# Inputs



## Introduction

Inputs allow scripts to receive values that users can change. Using them for key values will make your scripts more adaptable to user preferences.

The following script plots a 20-period simple moving average (SMA) using `ta.sma(close, 20)`. While it is simple to write, it is not very flexible because that specific MA is all it will ever plot:

```
//@version=5
indicator("MA", "", true)
plot(ta.sma(close, 20))
```

If instead we write our script this way, it becomes much more flexible because its users will be able to select the source and the length they want to use for the MA's calculation:

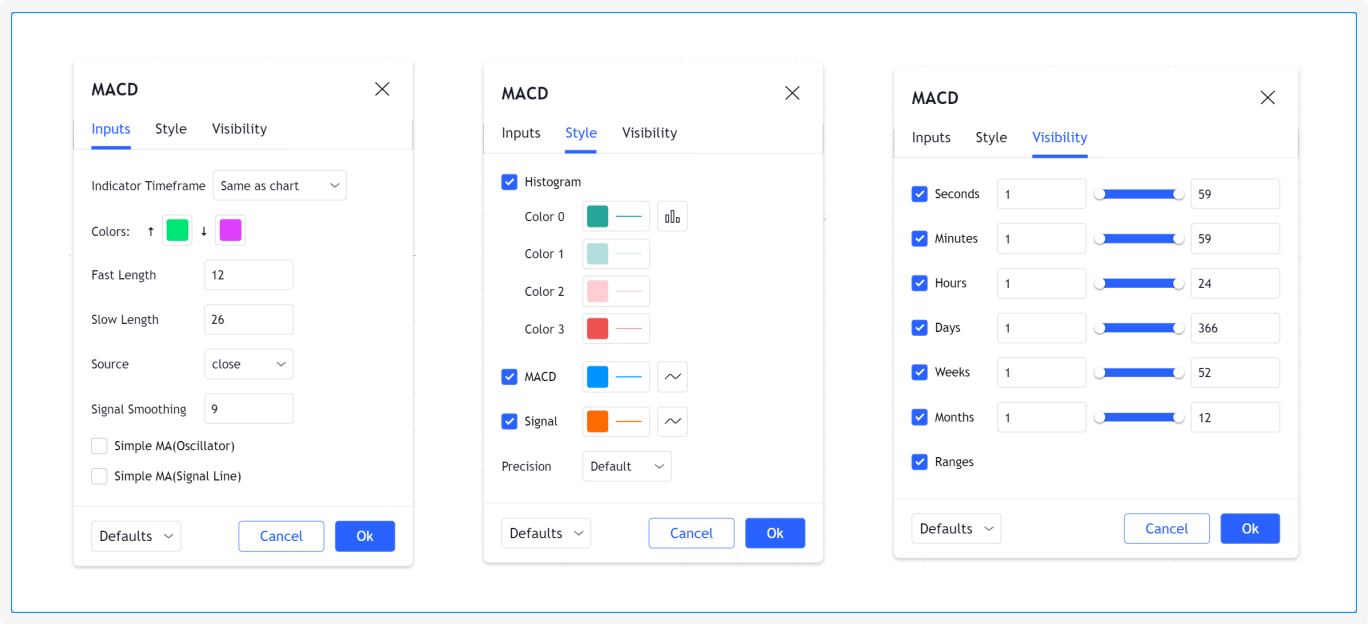
```
//@version=5
indicator("MA", "", true)
sourceInput = input(close, "Source")
lengthInput = input(20, "Length")
plot(ta.sma(sourceInput, lengthInput))
```

Inputs can only be accessed when a script is running on the chart. Script users access them through the script's "Settings" dialog box, which can be reached by either:

- Double-clicking on the name of an on-chart indicator
- Right-clicking on the script's name and choosing the "Settings" item from the dropdown menu
- Choosing the "Settings" item from the "More" menu icon (three dots) that appears when one hovers over the indicator's name on the chart
- Double-clicking on the indicator's name from the Data Window (fourth icon down to the right of the chart)

The "Settings" dialog box always contains the "Style" and "Visibility" tabs, which allow users to specify their preferences about the script's visuals and the chart timeframes where it should be visible.

When a script contains calls to `input.*()` functions, an "Inputs" tab appears in the "Settings" dialog box.




In the flow of a script's execution, inputs are processed when the script is already on a chart and a user changes values in the "Inputs" tab. The changes trigger a re-execution of the script on all the chart bars, so when a user changes an input value, your script recalculates using that new value.

## Input functions

The following input functions are available:

- input()
- input.int()
- input.float()
- input.bool()
- input.color()
- input.string()
- input.timeframe()
- input.symbol()
- input.price()
- input.source()
- input.session()
- input.time()

A specific input *widget* is created in the "Inputs" tab to accept each type of input. Unless otherwise specified in the `input.*()` call, each input appears on a new line of the "Inputs" tab, in the order the `input.*()` calls appear in the script.

Our Style guide recommends placing `input.*()` calls at the beginning of the script.

Input function definitions typically contain many parameters, which allow you to control the default value of inputs, their limits, and their organization in the “Inputs” tab.

An `input*.()` call being just another function call in Pine Script™, its result can be combined with arithmetic, comparison, logical or ternary operators to form an expression to be assigned to the variable. Here, we compare the result of our call to input.string() to the string `"On"`. The expression’s result is then stored in the `plotDisplayInput` variable. Since that variable holds a `true` or `false` value, it is a of “input bool” type:

```
//@version=5
indicator("Input in an expression`", "", true)
bool plotDisplayInput = input.string("On", "Plot Display", options = ["On", "Off"]) ==
plot(plotDisplayInput ? close : na)
```

All values returned by `input.*()` functions except “source” ones are of the “input” form (see the section on forms for more information).

## Input function parameters

The parameters common to all input functions are: `defval`, `title`, `tooltip`, `inline` and `group`. Some parameters are used by the other input functions: `options`, `minval`, `maxval`, `step` and `confirm`.

All these parameters expect arguments of “const” form (except if it’s an input used for a “source”, which returns a “series float” result). This means they must be known at compile time and cannot change during the script’s execution. Because the result of `input.*()` function is always of “input” or “series” form, it follows that the result of one `input.*()` function call cannot be used as an argument in a subsequent `input.*()` call because the “input” form is stronger than the “const” form.

Let’s go over each parameter:

- `defval` is the first parameter of all input functions. It is the default value that will appear in the input widget. It requires an argument of the type of input value the function is used for.
- `title` requires a “const string” argument. It is the field’s label.
- `tooltip` requires a “const string” argument. When the parameter is used, a question mark icon will appear to the right of the field. When users hover over it, the tooltip’s text will appear. Note that if multiple input fields are grouped on one line using `inline`, the tooltip will always appear to the right of the rightmost field, and display the text of the last `tooltip` argument used in the line. Newlines ( `\n` ) are supported in the argument string.
- `inline` requires a “const string” argument. Using the same argument for the parameter in multiple `input.*()` calls will group their input widgets on the same line. There is a limit to the width the “Inputs” tab will expand, so a limited quantity of input fields can be fitted on one line. Using one `input.*()` call with a unique argument for `inline` has the effect of bringing the input field left, immediately after the label, foregoing the default left-alignment of all input fields used when no `inline` argument is used.
- `group` requires a “const string” argument. It used to group any number of inputs in the same section. The string used as the `group` argument becomes the section’s heading. All `input.*()` calls to be grouped together must use the same string for their `group` argument.
- `options` requires a comma-separated list of elements enclosed in square brackets (e.g., `["ON", "OFF"]`. It is used to create a dropdown menu offering the list’s elements in the form of menu selections. Only one menu item can be selected. When an `options` list is used, the `defval` value must be one of the list’s elements. When `options` is used in input functions allowing `minval`, `maxval` or `step`, those parameters cannot be used simultaneously.
- `minval` requires a “const int/float” argument, depending on the type of the `defval` value. It is the minimum valid value for the input field.
- `maxval` requires a “const int/float” argument, depending on the type of the `defval` value. It is the maximum valid value for the input field.

- `step` is the increment by which the field's value will move when the widget's up/down arrows are used.
- `confirm` requires a "const bool" ( `true` or `false` ) argument. This parameter affect the behavior of the script when it is added to a chart. `input.*()` calls using `confirm = true` will cause the "Settings/Inputs" tab to popup when the script is added to the chart. `confirm` is useful to ensure that users configure a particular field.

The `minval` , `maxval` and `step` parameters are only present in the signature of the input.int() and input.float() functions.

## Input types

The next sections explain what each input function does. As we proceed, we will explore the different ways you can use input functions and organize their display.

## Simple input

input() is a simple, generic function that supports the fundamental Pine Script™ types: "int", "float", "bool", "color" and "string". It also supports "source" inputs, which are price-related values such as close, hl2, hlc3, and hlcc4, or which can be used to receive the output value of another script.

Its signature is:

```
input(defval, title, tooltip, inline, group) → input int/float/bool/color/string | seri
```

The function automatically detects the type of input by analyzing the type of the `defval` argument used in the function call. This script shows all the supported types and the form-type returned by the function when used with `defval` arguments of different types:

```
//@version=5
indicator("`input()`", "", true)
a = input(1, "input int")
b = input(1.0, "input float")
c = input(true, "input bool")
d = input(color.orange, "input color")
e = input("1", "input string")
f = input(close, "series float")
plot(na)
```

## Integer input

Two signatures exist for the input.int() function; one when `options` is not used, the other when it is:

```
input.int(defval, title, minval, maxval, step, tooltip, inline, group, confirm) → input
input.int(defval, title, options, tooltip, inline, group, confirm) → input int
```

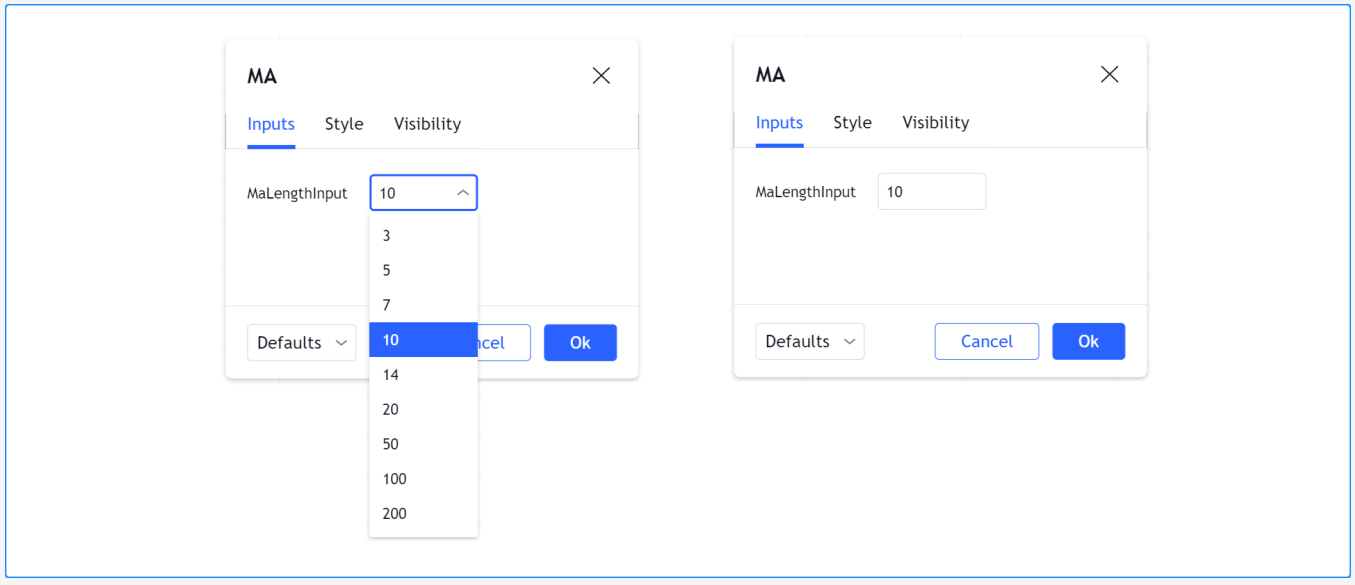
This call uses the `options` parameter to propose a pre-defined list of lengths for the MA:

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, options = [3, 5, 7, 10, 14, 20, 50, 100, 200])
ma = ta.sma(close, maLengthInput)
plot(ma)
```

This one uses the `minval` parameter to limit the length:

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, minval = 2)
ma = ta.sma(close, maLengthInput)
plot(ma)
```

The version with the `options` list uses a dropdown menu for its widget. When the `options` parameter is not used, a simple input widget is used to enter the value.




## Float input

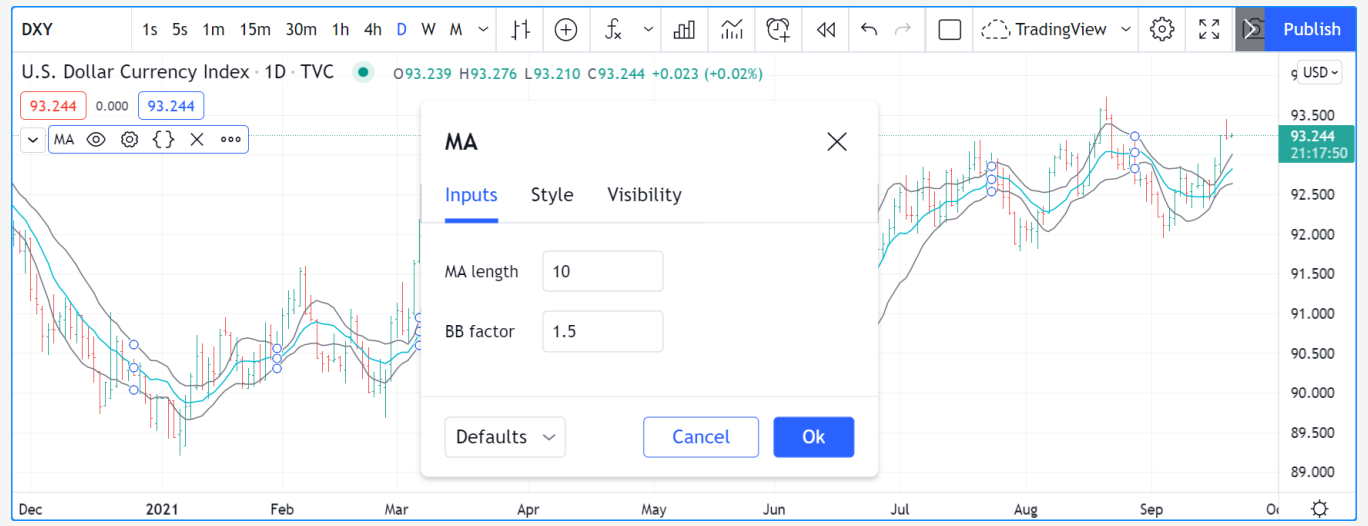
Two signatures exist for the input.float() function; one when `options` is not used, the other when it is:

```
input.int(defval, title, minval, maxval, step, tooltip, inline, group, confirm) → input
input.int(defval, title, options, tooltip, inline, group, confirm) → input int
```

Here, we use a “float” input for the factor used to multiple the standard deviation, to calculate Bollinger Bands:

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, minval = 1)
bbFactorInput = input.float(1.5, minval = 0, step = 0.5)
ma       = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi     = ma + bbWidth
bbLo     = ma - bbWidth
plot(ma)
plot(bbHi, "BB Hi", color.gray)
plot(bbLo, "BB Lo", color.gray)
```

The input widgets for floats are similar to the ones used for integer inputs.




# Boolean input

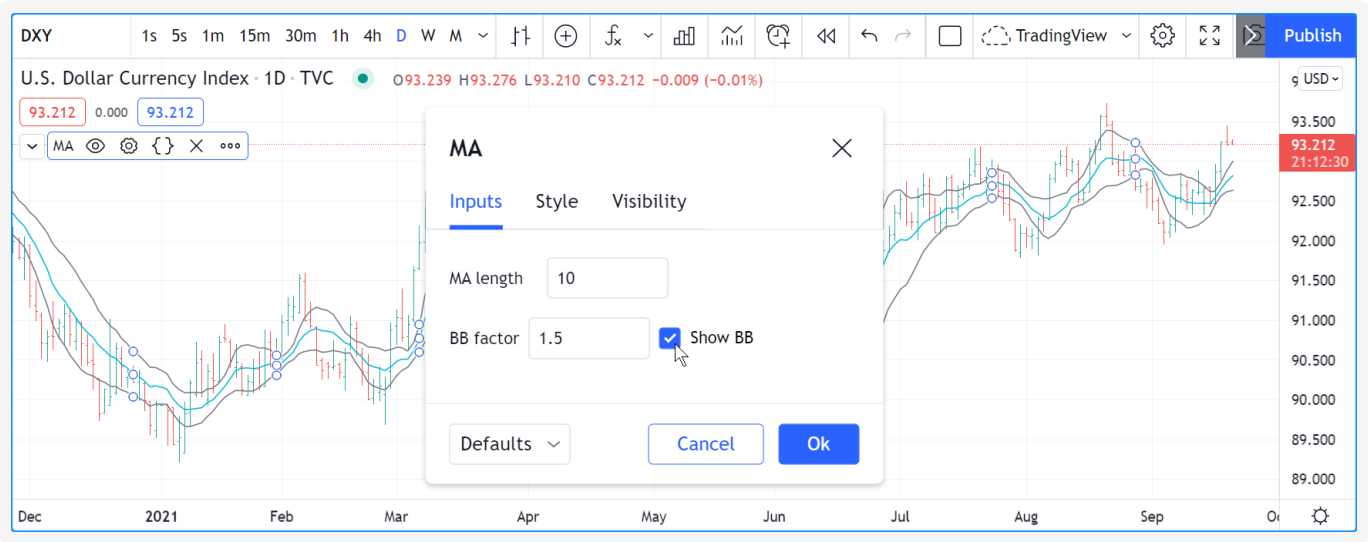
Let’s continue to develop our script further, this time by adding a boolean input to allow users to toggle the display of the BBs:

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10,     "MA length", minval = 1)
bbFactorInput = input.float(1.5, "BB factor", inline = "01", minval = 0, step = 0.5)
showBBInput   = input.bool(true, "Show BB",   inline = "01")
ma       = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi     = ma + bbWidth
bbLo     = ma - bbWidth
plot(ma, "MA", color.aqua)
plot(showBBInput ? bbHi : na, "BB Hi", color.gray)
plot(showBBInput ? bbLo : na, "BB Lo", color.gray)
```

Note that:

- We have added an input using input.bool() to set the value of `showBBInput`.
- We use the `inline` parameter in that input and in the one for `bbFactorInput` to bring them on the same line. We use `"01"` for its argument in both cases. That is how the Pine Script™ compiler recognizes that they belong on the same line. The particular string used as an argument is unimportant and does not appear anywhere in the “Inputs” tab; it is only used to identify which inputs go on the same line.
- We have vertically aligned the `title` arguments of our `input.*()` calls to make them easier to read.

- We use the `showBBInput` variable in our two plot() calls to plot conditionally. When the user unchecks the checkbox of the `showBBInput` input, the variable's value becomes `false`. When that happens, our plot() calls plot the na value, which displays nothing. We use `true` as the default value of the input, so the BBs plot by default.
- Because we use the `inline` parameter for the `bbFactorInput` variable, its input field in the "Inputs" tab does not align vertically with that of `maLengthInput`, which doesn't use `inline`.




## Color input

As is explained in the Color selection through script settings section of the "Colors" page, the color selections that usually appear in the "Settings/Style" tab are not always available. When that is the case, script users will have no means to change the colors your script uses. For those cases, it is essential to provide color inputs if you want your script's colors to be modifiable through the script's "Settings". Instead of using the "Settings/Style" tab to change colors, you will then allow your script users to change the colors using calls to input.color().

Suppose we wanted to plot our BBs in a ligther shade when the high and low values are higher/lower than the BBs. You could use code like this to create your colors:

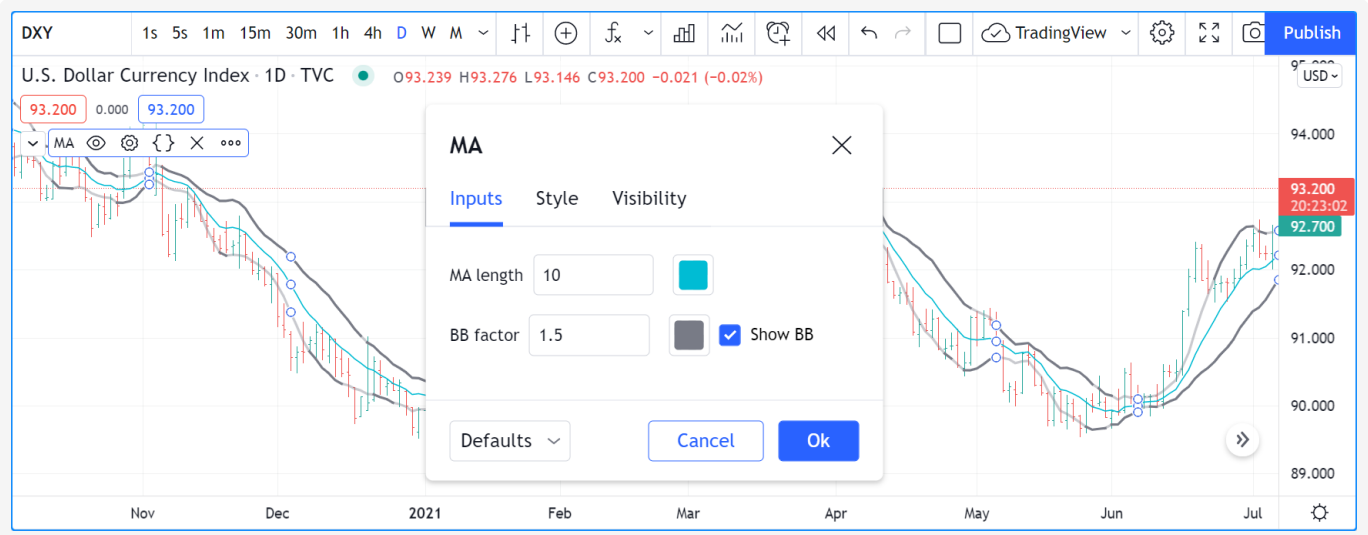
```
bbHiColor = color.new(color.gray, high > bbHi ? 60 : 0)
bbLoColor = color.new(color.gray, low  < bbLo ? 60 : 0)
```

When using dynamic (or "series") color components like the transparency here, the color widgets in the "Settings/Style" will no longer appear. Let's create our own, which will appear in our "Inputs" tab:

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10,           "MA length", inline = "01", minval = 1)
maColorInput  = input.color(color.aqua, "",          inline = "01")
bbFactorInput = input.float(1.5,        "BB factor", inline = "02", minval = 0, step =
bbColorInput  = input.color(color.gray, "",          inline = "02")
showBBInput   = input.bool(true,        "Show BB",   inline = "02")
ma       = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi     = ma + bbWidth
bbLo     = ma - bbWidth
bbHiColor = color.new(bbColorInput, high > bbHi ? 60 : 0)
bbLoColor = color.new(bbColorInput, low  < bbLo ? 60 : 0)
plot(ma, "MA", maColorInput)
plot(showBBInput ? bbHi : na, "BB Hi", bbHiColor, 2)
plot(showBBInput ? bbLo : na, "BB Lo", bbLoColor, 2)
```

Note that:

- We have added two calls to input.color() to gather the values of the `maColorInput` and `bbColorInput` variables. We use `maColorInput` directly in the `plot(ma, "MA", maColorInput)` call, and we use `bbColorInput` to build the `bbHiColor` and `bbLoColor` variables, which modulate the transparency using the position of price relative to the BBs. We use a conditional value for the `transp` value we call color.new() with, to generate different transparencies of the same base color.
- We do not use a `title` argument for our new color inputs because they are on the same line as other inputs allowing users to understand to which plots they apply.
- We have reorganized our `inline` arguments so they reflect the fact we have inputs grouped on two distinct lines.




# Timeframe input

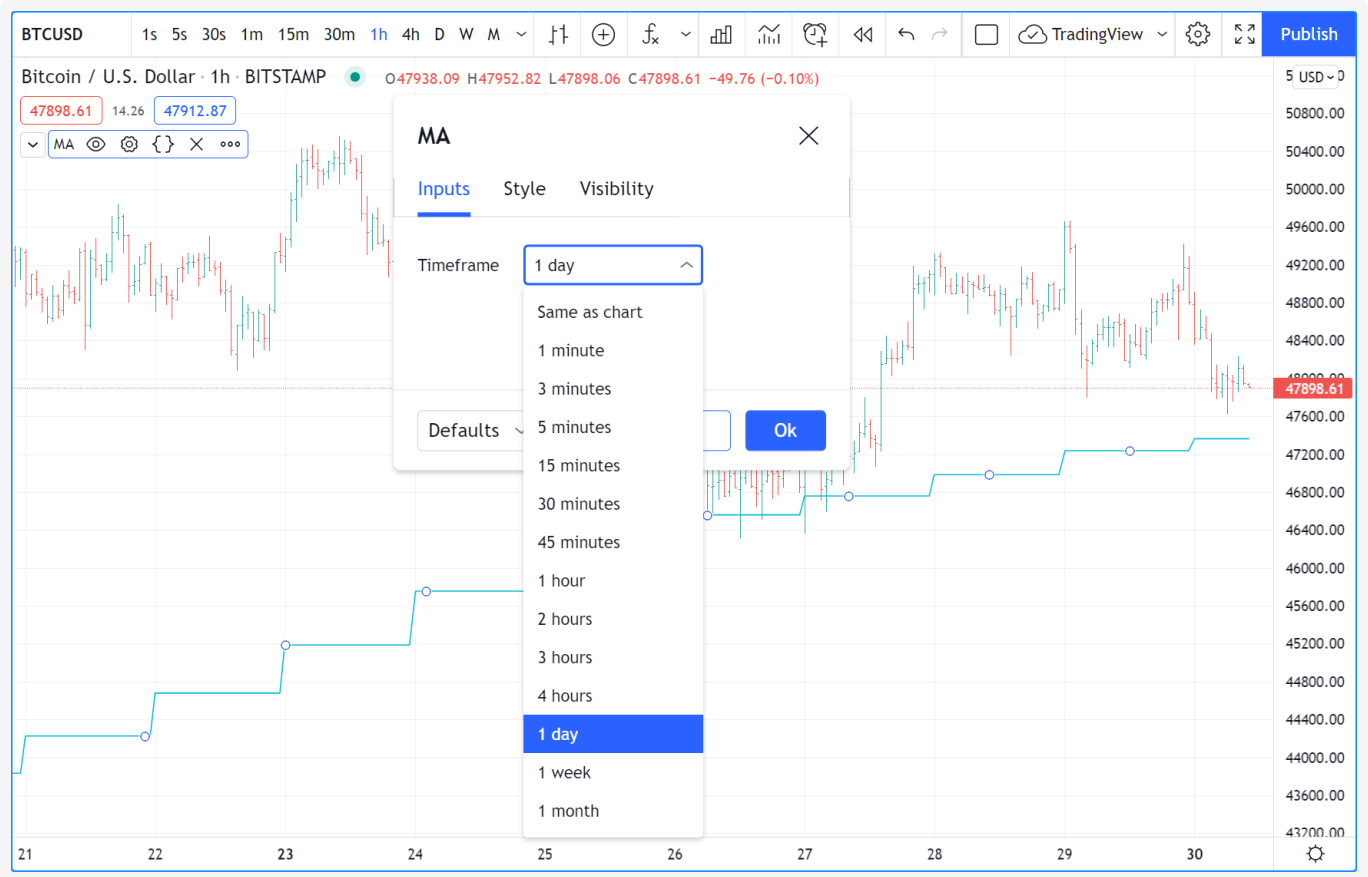
Timeframe inputs can be useful when you want to be able to change the timeframe used to calculate values in your scripts.

Let's do away with our BBs from the previous sections and add a timeframe input to a simple MA script:

```
//@version=5
indicator("MA", "", true)
tfInput = input.timeframe("D", "Timeframe")
ma = ta.sma(close, 20)
securityNoRepaint(sym, tf, src) =>
    request.security(sym, tf, src[barstate.isrealtime ? 1 : 0])[barstate.isrealtime ? (
maHTF = securityNoRepaint(syminfo.tickerid, tfInput, ma)
plot(maHTF, "MA", color.aqua)
```

Note that:

- We use the input.timeframe() function to receive the timeframe input.
- The function creates a dropdown widget where some standard timeframes are proposed. The list of timeframes also includes any you have favorated in the chart user interface.
- We use the `tfInput` in our request.security() call. We also use `gaps = barmerge.gaps_on` in the call, so the function only returns data when the higher timeframe has completed.




## Symbol input

The input.symbol() function creates a widget that allows users to search and select symbols like they would from the chart's user interface.

Let's add a symbol input to our script:

```
//@version=5
indicator("MA", "", true)
tfInput = input.timeframe("D", "Timeframe")
symbolInput = input.symbol("", "Symbol")
ma = ta.sma(close, 20)
securityNoRepaint(sym, tf, src) =>
    request.security(sym, tf, src[barstate.isrealtime ? 1 : 0])[barstate.isrealtime ? (
maHTF = securityNoRepaint(symbolInput, tfInput, ma)
plot(maHTF, "MA", color.aqua)
```

Note that:

- The `defval` argument we use is an empty string. This causes request.security(), where we use the `symbolInput` variable containing that input, to use the chart's symbol by default. If the user selects another symbol and wants to return to the default value using the chart's symbol, he will need to use the "Reset Settings" selection from the "Inputs" tab's "Defaults" menu.
- We use the `securityNoRepaint()` user-defined function to use request.security() in such a way that it does not repaint; it only returns values when the higher timeframe has completed.

## Session input

Session inputs are useful to gather start-stop values for periods of time. The input.session() built-in function creates an input widget allowing users to specify the beginning and end time of a session. Selections can be made using a dropdown menu, or by entering time values in "hh:mm" format.

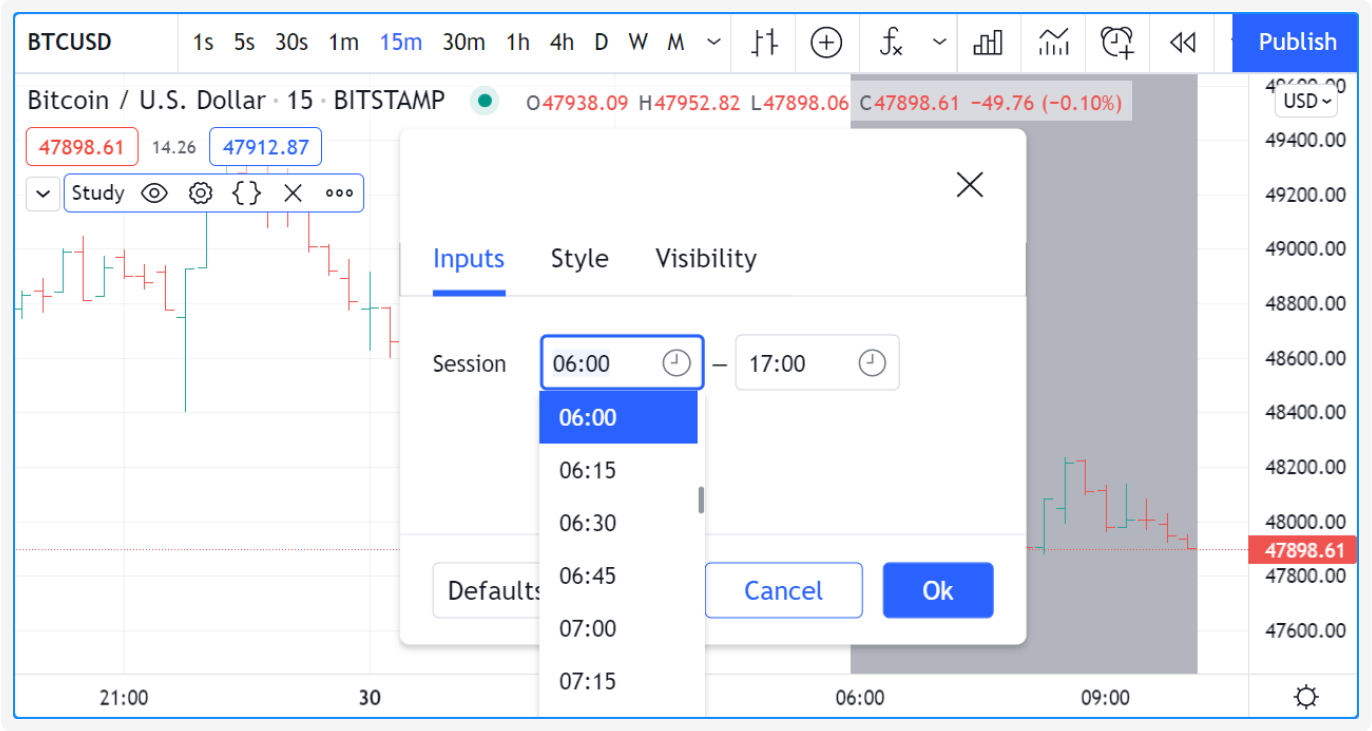
The value returned by input.session() is a valid string in session format. See the manual's page on sessions for more information.

Session information can also contain information on the days where the session is valid. We use an input.string() function call here to input that day information:

```
//@version=5
indicator("Session input", "", true)
string sessionInput = input.session("0600-1700", "Session")
string daysInput = input.string("1234567", tooltip = "1 = Sunday, 7 = Saturday")
sessionString = sessionInput + ":" + daysInput
inSession = not na(time(timeframe.period, sessionString))
bgcolor(inSession ? color.silver : na)
```

Note that:

- This script proposes a default session of "0600-1700".
- The input.string() call uses a tooltip to provide users with help on the format to use to enter day information.
- A complete session string is built by concatenating the two strings the script receives as inputs.
- We explicitly declare the type of our two inputs with the string keyword to make it clear those variables will contain a string.
- We detect if the chart bar is in the user-defined session by calling time() with the session string. If the current bar's time value (the time at the bar's open) is not in the session, time() returns na, so `inSession` will be `true` whenever time() returns a value that is not na.




## Source input

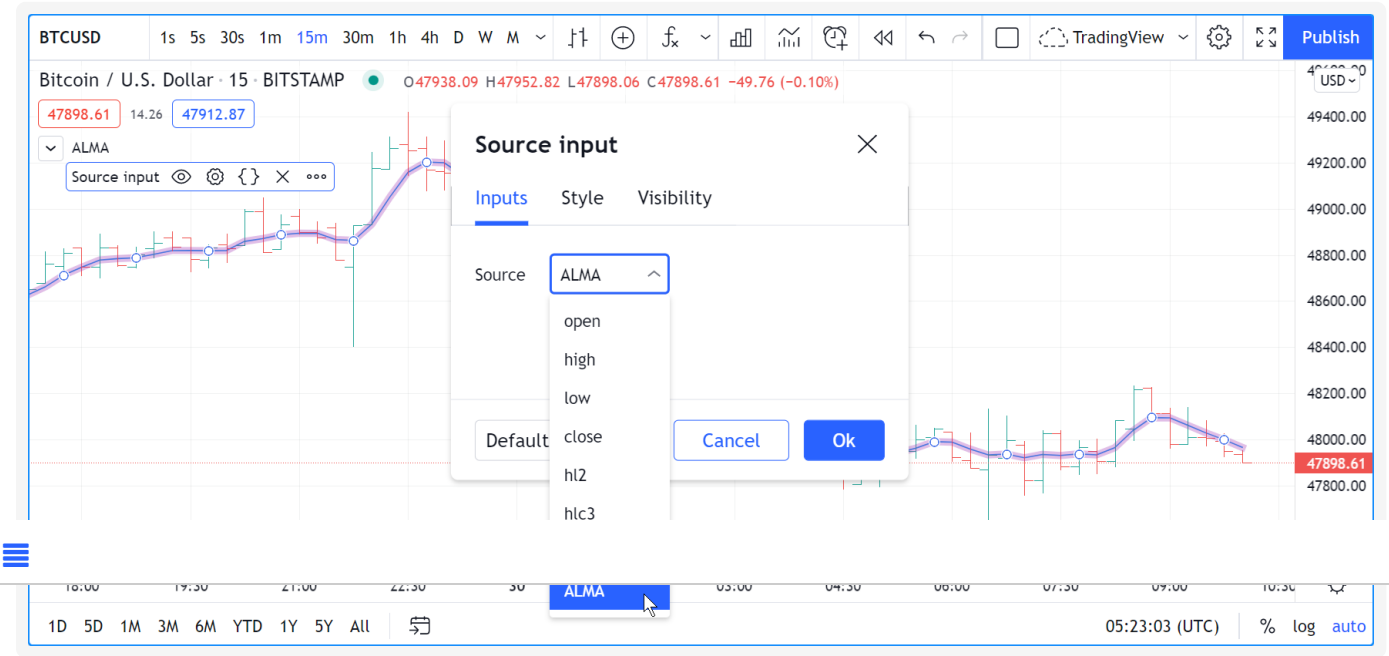
Source inputs are useful to provide a selection of two types of sources:

- Price values, namely: open, high, low, close, hl2, hlc3, and ohlc4.
- The values plotted by other scripts on the chart. This can be useful to “link” two scripts together by sending the output of one as an input to another script.

This script simply plots the user’s selection of source. We propose the high as the default value:

```
//@version=5
indicator("Source input", "", true)
srcInput = input.source(high, "Source")
plot(srcInput, "Src", color.new(color.purple, 70), 6)
```

This shows a chart where, in addition to our script, we have loaded an “Arnaud Legoux Moving Average” indicator. See here how we use our script’s source input widget to select the output of the ALMA script as an input into our script. Because our script plots that source in a light-purple thick line, you see the plots from the two scripts overlap because they plot the same value:




# Time input

Time inputs use the input.time() function. The function returns a Unix time in milliseconds (see the Time page for more information). This type of data also contains date information, so the input.time() function returns a time **and** a date. That is the reason why its widget allows for the selection of both.

Here, we test the bar's time against an input value, and we plot an arrow when it is greater:

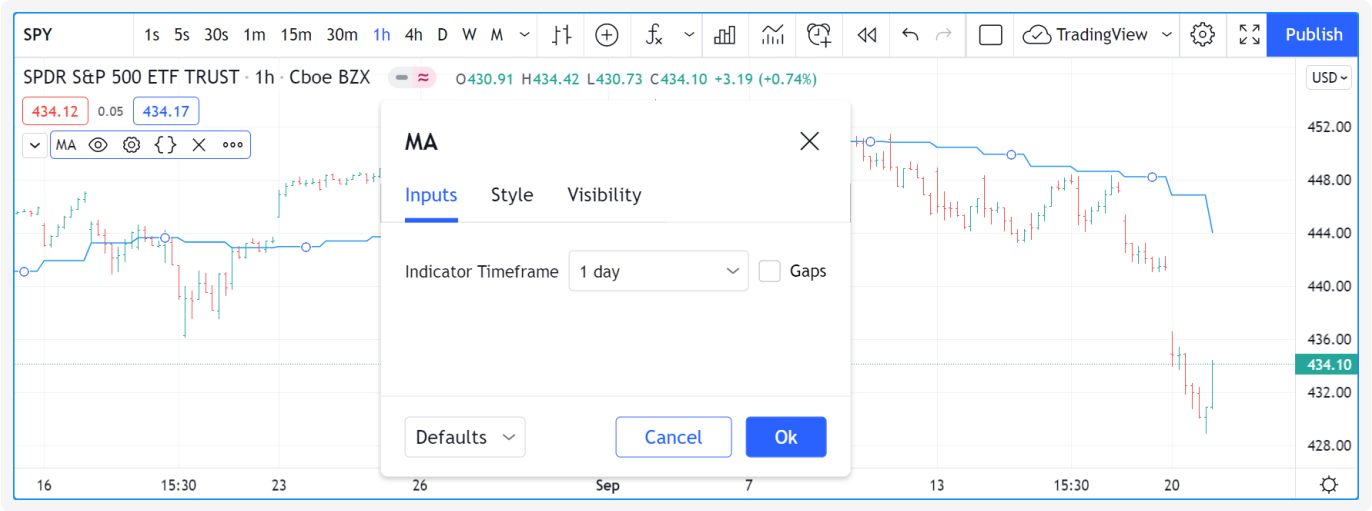
```
//@version=5
indicator("Time input", "T", true)
timeAndDateInput = input.time(timestamp("1 Aug 2021 00:00 +0300"), "Date and time")
barIsLater = time > timeAndDateInput
plotchar(barIsLater, "barIsLater", "🅰️", location.top, size = size.tiny)
```

Note that the `defval` value we use is a call to the timestamp() function.

# Other features affecting Inputs

Some parameters of the indicator() function, when used, will populate the script's "Inputs" tab with a field. The parameters are `timeframe` and `timeframe_gaps`. An example:

```
//@version=5
indicator("MA", "", true, timeframe = "D", timeframe_gaps = false)
plot(ta.vwma(close, 10))
```




# Tips

The design of your script's inputs has an important impact on the usability of your scripts. Well-designed inputs are more intuitively usable and make for a better user experience:

- Choose clear and concise labels (your input's `title` argument).
- Choose your default values carefully.
- Provide `minval` and `maxval` values that will prevent your code from producing unexpected results, e.g., limit the minimal value of lengths to 1 or 2, depending on the type of MA you are using.
- Provide a `step` value that is congruent with the value you are capturing. Steps of 5 can be more useful on a 0-200 range, for example, or steps of 0.05 on a 0.0-1.0 scale.
- Group related inputs on the same line using `inline`; bull and bear colors for example, or the width and color of a line.
- When you have many inputs, group them into meaningful sections using `group`. Place the most important sections at the top.
- Do the same for individual inputs **within** sections.

It can be advantageous to vertically align different arguments of multliple `input.*()` calls in your code. When you need to make global changes, this will allow you to use the Editor's multi-cursor feature to operate on all the lines at once.

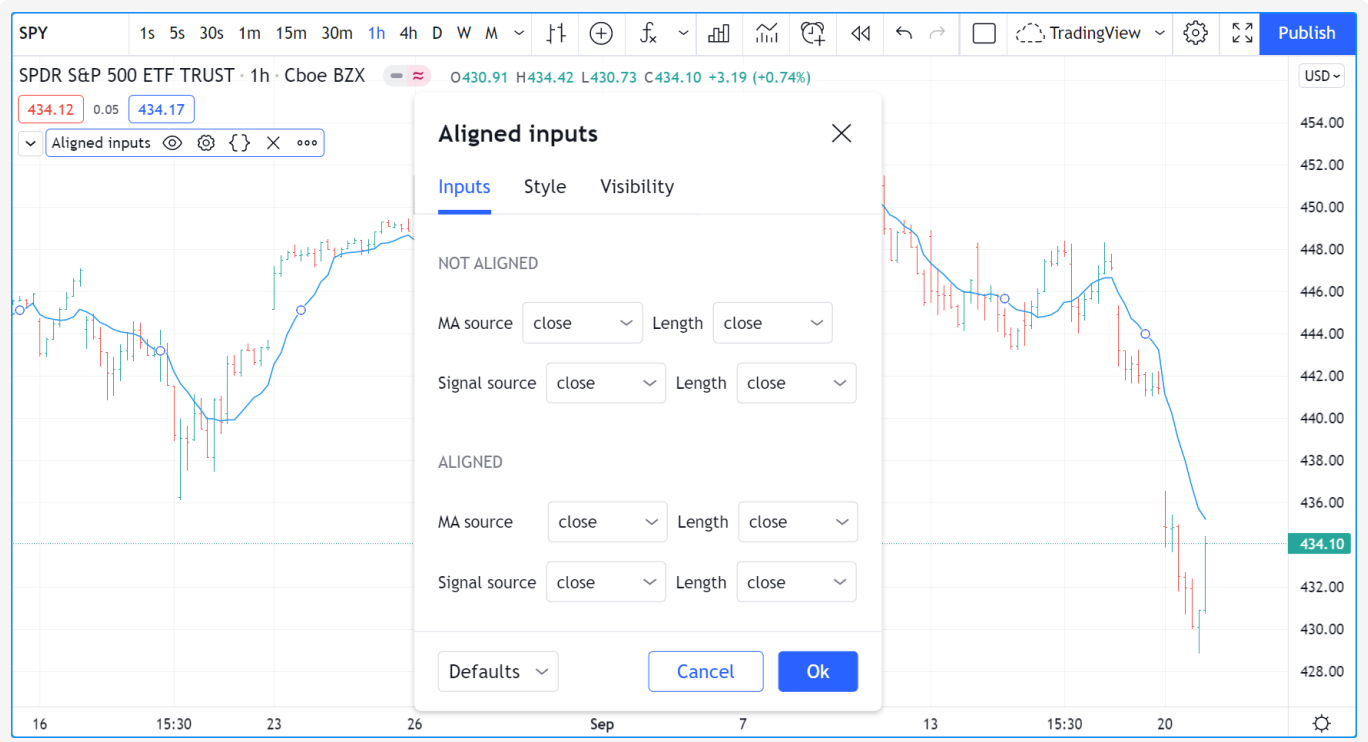
Because It is sometimes necessary to use Unicode spaces to In order to achieve optimal alignment in inputs. This is an example:

```
//@version=5
indicator("Aligned inputs", "", true)

var GRP1 = "Not aligned"
ma1SourceInput   = input(close, "MA source",      inline = "11", group = GRP1)
ma1LengthInput   = input(close, "Length",         inline = "11", group = GRP1)
long1SourceInput = input(close, "Signal source", inline = "12", group = GRP1)
long1LengthInput = input(close, "Length",         inline = "12", group = GRP1)

var GRP2 = "Aligned"
// The three spaces after "MA source" are Unicode EN spaces (U+2002).
ma2SourceInput   = input(close, "MA source   ",  inline = "21", group = GRP2)
ma2LengthInput   = input(close, "Length",         inline = "21", group = GRP2)
long2SourceInput = input(close, "Signal source", inline = "22", group = GRP2)
long2LengthInput = input(close, "Length",         inline = "22", group = GRP2)

plot(ta.vwma(close, 10))
```




Note that:

- We use the `group` parameter to distinguish between the two sections of inputs. We use a constant to hold the name of the groups. This way, if we decide to change the name of the group, we only need to change it in one place.
- The first sections inputs widgets do not align vertically. We are using `inline`, which places the input widgets immediately to the right of the label. Because the labels for the `ma1SourceInput` and `long1SourceInput` inputs are of different lengths the labels are in different y positions.
- To make up for the misalignment, we pad the `title` argument in the `ma2SourceInput` line with three Unicode EN spaces (U+2002). Unicode spaces are necessary because ordinary spaces would be stripped from the label. You can achieve precise alignment by combining different quantities and types of Unicode spaces. See here for a list of Unicode spaces of different widths.

Fills

Levels