0 to 255 in decimal) where:

- `RR`, `GG` and `BB` pairs are the values for the color’s red, green and blue components
- `AA` is an optional value for the color’s transparency (or *alpha* component) where `00` is invisible and `FF` opaque. When no `AA` pair is supplied, `FF` is used.
- The hexadecimal letters can be upper or lower case

Examples:

```
#000000      // black color
#FF0000      // red color
#00FF00      // green color
#0000FF      // blue color
#FFFFFF      // white color
#808080      // gray color
#3ff7a0      // some custom color
#FF000080    // 50% transparent red color
#FF0000ff    // same as #FF0000, fully opaque red color
#FF000000    // completely transparent color
```

Pine Script™ also has built-in color constants such as color.green, color.red, color.orange, color.blue (the default color used in plot() and other plotting functions), etc.

When using color built-ins, is possible to add transparency information to them with color.new.

Note that when specifying red, green or blue components in `color.*()` functions, a 0-255 decimal value must be

used. When specifying transparency in such functions, it is in the form of a 0-100 value (which can be of "float" type to access the underlying 255 potential valoues) where the scale 00-FF scale for color literals is inverted: 100 is thus invisible and 0 is opaque.

Here is an example:

```
//@version=5
indicator("Shading the chart's background", "", true)
BASE_COLOR = color.navy
bgColor = dayofweek == dayofweek.monday    ? color.new(BASE_COLOR, 50) :
          dayofweek == dayofweek.tuesday   ? color.new(BASE_COLOR, 60) :
          dayofweek == dayofweek.wednesday ? color.new(BASE_COLOR, 70) :
          dayofweek == dayofweek.thursday  ? color.new(BASE_COLOR, 80) :
          dayofweek == dayofweek.friday    ? color.new(BASE_COLOR, 90) :
          color.new(color.blue, 80)
bgcolor(bgColor)
```

See the page on colors for more information on using colors in Pine Script™.

string

String literals may be enclosed in single or double quotation marks, e.g.:

```
"This is a double quoted string literal"
'This is a single quoted string literal'
```

Single and double quotation marks are functionally equivalent. A string enclosed within double quotation marks may contain any number of single quotation marks, and vice versa:

```
"It's an example"
'The "Star" indicator'
```

You can escape the string's delimiter in the string by using a backslash. For example:

```
'It\'s an example'
"The \"Star\" indicator"
```

You can concatenate strings using the `+` operator.

plot and hline

Pine Script™'s fill() function fills the space between two lines with a color. Both lines must have been plotted with either plot() or hline() function calls. Each plotted line is referred to in the fill() function using IDs which are of "plot" or "hline" type, e.g.:

```
//@version=5
indicator("", "", true)
plotID1 = plot(high)
plotID2 = plot(math.max(close, open))
fill(plotID1, plotID2, color.yellow)
```

Note that there is no `plot` or `hline` keyword to explicitly declare the type of plot() or hline() IDs.

line, linefill, label, box and table

Drawings appeared in Pine Script™ starting with v4. Each drawing has its own type: line, linefill, label, box, table.

Each type is also used as a namespace containing all the built-in functions used to operate on each type of drawing. One of these is a `new()` constructor used to create an object of that type: line.new(), linefill.new(), label.new(), box.new() and table.new().

These functions all return an ID which is a reference that uniquely identifies a drawing object. IDs are always of "series" form, thus their form and type is "series line", "series label", etc. Drawing IDs act like a pointer in that they are used to reference a specific instance of a drawing in all the functions of that drawing's namespace. For example, the line ID returned by a line.new() call will then be used to refer to it when comes time to delete the line using line.delete().

Arrays and matrices

Arrays and matrices in Pine Script™ are identified by an ID, much like drawings such as lines. The type of the ID defines the type of elements contained in the array or matrix. Array and matrix types are specified by appending a type template to the array or matrix keywords:

- `array<int>` defines an array containing "int" elements.
- `array<label>` defines an array containing "label" IDs.
- `array<UDF>` defines an array containing objects of a user-defined type (UDT).
- `matrix<float>` defines a matrix containing "float" elements.
- `matrix<UDF>` defines a matrix containing objects of a user-defined type (UDT).

An array containing elements of type "int" initalized with one element of value 10 can be declared in the following, equivalent ways:

```
a1 = array.new<int>(1, 10)
array<int> a2 = array.new<int>(1, 10)
a3 = array.from(10)
array<int> a4 = array.from(10)
```

Note that the `int[]` syntax can also be used to declare an array of "int" elements, but that use is discouraged. No equivalent exists to specify the type of matrices in that way. Also note that type-specific built-ins such as array.new_int() also exist, but the more generic array.new<type> form is preferred, which would be `array.new<int>()` to create an array of "int" elements.

User-defined types

The type keyword allows the creation of *user-defined types* (UDTs) from which objects can be created. UDTs are composite types; they contain an arbitrary number of *fields* that can be of any type. The syntax to define a *user-defined type* is:

where:

- `export` is used to export the UDT from a library. See the Libraries page for more information.
- `<UDT_identifier>` is the name of the user-defined type.
- `<field_type>` is the type of the field.
- `<field_name>` is the name of the field.
- `<value>` is an optional default value for the field, which will be assigned to it when new objects of that UDT are created. The field's default value will be na if none is specified. The same rules as those governing the default values of parameters in function signatures apply to the default values of fields. For example, the [] history-referencing operator cannot be used with them, and expressions are not allowed.

In this example, we create a UDT containing two fields to hold pivot information, the time of the pivot's bar and its price level:

```
type pivotPoint
    int openTime
    float level
```

User-defined types can be embedded, so a field can be of the same type as the UDT it belongs to. Here, we add a field to our previous `pivotPoint` type that will hold the pivot information for another pivot point:

```
type pivotPoint
    int openTime
    float level
    pivotPoint nextPivot
```

Two built-in methods can be used with a UDT: `new()` and `copy()`. Read about them in the Objects page.

void

There is a "void" type in Pine Script™. Functions having only side-effects and returning no usable result return the "void" type. An example of such a function is alert(); it does something (triggers an alert event), but it returns no useful value.

A "void" result cannot be used in an expression or assigned to a variable. No `void` keyword exists in Pine Script™, as variables cannot be declared using the "void" type.

`na` value

In Pine Script™ there is a special value called na, which is an acronym for *not available*, meaning the value of an expression or variable is undefined. It is similar to the `null` value in Java, or `None` in Python.

na values can be automatically cast to almost any type. In some cases, however, the Pine Script™ compiler cannot automatically infer a type for an na value because more than one automatic type-casting rule can be applied. For example:

```
// Compilation error!
myVar = na
```

Here, the compiler cannot determine if `myVar` will be used to plot something, as in `plot(myVar)` where its type would be "float", or to set some text as in `label.set_text(lb, text = myVar)` where its type would be "string", or for some other purpose. Such cases must be explicitly resolved in one of two ways:

```
float myVar = na
```

or

```
myVar = float(na)
```

To test if some value is na, a special function must be used: na(). For example:

```
myClose = na(myVar) ? 0 : close
```

Do not use the `==` operator to test for na values, as this method is unreliable.

Designing your calculations so they are na-resistant is often useful. In this example, we define a condition that is

`true` when the bar's close is higher than the previous one. For this calculation to work correctly on the dataset's first bar where no previous close exists and `close[1]` will return na, we use the nz() function to replace it with the current bar's open for that special case:

```
bool risingClose = close > nz(close[1], open)
```

Protecting against na values can also be useful to prevent an initial na value from propagating in a calculation's result on all bars. This happens here because the initial value of `ath` is na, and math.max() returns na if one of its arguments is na:

```
// Declare `ath` and initialize it with `na` on the first bar.
var float ath = na
// On all bars, calculate the maximum between the `high` and the previous value of `ath
ath := math.max(ath, high)
```

To protect against this, we could instead use:

```
var float ath = na
ath := math.max(nz(ath), high)
```

where we are replacing any na values of `ath` with zero. Even better would be:

```
var float ath = high
ath := math.max(ath, high)
```

Type templates

Type templates are used to build array and matrix types. They are a type name enclosed in angle brackets, for example: `<int>`, `<label>`, `<pivotPoint>` (where `pivotPoint` is a user-defined type (UDT)). Type templates can be constructed from:

- Fundamental types: "int", "float", "bool", "color" and "string"
- The following special types: "line", "linefill", "label", "box", "table"
- User-defined types (UDTs)

They can be used to declare the type of a variable and in the array.new<type> or matrix.new<type> function calls used to create a new array or matrix:

```
// Declare an empty array variable.
var array<int> a = na
// Create an array of 10 "int" elements.
var a = array.new<int>(10)
```

Type casting

There is an automatic type-casting mechanism in Pine Script™ which can *cast* (or convert) certain types to another. The auto-casting rules are: **"int"** 🅰️ **"float"** 🅰️ **"bool"**, which means that when a "float" is required, an "int" can be used in its place, and when a "bool" value is required, an "int" or "float" value can be used in its place.

See auto-casting in action in this code:

```
//@version=5
indicator("")
plotshape(close)
```

Note that:

- plotshape() requires a "series bool" argument for its first parameter named `series`. The `true` / `false` value of that "bool" argument determines if the function plots a shape or not.
- We are here calling plotshape() with close as its first argument. This would not be allowed without Pine's auto-casting rules, which allow a "float" to be cast to a "bool". When a "float" is cast to a "bool", any non-zero values are converted to `true`, and zero values are converted to `false`. As a result of this, our code will plot an "X" on all bars, as long as close is not equal to zero.

It may sometimes be necessary to cast one type into another because auto-casting rules will not suffice. For these cases, explicit type-casting functions exist. They are: int(), float(), bool(), color(), string(), line(), linefill(), label(), box(), and table().

This is code that will not compile because we fail to convert the type of the argument used for `length` when calling ta.sma():

```
//@version=5
indicator("")
len = 10.0
s = ta.sma(close, len) // Compilation error!
plot(s)
```

The code fails to compile with the following error: *Cannot call 'ta.sma` with argument 'length'='len'. An argument of 'const float' type was used but a 'series int' is expected.* The compiler is telling us that we supplied a "float" value where an "int" is required. There is no auto-casting rule that can automatically cast a "float" to an "int", so we will need to do the job ourselves. For this, we will use the int() function to force the type conversion of the value we supply as a length to ta.sma() from "float" to "int":

```
//@version=5
indicator("")
len = 10.0
s = ta.sma(close, int(len))
plot(s)
```

Explicit type-casting can also be useful when declaring variables and initializing them to na which can be done in two ways:

```
// Cast `na` to the "label" type.
lbl = label(na)
// Explicitly declare the type of the new variable.
label lbl = na
```

Tuples

A *tuple* is a comma-separated set of expressions enclosed in brackets that can be used when a function or a local block must return more than one variable as a result. For example:

```
calcSumAndMult(a, b) =>
    sum = a + b
    mult = a * b
    [sum, mult]
```

```
    [sum, mult]
```

In this example there is a two-element tuple on the last statement of the function's code block, which is the result returned by the function. Tuple elements can be of any type. There is also a special syntax for calling functions that return tuples, which uses a *tuple declaration* on the left side of the equal sign in what is a multi-variable declaration. The result of a function such as `calcSumAndMult()` that returns a tuple must be assigned to a *tuple declaration*, i.e., a set of comma-separated list of *new* variables that will receive the values returned by the function. Here, the value of the `sum` and `mult` variables calculated by the function will be assigned to the `s` and `m` variables:

```
[s, m] = calcSumAndMul(high, low)
```

Note that the type of `s` and `m` cannot be explicitly defined; it is always inferred by the type of the function return results.

Tuples can be useful to request multiple values in one request.security() call:

```
roundedOHLC() =>
    [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_mintick(lo
[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", roundedOHLC())
```

or:

```
[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", [math.round_to_mintick(open)
```

or this form if no rounding is required

```
[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", [open, high, low, close])
```

Tuples can also be used as return results of local blocks, in an if statement for example:

```
[v1, v2] = if close > open
    [high, close]
else
    [close, low]
```

They cannot be used in ternaries, however, because the return values of a ternary statement are not considered as local blocks. This is not allowed:

```
// Not allowed.
[v1, v2] = close > open ? [high, close] : [close, low]
```

Please note that the items within a tuple returned from a function may be of simple or series form, depending on its contents. If a tuple contains a series value, all other elements within the tuple will also be of the series form. For example:

```
//@version=5
indicator("tuple_typeforms")

makeTicker(simple string prefix, simple string ticker) =>
    tId = prefix + ":" + ticker // simple string
    source = close  // series float
    [tId, source]

// Both variables are series now.
[tId, source] = makeTicker("BATS", "AAPL")

// Error cannot call 'request.security' with 'series string' tId.
r = request.security(tId, "", source)

plot(r)
```

Loops

Built-ins