

---

# Pine Script™ v5 User Manual

TradingView

May 14, 2024



# CONTENTS

<b>1</b>	<b>Welcome to Pine Script™ v5</b>	<b>1</b>
<b>2</b>	<b>Pine Script™ primer</b>	<b>3</b>
2.1	First steps . . . . .	3
2.1.1	Introduction . . . . .	3
2.1.2	Using scripts . . . . .	4
2.1.3	Reading scripts . . . . .	7
2.1.4	Writing scripts . . . . .	8
2.2	First indicator . . . . .	8
2.2.1	The Pine Editor . . . . .	8
2.2.2	First version . . . . .	9
2.2.3	Second version . . . . .	10
2.2.4	Next . . . . .	12
2.3	Next steps . . . . .	12
2.3.1	“indicators” vs “strategies” . . . . .	12
2.3.2	How scripts are executed . . . . .	13
2.3.3	Time series . . . . .	13
2.3.4	Publishing scripts . . . . .	13
2.3.5	Getting around the Pine Script™ documentation . . . . .	14
2.3.6	Where to go from here? . . . . .	14
<b>3</b>	<b>Language</b>	<b>15</b>
3.1	Execution model . . . . .	15
3.1.1	Calculation based on historical bars . . . . .	16
3.1.2	Calculation based on realtime bars . . . . .	17
3.1.3	Events triggering the execution of a script . . . . .	18
3.1.4	More information . . . . .	18
3.1.5	Historical values of functions . . . . .	19
3.2	Time series . . . . .	22
3.3	Script structure . . . . .	23
3.3.1	Version . . . . .	24
3.3.2	Declaration statement . . . . .	24
3.3.3	Code . . . . .	24
3.3.4	Comments . . . . .	26
3.3.5	Line wrapping . . . . .	26
3.3.6	Compiler annotations . . . . .	27
3.4	Identifiers . . . . .	29
3.5	Operators . . . . .	30
3.5.1	Introduction . . . . .	30
3.5.2	Arithmetic operators . . . . .	31

3.5.3	Comparison operators . . . . .	31
3.5.4	Logical operators . . . . .	32
3.5.5	`?:` ternary operator . . . . .	32
3.5.6	`[ ]` history-referencing operator . . . . .	33
3.5.7	Operator precedence . . . . .	34
3.5.8	`:=` assignment operator . . . . .	34
3.5.9	`:=` reassignment operator . . . . .	34
3.6	Variable declarations . . . . .	36
3.6.1	Introduction . . . . .	36
3.6.2	Variable reassignment . . . . .	37
3.6.3	Declaration modes . . . . .	38
3.7	Conditional structures . . . . .	41
3.7.1	Introduction . . . . .	41
3.7.2	`if` structure . . . . .	42
3.7.3	`switch` structure . . . . .	44
3.7.4	Matching local block type requirement . . . . .	46
3.8	Loops . . . . .	47
3.8.1	Introduction . . . . .	47
3.8.2	`for` . . . . .	48
3.8.3	`while` . . . . .	51
3.9	Type system . . . . .	53
3.9.1	Introduction . . . . .	53
3.9.2	Qualifiers . . . . .	53
3.9.3	Types . . . . .	57
3.9.4	`na` value . . . . .	63
3.9.5	Type templates . . . . .	65
3.9.6	Type casting . . . . .	66
3.9.7	Tuples . . . . .	67
3.10	Built-ins . . . . .	69
3.10.1	Introduction . . . . .	70
3.10.2	Built-in variables . . . . .	70
3.10.3	Built-in functions . . . . .	71
3.11	User-defined functions . . . . .	73
3.11.1	Introduction . . . . .	74
3.11.2	Single-line functions . . . . .	74
3.11.3	Multi-line functions . . . . .	75
3.11.4	Scopes in the script . . . . .	75
3.11.5	Functions that return multiple results . . . . .	76
3.11.6	Limitations . . . . .	76
3.12	Objects . . . . .	76
3.12.1	Introduction . . . . .	77
3.12.2	Creating objects . . . . .	77
3.12.3	Changing field values . . . . .	79
3.12.4	Collecting objects . . . . .	79
3.12.5	Copying objects . . . . .	81
3.12.6	Shadowing . . . . .	83
3.13	Methods . . . . .	83
3.13.1	Introduction . . . . .	83
3.13.2	Built-in methods . . . . .	83
3.13.3	User-defined methods . . . . .	86
3.13.4	Method overloading . . . . .	89
3.13.5	Advanced example . . . . .	90
3.14	Arrays . . . . .	94
3.14.1	Introduction . . . . .	95

3.14.2	Declaring arrays . . . . .	95
3.14.3	Reading and writing array elements . . . . .	97
3.14.4	Looping through array elements . . . . .	99
3.14.5	Scope . . . . .	100
3.14.6	History referencing . . . . .	101
3.14.7	Inserting and removing array elements . . . . .	101
3.14.8	Calculations on arrays . . . . .	105
3.14.9	Manipulating arrays . . . . .	105
3.14.10	Searching arrays . . . . .	109
3.14.11	Error handling . . . . .	109
3.15	Matrices . . . . .	112
3.15.1	Introduction . . . . .	112
3.15.2	Declaring a matrix . . . . .	112
3.15.3	Reading and writing matrix elements . . . . .	114
3.15.4	Rows and columns . . . . .	116
3.15.5	Looping through a matrix . . . . .	122
3.15.6	Copying a matrix . . . . .	125
3.15.7	Scope and history . . . . .	130
3.15.8	Inspecting a matrix . . . . .	132
3.15.9	Manipulating a matrix . . . . .	134
3.15.10	Matrix calculations . . . . .	141
3.15.11	Error handling . . . . .	151
3.16	Maps . . . . .	154
3.16.1	Introduction . . . . .	155
3.16.2	Declaring a map . . . . .	155
3.16.3	Reading and writing . . . . .	157
3.16.4	Looping through a map . . . . .	165
3.16.5	Copying a map . . . . .	168
3.16.6	Scope and history . . . . .	170
3.16.7	Maps of other collections . . . . .	172
<b>4</b>	<b>Concepts</b>	<b>175</b>
4.1	Alerts . . . . .	175
4.1.1	Introduction . . . . .	175
4.1.2	Script alerts . . . . .	177
4.1.3	`alertcondition()` events . . . . .	181
4.1.4	Avoiding repainting with alerts . . . . .	184
4.2	Backgrounds . . . . .	185
4.3	Bar coloring . . . . .	188
4.4	Bar plotting . . . . .	189
4.4.1	Introduction . . . . .	189
4.4.2	Plotting candles with `plotcandle()` . . . . .	189
4.4.3	Plotting bars with `plotbar()` . . . . .	192
4.5	Bar states . . . . .	192
4.5.1	Introduction . . . . .	193
4.5.2	Bar state built-in variables . . . . .	193
4.5.3	Example . . . . .	195
4.6	Chart information . . . . .	197
4.6.1	Introduction . . . . .	197
4.6.2	Prices and volume . . . . .	198
4.6.3	Symbol information . . . . .	198
4.6.4	Chart timeframe . . . . .	200
4.6.5	Session information . . . . .	200
4.7	Colors . . . . .	201

4.7.1	Introduction . . . . .	201
4.7.2	Constant colors . . . . .	202
4.7.3	Conditional coloring . . . . .	203
4.7.4	Calculated colors . . . . .	205
4.7.5	Mixing transparencies . . . . .	209
4.7.6	Tips . . . . .	211
4.8	Fills . . . . .	214
4.8.1	Introduction . . . . .	215
4.8.2	`plot()` and `hline()` fills . . . . .	215
4.8.3	Line fills . . . . .	218
4.8.4	Box and polyline fills . . . . .	219
4.9	Inputs . . . . .	221
4.9.1	Introduction . . . . .	221
4.9.2	Input functions . . . . .	222
4.9.3	Input function parameters . . . . .	223
4.9.4	Input types . . . . .	224
4.9.5	Other features affecting Inputs . . . . .	232
4.9.6	Tips . . . . .	232
4.10	Levels . . . . .	234
4.10.1	`hline()` levels . . . . .	234
4.10.2	Fills between levels . . . . .	235
4.11	Libraries . . . . .	236
4.11.1	Introduction . . . . .	237
4.11.2	Creating a library . . . . .	237
4.11.3	Publishing a library . . . . .	241
4.11.4	Using a library . . . . .	243
4.12	Lines and boxes . . . . .	244
4.12.1	Introduction . . . . .	244
4.12.2	Lines . . . . .	245
4.12.3	Boxes . . . . .	257
4.12.4	Polylines . . . . .	268
4.12.5	Realtime behavior . . . . .	279
4.12.6	Limitations . . . . .	280
4.13	Non-standard charts data . . . . .	282
4.13.1	Introduction . . . . .	282
4.13.2	`ticker.heikinashi()` . . . . .	282
4.13.3	`ticker.renko()` . . . . .	285
4.13.4	`ticker.linebreak()` . . . . .	285
4.13.5	`ticker.kagi()` . . . . .	285
4.13.6	`ticker.pointfigure()` . . . . .	285
4.14	Other timeframes and data . . . . .	286
4.14.1	Introduction . . . . .	286
4.14.2	Common characteristics . . . . .	288
4.14.3	Data feeds . . . . .	294
4.14.4	`request.security()` . . . . .	295
4.14.5	`request.security_lower_tf()` . . . . .	310
4.14.6	Custom contexts . . . . .	317
4.14.7	Historical and realtime behavior . . . . .	321
4.14.8	`request.currency_rate()` . . . . .	326
4.14.9	`request.dividends()`, `request.splits()`, and `request.earnings()` . . . . .	327
4.14.10	`request.quandl()` . . . . .	330
4.14.11	`request.financial()` . . . . .	331
4.14.12	`request.economic()` . . . . .	340
4.14.13	`request.seed()` . . . . .	352

4.15	Plots . . . . .	354
4.15.1	Introduction . . . . .	354
4.15.2	`plot()` parameters . . . . .	356
4.15.3	Plotting conditionally . . . . .	359
4.15.4	Levels . . . . .	362
4.15.5	Offsets . . . . .	363
4.15.6	Plot count limit . . . . .	363
4.15.7	Scale . . . . .	364
4.16	Repainting . . . . .	366
4.16.1	Introduction . . . . .	366
4.16.2	Historical vs realtime calculations . . . . .	368
4.16.3	Plotting in the past . . . . .	373
4.16.4	Dataset variations . . . . .	374
4.17	Sessions . . . . .	375
4.17.1	Introduction . . . . .	375
4.17.2	Session strings . . . . .	376
4.17.3	Session states . . . . .	378
4.17.4	Using sessions with `request.security()` . . . . .	378
4.18	Strategies . . . . .	380
4.18.1	Introduction . . . . .	380
4.18.2	A simple strategy example . . . . .	381
4.18.3	Applying a strategy to a chart . . . . .	381
4.18.4	Strategy tester . . . . .	382
4.18.5	Broker emulator . . . . .	385
4.18.6	Orders and entries . . . . .	388
4.18.7	Position sizing . . . . .	410
4.18.8	Closing a market position . . . . .	412
4.18.9	OCA groups . . . . .	414
4.18.10	Currency . . . . .	418
4.18.11	Altering calculation behavior . . . . .	419
4.18.12	Simulating trading costs . . . . .	423
4.18.13	Risk management . . . . .	429
4.18.14	Margin . . . . .	430
4.18.15	Strategy Alerts . . . . .	431
4.18.16	Notes on testing strategies . . . . .	434
4.19	Tables . . . . .	435
4.19.1	Introduction . . . . .	435
4.19.2	Creating tables . . . . .	436
4.19.3	Tips . . . . .	442
4.20	Text and shapes . . . . .	443
4.20.1	Introduction . . . . .	443
4.20.2	`plotchar()` . . . . .	445
4.20.3	`plotshape()` . . . . .	447
4.20.4	`plotarrow()` . . . . .	449
4.20.5	Labels . . . . .	451
4.21	Time . . . . .	459
4.21.1	Introduction . . . . .	460
4.21.2	Time variables . . . . .	463
4.21.3	Time functions . . . . .	465
4.21.4	Formatting dates and time . . . . .	468
4.22	Timeframes . . . . .	470
4.22.1	Introduction . . . . .	470
4.22.2	Timeframe string specifications . . . . .	471
4.22.3	Comparing timeframes . . . . .	471

<b>5 Writing scripts</b>	<b>473</b>
5.1 Style guide . . . . .	473
5.1.1 Introduction . . . . .	473
5.1.2 Naming Conventions . . . . .	474
5.1.3 Script organization . . . . .	474
5.1.4 Spacing . . . . .	478
5.1.5 Line wrapping . . . . .	478
5.1.6 Vertical alignment . . . . .	478
5.1.7 Explicit typing . . . . .	479
5.2 Debugging . . . . .	479
5.2.1 Introduction . . . . .	479
5.2.2 The lay of the land . . . . .	480
5.2.3 Numeric values . . . . .	483
5.2.4 Conditions . . . . .	491
5.2.5 Strings . . . . .	502
5.2.6 Pine Logs . . . . .	511
5.2.7 Debugging functions . . . . .	520
5.2.8 Debugging loops . . . . .	525
5.2.9 Tips . . . . .	531
5.3 Profiling and optimization . . . . .	533
5.3.1 Introduction . . . . .	533
5.3.2 Pine Profiler . . . . .	533
5.3.3 Optimization . . . . .	559
5.3.4 Tips . . . . .	588
5.4 Publishing scripts . . . . .	590
5.4.1 Script visibility and access . . . . .	591
5.4.2 Preparing a publication . . . . .	593
5.4.3 Publishing a script . . . . .	595
5.4.4 Updating a publication . . . . .	595
5.5 Limitations . . . . .	597
5.5.1 Introduction . . . . .	597
5.5.2 Time . . . . .	597
5.5.3 Chart visuals . . . . .	598
5.5.4 `request.*()` calls . . . . .	602
5.5.5 Script size and memory . . . . .	604
5.5.6 Other limitations . . . . .	606
<b>6 FAQ</b>	<b>609</b>
6.1 Get real OHLC price on a Heikin Ashi chart . . . . .	609
6.2 Get non-standard OHLC values on a standard chart . . . . .	610
6.3 Plot arrows on the chart . . . . .	610
6.4 Plot a dynamic horizontal line . . . . .	611
6.5 Plot a vertical line on condition . . . . .	611
6.6 Access the previous value . . . . .	612
6.7 Get a 5-days high . . . . .	612
6.8 Count bars in a dataset . . . . .	613
6.9 Enumerate bars in a day . . . . .	613
6.10 Find the highest and lowest values for the entire dataset . . . . .	613
6.11 Query the last non-na value . . . . .	614
<b>7 Error messages</b>	<b>615</b>
7.1 The if statement is too long . . . . .	615
7.2 Script requesting too many securities . . . . .	615
7.3 Script could not be translated from: null . . . . .	616

7.4	line 2: no viable alternative at character '\$' . . . . .	616
7.5	Mismatched input <...> expecting <??> . . . . .	616
7.6	Loop is too long (> 500 ms) . . . . .	617
7.7	Script has too many local variables . . . . .	618
7.8	Pine Script™ cannot determine the referencing length of a series. Try using max_bars_back in the indicator or strategy function . . . . .	618
<b>8</b>	<b>Release notes</b>	<b>621</b>
8.1	2024 . . . . .	621
	8.1.1 May 2024 . . . . .	621
	8.1.2 April 2024 . . . . .	622
	8.1.3 March 2024 . . . . .	622
	8.1.4 February 2024 . . . . .	622
	8.1.5 January 2024 . . . . .	623
8.2	2023 . . . . .	624
	8.2.1 December 2023 . . . . .	624
	8.2.2 November 2023 . . . . .	624
	8.2.3 October 2023 . . . . .	625
	8.2.4 September 2023 . . . . .	625
	8.2.5 August 2023 . . . . .	626
	8.2.6 July 2023 . . . . .	626
	8.2.7 June 2023 . . . . .	626
	8.2.8 May 2023 . . . . .	626
	8.2.9 April 2023 . . . . .	627
	8.2.10 March 2023 . . . . .	627
	8.2.11 February 2023 . . . . .	627
	8.2.12 January 2023 . . . . .	627
8.3	2022 . . . . .	627
	8.3.1 December 2022 . . . . .	627
	8.3.2 November 2022 . . . . .	628
	8.3.3 October 2022 . . . . .	628
	8.3.4 September 2022 . . . . .	628
	8.3.5 August 2022 . . . . .	628
	8.3.6 July 2022 . . . . .	629
	8.3.7 June 2022 . . . . .	629
	8.3.8 May 2022 . . . . .	630
	8.3.9 April 2022 . . . . .	631
	8.3.10 March 2022 . . . . .	633
	8.3.11 February 2022 . . . . .	634
	8.3.12 January 2022 . . . . .	634
8.4	2021 . . . . .	635
	8.4.1 December 2021 . . . . .	635
	8.4.2 November 2021 . . . . .	636
	8.4.3 October 2021 . . . . .	638
	8.4.4 September 2021 . . . . .	639
	8.4.5 July 2021 . . . . .	639
	8.4.6 June 2021 . . . . .	640
	8.4.7 May 2021 . . . . .	640
	8.4.8 April 2021 . . . . .	640
	8.4.9 March 2021 . . . . .	641
	8.4.10 February 2021 . . . . .	642
	8.4.11 January 2021 . . . . .	642
8.5	2020 . . . . .	643
	8.5.1 December 2020 . . . . .	643

8.5.2	November 2020 . . . . .	643
8.5.3	October 2020 . . . . .	643
8.5.4	September 2020 . . . . .	643
8.5.5	August 2020 . . . . .	645
8.5.6	July 2020 . . . . .	645
8.5.7	June 2020 . . . . .	645
8.5.8	May 2020 . . . . .	648
8.5.9	April 2020 . . . . .	648
8.5.10	March 2020 . . . . .	648
8.5.11	February 2020 . . . . .	648
8.5.12	January 2020 . . . . .	649
8.6	2019 . . . . .	649
8.6.1	December 2019 . . . . .	649
8.6.2	October 2019 . . . . .	649
8.6.3	September 2019 . . . . .	650
8.6.4	July-August 2019 . . . . .	650
8.6.5	June 2019 . . . . .	651
8.7	2018 . . . . .	651
8.7.1	October 2018 . . . . .	651
8.7.2	April 2018 . . . . .	651
8.8	2017 . . . . .	651
8.8.1	August 2017 . . . . .	651
8.8.2	June 2017 . . . . .	651
8.8.3	May 2017 . . . . .	652
8.8.4	April 2017 . . . . .	652
8.8.5	March 2017 . . . . .	652
8.8.6	February 2017 . . . . .	652
8.9	2016 . . . . .	653
8.9.1	December 2016 . . . . .	653
8.9.2	October 2016 . . . . .	653
8.9.3	September 2016 . . . . .	653
8.9.4	July 2016 . . . . .	653
8.9.5	March 2016 . . . . .	653
8.9.6	February 2016 . . . . .	653
8.9.7	January 2016 . . . . .	653
8.10	2015 . . . . .	654
8.10.1	October 2015 . . . . .	654
8.10.2	September 2015 . . . . .	654
8.10.3	July 2015 . . . . .	654
8.10.4	June 2015 . . . . .	654
8.10.5	April 2015 . . . . .	654
8.10.6	March 2015 . . . . .	654
8.10.7	February 2015 . . . . .	654
8.11	2014 . . . . .	655
8.11.1	August 2014 . . . . .	655
8.11.2	July 2014 . . . . .	655
8.11.3	June 2014 . . . . .	655
8.11.4	April 2014 . . . . .	655
8.11.5	February 2014 . . . . .	655
8.12	2013 . . . . .	656
<b>9</b>	<b>Migration guides</b>	<b>657</b>
9.1	To Pine Script™ version 5 . . . . .	657
9.1.1	Introduction . . . . .	658

9.1.2	v4 to v5 converter . . . . .	658
9.1.3	Renamed functions and variables . . . . .	659
9.1.4	Renamed function parameters . . . . .	659
9.1.5	Removed an `rsi()` overload . . . . .	660
9.1.6	Reserved keywords . . . . .	660
9.1.7	Removed `iff()` and `offset()` . . . . .	660
9.1.8	Split of `input()` into several functions . . . . .	661
9.1.9	Some function parameters now require built-in arguments . . . . .	661
9.1.10	Deprecated the `transp` parameter . . . . .	662
9.1.11	Changed the default session days for `time()` and `time_close()` . . . . .	663
9.1.12	`strategy.exit()` now must do something . . . . .	663
9.1.13	Common script conversion errors . . . . .	663
9.1.14	All variable, function, and parameter name changes . . . . .	665
9.2	To Pine Script™ version 4 . . . . .	668
9.2.1	Converter . . . . .	669
9.2.2	Renaming of built-in constants, variables, and functions . . . . .	669
9.2.3	Explicit variable type declaration . . . . .	670
9.3	To Pine Script™ version 3 . . . . .	670
9.3.1	Default behaviour of security function has changed . . . . .	670
9.3.2	Self-referenced variables are removed . . . . .	671
9.3.3	Forward-referenced variables are removed . . . . .	672
9.3.4	Resolving a problem with a mutable variable in a security expression . . . . .	672
9.3.5	Math operations with booleans are forbidden . . . . .	672
<b>10</b>	<b>Where can I get more information?</b>	<b>675</b>
10.1	External resources . . . . .	675
10.2	Download this manual . . . . .	675



---

**CHAPTER  
ONE**

---

## **WELCOME TO PINE SCRIPT™ V5**



# **Pine Script™**

Pine Script™ is TradingView's programming language. It allows traders to create their own trading tools and run them on our servers. We designed Pine Script™ as a lightweight, yet powerful, language for developing indicators and strategies that you can then backtest. Most of TradingView's built-in indicators are written in Pine Script™, and our thriving community of Pine Script™ programmers has published more than 100,000 [Community Scripts](#).

It's our explicit goal to keep Pine Script™ accessible and easy to understand for the broadest possible audience. Pine Script™ is cloud-based and therefore different from client-side programming languages. While we likely won't develop Pine Script™ into a full-fledged language, we do constantly improve it and are always happy to consider requests for new features.

Because each script uses computational resources in the cloud, we must impose limits in order to share these resources fairly among our users. We strive to set as few limits as possible, but will of course have to implement as many as needed for the platform to run smoothly. Limitations apply to the amount of data requested from additional symbols, execution time, memory usage and script size.

 **TradingView**



---

CHAPTER  
TWO

---

## PINE SCRIPT™ PRIMER



### 2.1 First steps

- *Introduction*
- *Using scripts*
- *Reading scripts*
- *Writing scripts*

#### 2.1.1 Introduction

Welcome to the Pine Script™ v5 User Manual, which will accompany you in your journey to learn to program your own trading tools in Pine Script™. Welcome also to the very active community of Pine Script™ programmers on TradingView.

In this page, we present a step-by-step approach that you can follow to gradually become more familiar with indicators and strategies (also called *scripts*) written in the Pine Script™ programming language on TradingView. We will get you started on your journey to:

1. **Use** some of the tens of thousands of existing scripts on the platform.
2. **Read** the Pine Script™ code of existing scripts.
3. **Write** Pine Script™ scripts.

If you are already familiar with the use of Pine scripts on TradingView and are now ready to learn how to write your own, then jump to the *Writing scripts* section of this page.

If you are new to our platform, then please read on!

## 2.1.2 Using scripts

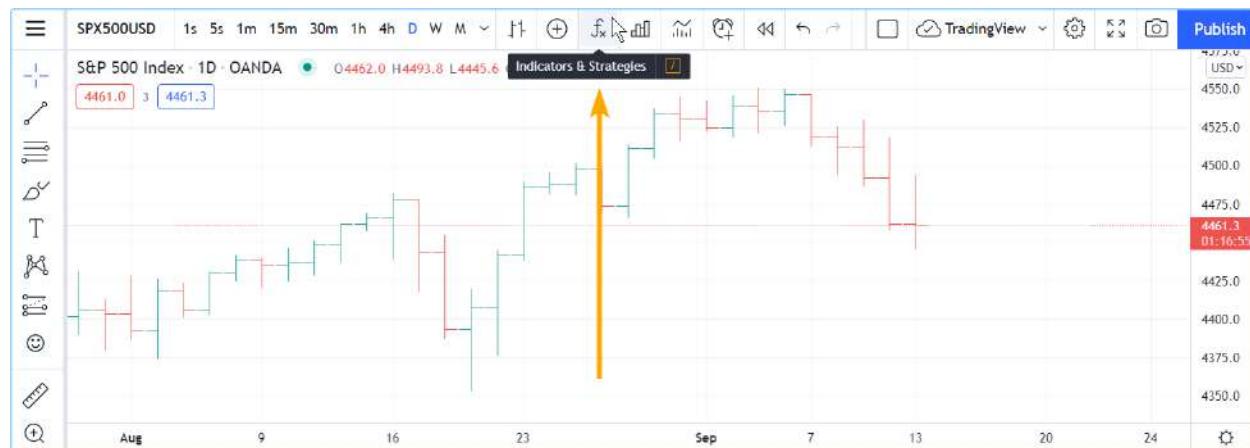
If you are interested in using technical indicators or strategies on TradingView, you can first start exploring the thousands of indicators already available on our platform. You can access existing indicators on the platform in two different ways:

- By using the chart's "Indicators & Strategies" button, or
- By browsing TradingView's [Community Scripts](#), the largest repository of trading scripts in the world, with more than 100,000 scripts, most of which are free and open-source, which means you can see their Pine Script™ code.

If you can find the tools you need already written for you, it can be a good way to get started and gradually become proficient as a script user, until you are ready to start your programming journey in Pine Script™.

### Loading scripts from the chart

To explore and load scripts from your chart, use the "Indicators & Strategies" button:



The dialog box presents different categories of scripts in its left pane:

- **Favorites** lists the scripts you have “favorited” by clicking on the star that appears to the left of its name when you mouse over it.
- **My scripts** displays the scripts you have written and saved in the Pine Editor. They are saved in TradingView’s cloud.
- **Built-ins** groups all TradingView built-ins organized in four categories: indicators, strategies, candlestick patterns and volume profiles. Most are written in Pine Script™ and available for free.
- **Community Scripts** is where you can search from the 100,000+ published scripts written by TradingView users.
- **Invite-only scripts** contains the list of the invite-only scripts you have been granted access to by their authors.

Here, the section containing the TradingView built-ins is selected:

The screenshot shows a sidebar with icons for Favorites, My scripts, Built-ins (which is highlighted in blue), Community Scripts, and Invite-only scripts. Below the sidebar, there are tabs for Indicators, Strategies, Candlestick patterns, and Volume profile. The main area displays a list of built-in scripts: Accumulation/Distribution, Advance Decline Line, Advance Decline Ratio, and Advance/Decline Ratio (Bars). A search bar is at the top left.

When you click on one of the indicators or strategies (the ones with the green and red arrows following their name), it loads on your chart.

## Browsing Community Scripts

From [TradingView's homepage](#), you can bring up the Community Scripts stream from the “Community” menu. Here, we are pointing to the “Editors’ Picks” section, but there are many other categories you can choose from:

The screenshot shows the TradingView homepage with the “SCRIPTS” tab selected. The “Community” menu is open, showing categories like Ideas, Scripts, Streams, and More, with “Editors’ picks” highlighted. On the left, there’s a “Monthly Returns in PineScript Strategies” chart and a “Top authors: Scripts” list. A yellow box highlights the “Editors’ picks” section of the menu.

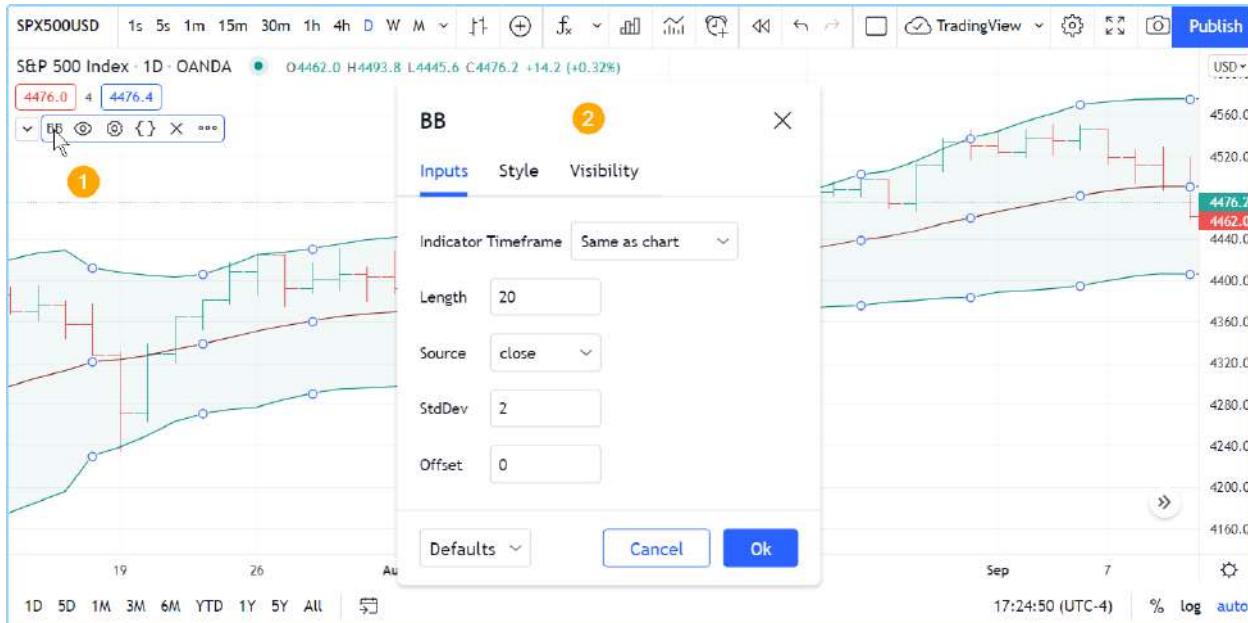
You can also search for scripts using the homepage’s “Search” field, and filter scripts using different criteria. The Help Center has a page explaining the [different types of scripts](#) that are available.

The scripts stream shows script *widgets*, i.e., placeholders showing a miniature view of each publication’s chart and description, and its author. By clicking on it you will open the *script’s page*, where you can see the script on a chart, read the author’s description, like the script, leave comments or read the script’s source code if it was published open-source.

Once you find an interesting script in the Community Scripts, follow the instructions in the Help Center to [load it on your chart](#).

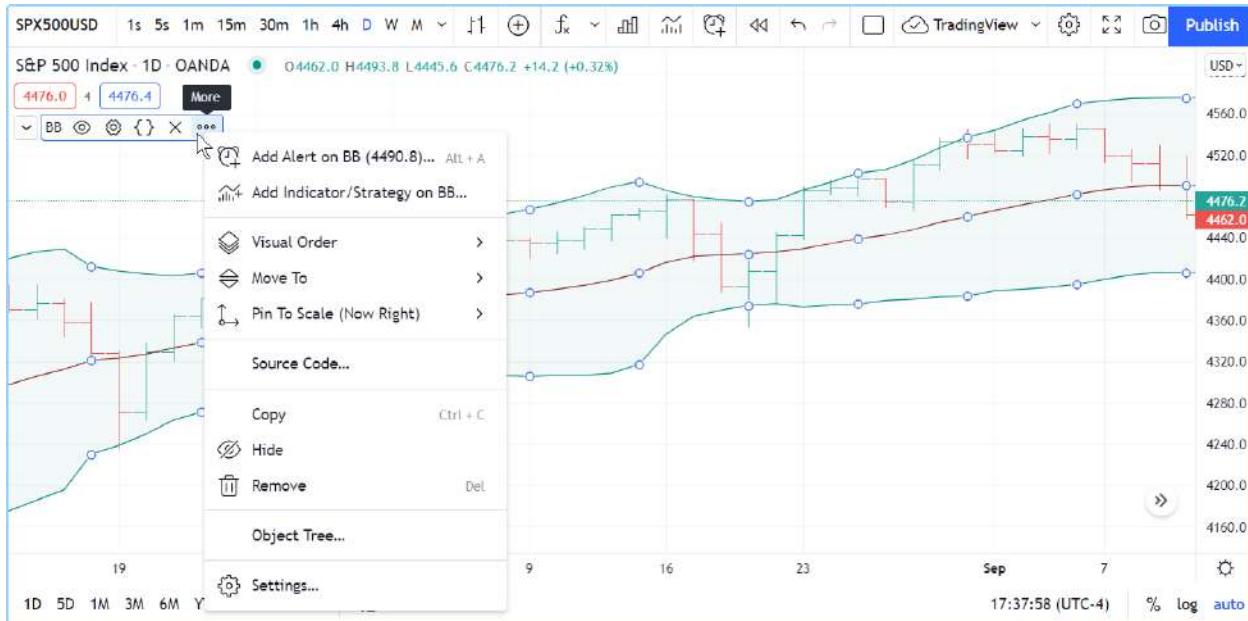
## Changing script settings

Once a script is loaded on the chart, you can double-click on its name (#1) to bring up its “Settings/Inputs” tab (#2):



The “Inputs” tab allows you to change the settings which the script’s author has decided to make editable. You can configure some of the script’s visuals using the “Style” tab of the same dialog box, and which timeframes the script should appear on using the “Visibility” tab.

Other settings are available to all scripts from the buttons that appear to the right of its name when you mouse over it, and from the “More” menu (the three dots):



## 2.1.3 Reading scripts

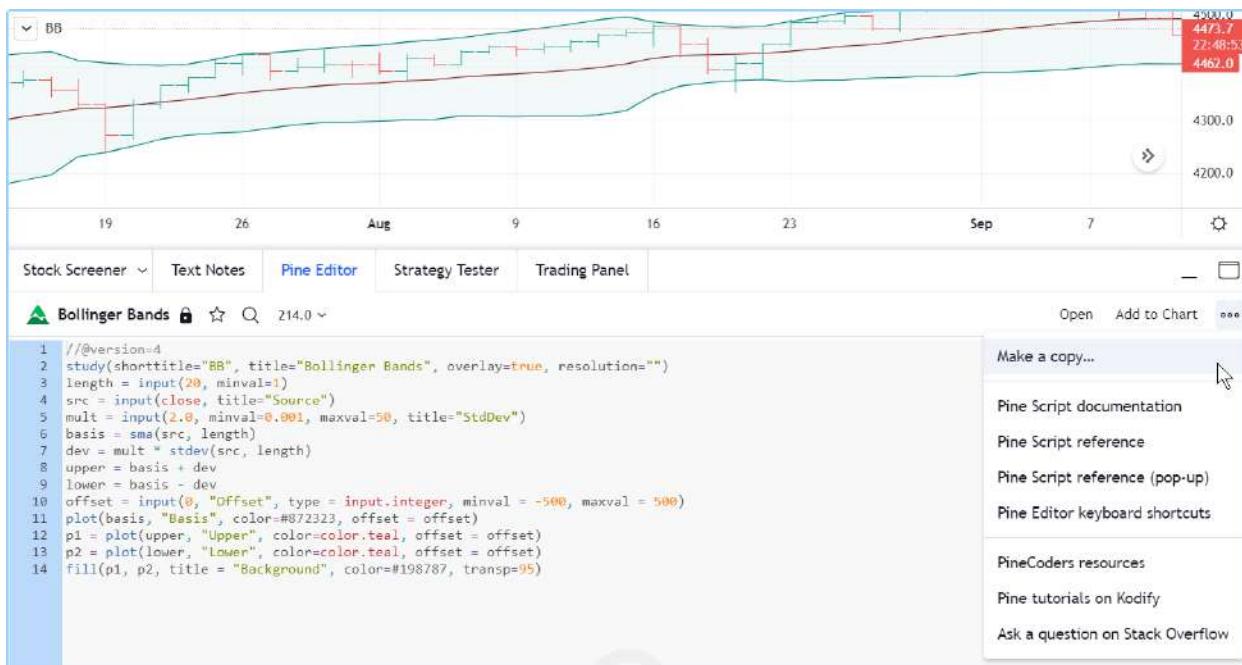
Reading code written by **good** programmers is the best way to develop your understanding of the language. This is as true for Pine Script™ as it is for all other programming languages. Finding good open-source Pine Script™ code is relatively easy. These are reliable sources of code written by good programmers on TradingView:

- The TradingView built-in indicators
- Scripts selected as Editors' Picks
- Scripts by the authors the PineCoders account follows
- Many scripts by authors with high reputation and open-source publications.

Reading code from [Community Scripts](#) is easy; if you don't see a grey or red "lock" icon in the upper-right corner of the script's widget, this indicates the script is open-source. By opening its script page, you will be able to see its source.

To see the code of TradingView built-ins, load the indicator on your chart, then hover over its name and select the "Source code" curly braces icon (if you don't see it, it's because the indicator's source is unavailable). When you click on the icon, the Pine Editor will open and from there, you can see the script's code. If you want to play with it, you will need to use the Editor's "More" menu button at the top-right of the Editor's pane, and select "Make a copy...". You will then be able to modify and save the code. Because you will have created a different version of the script, you will need to use the Editor's "Add to Chart" button to add that new copy to the chart.

This shows the Pine Editor having just opened after we selected the "View source" button from the indicator on our chart. We are about to make a copy of its source because it is read-only for now (indicated by the "lock" icon near its filename in the Editor):



You can also open TradingView built-in indicators from the Pine Editor (accessible from the "Pine Editor" tab at the bottom of the chart) by using the "Open/New default built-in script..." menu selection.

## 2.1.4 Writing scripts

We have built Pine Script™ to empower both budding and seasoned traders to create their own trading tools. We have designed it so it is relatively easy to learn for first-time programmers — although learning a first programming language, like trading, is rarely **very** easy for anyone — yet powerful enough for knowledgeable programmers to build tools of moderate complexity.

Pine Script™ allows you to write three types of scripts:

- **Indicators** like RSI, MACD, etc.
- **Strategies** which include logic to issue trading orders and can be backtested and forward-tested.
- **Libraries** which are used by more advanced programmers to package oft-used functions that can be reused by other scripts.

The next step we recommend is to write your *first indicator*.



## 2.2 First indicator

- *The Pine Editor*
- *First version*
- *Second version*
- *Next*

### 2.2.1 The Pine Editor

The Pine Editor is where you will be working on your scripts. While you can use any text editor you want to write your Pine scripts, using our Editor has many advantages:

- It highlights your code following Pine Script™ syntax.
- It pops up syntax reminders for built-in and library functions when you hover over them.
- It provides quick access to the Pine Script™ v5 Reference Manual popup when you `ctrl + click / cmd + click` on Pine Script™ keywords.
- It provides an auto-complete feature that you can activate with `ctrl + space / cmd + space`.
- It makes the write/compile/run cycle fast because saving a new version of a script loaded on the chart also executes it immediately.
- While not as feature-rich as the top editors out there, it provides key functionality such as search and replace, multi-cursor and versioning.

To open the Editor, click on the “Pine Editor” tab at the bottom of your TradingView chart. This will open up the Editor’s pane.

## 2.2.2 First version

We will now create our first working Pine script, an implementation of the **MACD** indicator in Pine Script™:

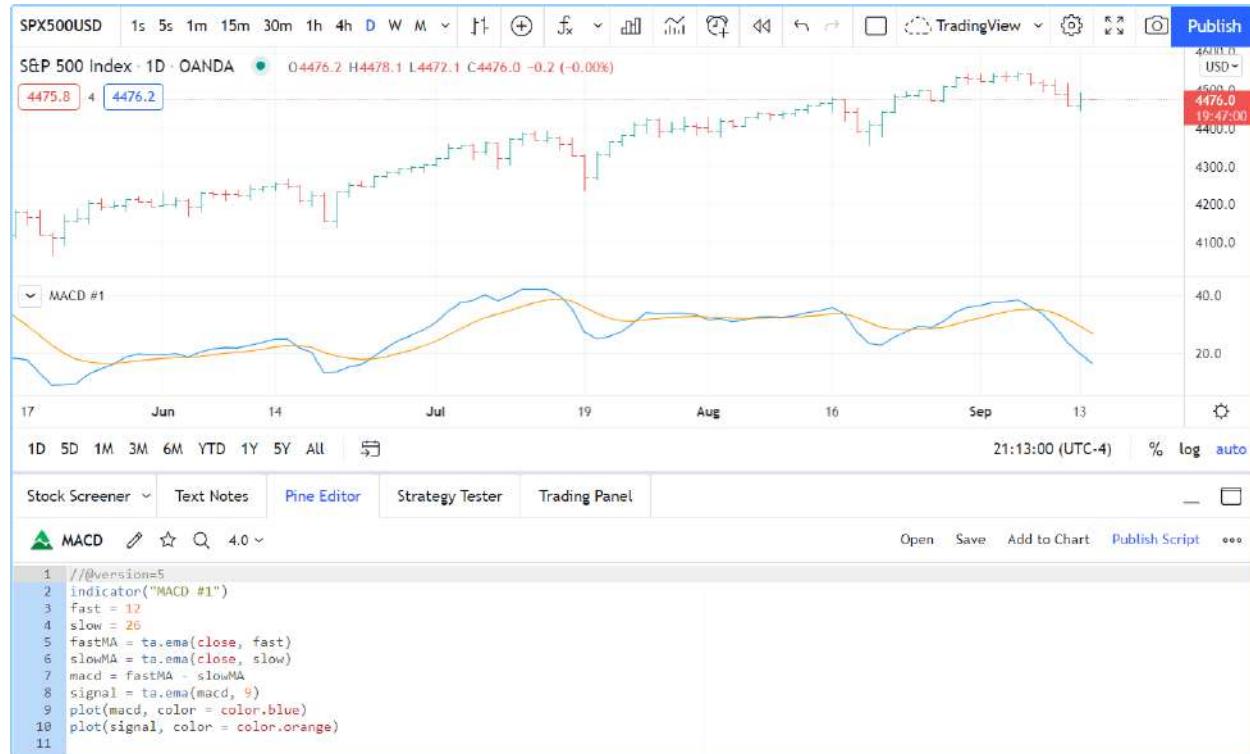
```

1 //@version=5
2 indicator("MACD #1")
3 fast = 12
4 slow = 26
5 fastMA = ta.ema(close, fast)
6 slowMA = ta.ema(close, slow)
7 macd = fastMA - slowMA
8 signal = ta.ema(macd, 9)
9 plot(macd, color = color.blue)
10 plot(signal, color = color.orange)

```

- Start by bringing up the “Open” dropdown menu at the top right of the Editor and choose “New blank indicator”.
- Then copy the example script above, taking care not to include the line numbers in your selection.
- Select all the code already in the editor and replace it with the example script.
- Click “Save” and choose a name for your script. Your script is now saved in TradingView’s cloud, but under your account’s name. Nobody but you can use it.
- Click “Add to Chart” in the Editor’s menu bar. The MACD indicator appears in a separate *Pane* under your chart.

Your first Pine script is running on your chart, which should look like this:



Let’s look at our script’s code, line by line:

**Line 1: //@version=5**

This is a *compiler annotation* telling the compiler the script will use version 5 of Pine Script™.

**Line 2: indicator("MACD #1")**

Defines the name of the script that will appear on the chart as “MACD”.

**Line 3: fast = 12**

Defines a `fast` integer variable which will be the length of the fast EMA.

**Line 4: slow = 26**

Defines a `slow` integer variable which will be the length of the slow EMA.

**Line 5: fastMA = ta.ema(close, fast)**

Defines the variable `fastMA`, containing the result of the EMA calculation (Exponential Moving Average) with a length equal to `fast` (12), on the `close` series, i.e., the closing price of bars.

**Line 6: slowMA = ta.ema(close, slow)**

Defines the variable `slowMA`, containing the result of the EMA calculation with a length equal to `slow` (26), from `close`.

**Line 7: macd = fastMA - slowMA**

Defines the variable `macd` as the difference between the two EMAs.

**Line 8: signal = ta.ema(macd, 9)**

Defines the variable `signal` as a smoothed value of `macd` using the EMA algorithm (Exponential Moving Average) with a length of 9.

**Line 9: plot(macd, color = color.blue)**

Calls the `plot` function to output the variable `macd` using a blue line.

**Line 10: plot(signal, color = color.orange)**

Calls the `plot` function to output the variable `signal` using an orange line.

### 2.2.3 Second version

The first version of our script calculated MACD “manually”, but because Pine Script™ is designed to write indicators and strategies, built-in Pine Script™ functions exist for many common indicators, including one for... MACD: `ta.macd()`.

This is the second version of our script:

```

1 //@version=5
2 indicator("MACD #2")
3 fastInput = input(12, "Fast length")
4 slowInput = input(26, "Slow length")
5 [macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)
6 plot(macdLine, color = color.blue)
7 plot(signalLine, color = color.orange)

```

Note that we have:

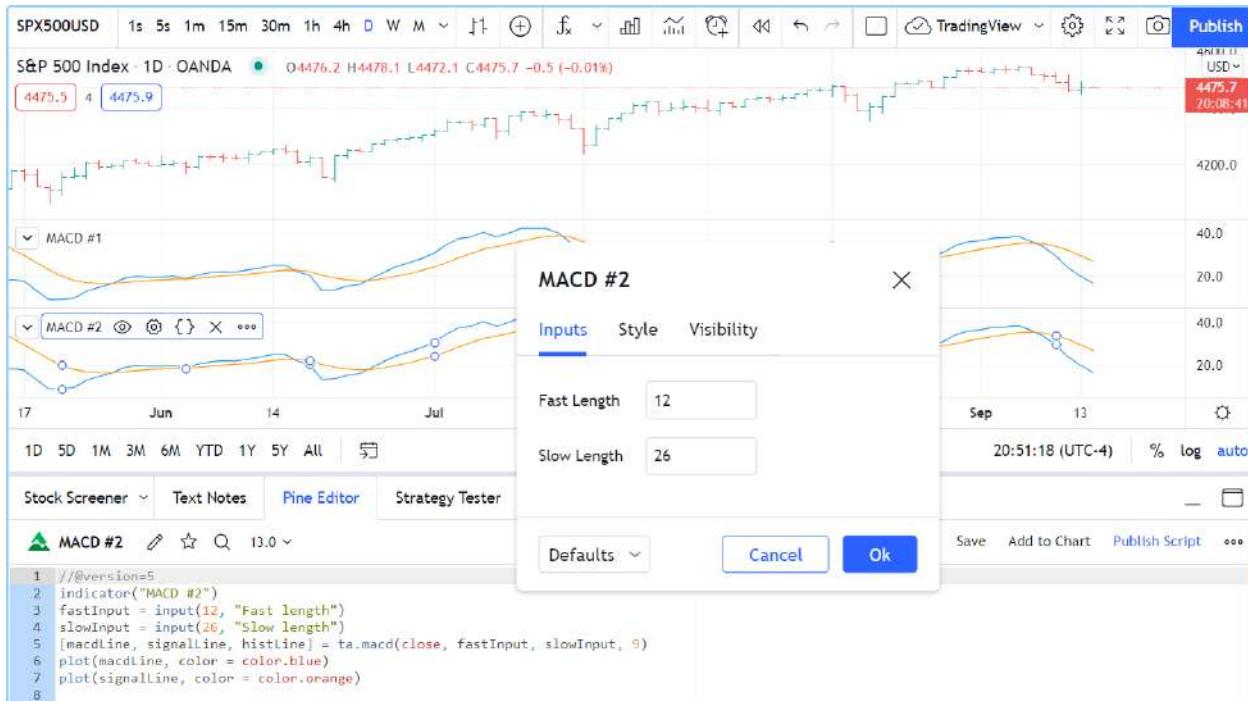
- Added inputs so we can change the lengths for the MAs
- We now use the `ta.macd()` built-in to calculate our MACD, which saves us three line and makes our code easier to read.

Let's repeat the same process as before to copy that code in a new indicator:

- Start by bringing up the “Open” dropdown menu at the top right of the Editor and choose “New blank indicator”.
- Then copy the example script above, again taking care not to include the line numbers in your selection.
- Select all the code already in the editor and replace it with the second version of our script.

- Click “Save” and choose a name for your script different than the previous one.
- Click “Add to Chart” in the Editor’s menu bar. The “MACD #2” indicator appears in a separate *Pane* under the “MACD #1” indicator.

Your second Pine script is running on your chart. If you double-click on the indicator’s name on your chart, you will bring up the script’s “Settings/Inputs” tab, where you can now change the slow and fast lengths:



Let’s look at the lines that have changed in the second version of our script:

#### Line 2: `indicator("MACD #2")`

We have changed #1 to #2 so the second version of our indicator displays a different name on the chart.

#### Line 3: `fastInput = input(12, "Fast length")`

Instead of assigning a constant value to a variable, we have used the `input()` function so we can change the value in our script’s “Settings/Inputs” tab. 12 will be the default value and the field’s label will be “Fast length”. If the value is changed in the “Inputs” tab, the `fastInput` variable’s content will contain the new value and the script will re-execute on the chart with that new value. Note that, as our Pine Script™ *Style Guide* recommends, we add `Input` to the end of the variable’s name to remind us, later in the script, that its value comes from a user input.

#### Line 4: `slowInput = input(26, "Slow length")`

We do the same for the slow length, taking care to use a different variable name, default value and text string for the field’s label.

#### Line 5: `[macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)`

This is where we call the `ta.macd()` built-in to perform all the first version’s calculations in one line only. The function requires four parameters (the values after the function name, enclosed in parentheses). It returns three values into the three variables instead of only one, like the functions we used until now, which is why we need to enclose the list of three variables receiving the function’s result in square brackets, to the left of the = sign. Note that two of the values we pass to the function are the “input” variables containing the fast and slow lengths: `fastInput` and `slowInput`.

#### Line 6 and 7:

The variable names we are plotting there have changed, but the lines are doing the same thing as in our first version. Our second version performs the same calculations as our first, but we can change the two lengths used to calculate it. Our code is also simpler and shorter by three lines. We have improved our script.

### 2.2.4 Next

We now recommend you go to our [Next Steps](#) page.



## 2.3 Next steps

- “*indicators*” vs “*strategies*”
- *How scripts are executed*
- *Time series*
- *Publishing scripts*
- *Getting around the Pine Script™ documentation*
- *Where to go from here?*

After your [first steps](#) and your [first indicator](#), let us explore a bit more of the Pine Script™ landscape by sharing some pointers to guide you in your journey to learn Pine Script™.

### 2.3.1 “indicators” vs “strategies”

Pine Script™ *strategies* are used to backtest on historical data and forward test on open markets. In addition to indicator calculations, they contain `strategy . * ()` calls to send trade orders to Pine Script™’s broker emulator, which can then simulate their execution. Strategies display backtest results in the “Strategy Tester” tab at the bottom of the chart, next to the “Pine Editor” tab.

Pine Script™ indicators also contain calculations, but cannot be used in backtesting. Because they do not require the broker emulator, they use less resources and will run faster. It is thus advantageous to use indicators whenever you can.

Both indicators and strategies can run in either overlay mode (over the chart’s bars) or pane mode (in a separate section below or above the chart). Both can also plot information in their respective space, and both can generate [alert events](#).

### 2.3.2 How scripts are executed

A Pine script is **not** like programs in many programming languages that execute once and then stop. In the Pine Script™ *runtime* environment, a script runs in the equivalent of an invisible loop where it is executed once on each bar of whatever chart you are on, from left to right. Chart bars that have already closed when the script executes on them are called *historical bars*. When execution reaches the chart's last bar and the market is open, it is on the *realtime bar*. The script then executes once every time a price or volume change is detected, and one last time for that realtime bar when it closes. That realtime bar then becomes an *elapsed realtime bar*. Note that when the script executes in realtime, it does not recalculate on all the chart's historical bars on every price/volume update. It has already calculated once on those bars, so it does not need to recalculate them on every chart tick. See the [Execution model](#) page for more information.

When a script executes on a historical bar, the `close` built-in variable holds the value of that bar's close. When a script executes on the realtime bar, `close` returns the **current** price of the symbol until the bar closes.

Contrary to indicators, strategies normally execute only once on realtime bars, when they close. They can also be configured to execute on each price/volume update if that is what you need. See the page on [Strategies](#) for more information, and to understand how strategies calculate differently than indicators.

### 2.3.3 Time series

The main data structure used in Pine Script™ is called a *time series*. Time series contain one value for each bar the script executes on, so they continuously expand as the script executes on more bars. Past values of the time series can be referenced using the history-referencing operator: `[]`. `close[1]`, for example, refers to the value of `close` on the bar preceding the one where the script is executing.

While this indexing mechanism may remind many programmers of arrays, a time series is different and thinking in terms of arrays will be detrimental to understanding this key Pine Script™ concept. A good comprehension of both the [execution model](#) and *time series* is essential in understanding how Pine scripts work. If you have never worked with data organized in time series before, you will need practice to put them to work for you. Once you familiarize yourself with these key concepts, you will discover that by combining the use of time series with our built-in functions specifically designed to handle them efficiently, much can be accomplished in very few lines of code.

### 2.3.4 Publishing scripts

TradingView is home to a large community of Pine Script™ programmers and millions of traders from all around the world. Once you become proficient enough in Pine Script™, you can choose to share your scripts with other traders. Before doing so, please take the time to learn Pine Script™ well-enough to supply traders with an original and reliable tool. All publicly published scripts are analyzed by our team of moderators and must comply with our [Script Publishing Rules](#), which require them to be original and well-documented.

If want to use Pine scripts for your own use, simply write them in the Pine Editor and add them to your chart from there; you don't have to publish them to use them. If you want to share your scripts with just a few friends, you can publish them privately and send your friends the browser's link to your private publication. See the page on [Publishing](#) for more information.

## 2.3.5 Getting around the Pine Script™ documentation

While reading code from published scripts is no doubt useful, spending time in our documentation will be necessary to attain any degree of proficiency in Pine Script™. Our two main sources of documentation on Pine Script™ are:

- This Pine Script™ v5 User Manual
- Our Pine Script™ v5 Reference Manual

The Pine Script™ v5 User Manual is in HTML format and in English only.

The Pine Script™ v5 Reference Manual documents what each variable, function or keyword does. It is an essential tool for all Pine Script™ programmers; your life will be miserable if you try to write scripts of any reasonable complexity without consulting it. It exists in two formats: the HTML format we just linked to, and the popup version, which can be accessed from the Pine Editor, by either `ctrl + clicking` on a keyword, or by using the Editor's "More/Pine Script™ reference (pop-up)" menu. The Reference Manual is translated in other languages.

There are five different versions of Pine Script™. Ensure the documentation you use corresponds to the Pine Script™ version you are coding with.

## 2.3.6 Where to go from here?

This Pine Script™ v5 User Manual contains numerous examples of code used to illustrate the concepts we discuss. By going through it, you will be able to both learn the foundations of Pine Script™ and study the example scripts. Reading about key concepts and trying them out right away with real code is a productive way to learn any programming language. As you hopefully have already done in the [First indicator](#) page, copy this documentation's examples in the Editor and play with them. Explore! You won't break anything.

This is how the Pine Script™ v5 User Manual you are reading is organized:

- The [Language](#) section explains the main components of the Pine Script™ language and how scripts execute.
- The [Concepts](#) section is more task-oriented. It explains how to do things in Pine Script™.
- The [Writing](#) section explores tools and tricks that will help you write and publish scripts.
- The [FAQ](#) section answers common questions from Pine Script™ programmers.
- The [Error messages](#) page documents causes and fixes for the most common runtime and compiler errors.
- The [Release Notes](#) page is where you can follow the frequent updates to Pine Script™.
- The [Migration guides](#) section explains how to port between different versions of Pine Script™.
- The [Where can I get more information](#) page lists other useful Pine Script™-related content, including where to ask questions when you are stuck on code.

We wish you a successful journey with Pine Script™... and trading!





## 3.1 Execution model

- *Calculation based on historical bars*
- *Calculation based on realtime bars*
- *Events triggering the execution of a script*
- *More information*
- *Historical values of functions*

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

The execution model determines how your script is executed on charts, and thus how the code you write in scripts works. Your code would do nothing were it not for Pine Script™'s runtime, which kicks in after your code has compiled and it is executed on your chart because one of the *events triggering the execution of a script* has occurred.

When a Pine script is loaded on a chart it executes once on each historical bar using the available OHLCV (open, high, low, close, volume) values for each bar. Once the script's execution reaches the rightmost bar in the dataset, if trading is currently active on the chart's symbol, then Pine Script™ *indicators* will execute once every time an *update* occurs, i.e., price or volume changes. Pine Script™ *strategies* will by default only execute when the rightmost bar closes, but they can also be configured to execute on every update, like indicators do.

All symbol/timeframe pairs have a dataset comprising a limited number of bars. When you scroll a chart to the left to see the dataset's earlier bars, the corresponding bars are loaded on the chart. The loading process stops when there are no more bars for that particular symbol/timeframe pair or the *maximum number of bars* your account type permits has been loaded. You can scroll the chart to the left until the very first bar of the dataset, which has an index value of 0 (see `bar_index`).

When the script first runs on a chart, all bars in a dataset are *historical bars*, except the rightmost one if a trading session is active. When trading is active on the rightmost bar, it is called the *realtime bar*. The realtime bar updates when a price

or volume change is detected. When the realtime bar closes, it becomes an *elapsed realtime bar* and a new realtime bar opens.

### 3.1.1 Calculation based on historical bars

Let's take a simple script and follow its execution on historical bars:

```
1 //@version=5
2 indicator("My Script", overlay = true)
3 src = close
4 a = ta.sma(src, 5)
5 b = ta.sma(src, 50)
6 c = ta.cross(a, b)
7 plot(a, color = color.blue)
8 plot(b, color = color.black)
9 plotshape(c, color = color.red)
```

On historical bars, a script executes at the equivalent of the bar's close, when the OHLCV values are all known for that bar. Prior to execution of the script on a bar, the built-in variables such as `open`, `high`, `low`, `close`, `volume` and `time` are set to values corresponding to those from that bar. A script executes **once per historical bar**.

Our example script is first executed on the very first bar of the dataset at index 0. Each statement is executed using the values for the current bar. Accordingly, on the first bar of the dataset, the following statement:

```
src = close
```

initializes the variable `src` with the `close` value for that first bar, and each of the next lines is executed in turn. Because the script only executes once for each historical bar, the script will always calculate using the same `close` value for a specific historical bar.

The execution of each line in the script produces calculations which in turn generate the indicator's output values, which can then be plotted on the chart. Our example uses the `plot` and `plotshape` calls at the end of the script to output some values. In the case of a strategy, the outcome of the calculations can be used to plot values or dictate the orders to be placed.

After execution and plotting on the first bar, the script is executed on the dataset's second bar, which has an index of 1. The process then repeats until all historical bars in the dataset are processed and the script reaches the rightmost bar on the chart.



### 3.1.2 Calculation based on realtime bars

The behavior of a Pine script on the realtime bar is very different than on historical bars. Recall that the realtime bar is the rightmost bar on the chart when trading is active on the chart's symbol. Also, recall that strategies can behave in two different ways in the realtime bar. By default, they only execute when the realtime bar closes, but the `calc_on_every_tick` parameter of the `strategy` declaration statement can be set to true to modify the strategy's behavior so that it executes each time the realtime bar updates, as indicators do. The behavior described here for indicators will thus only apply to strategies using `calc_on_every_tick=true`.

The most important difference between execution of scripts on historical and realtime bars is that while they execute only once on historical bars, scripts execute every time an update occurs during a realtime bar. This entails that built-in variables such as `high`, `low` and `close` which never change on a historical bar, **can** change at each of a script's iteration in the realtime bar. Changes in the built-in variables used in the script's calculations will, in turn, induce changes in the results of those calculations. This is required for the script to follow the realtime price action. As a result, the same script may produce different results every time it executes during the realtime bar.

**Note:** In the realtime bar, the `close` variable always represents the **current price**. Similarly, the `high` and `low` built-in variables represent the highest high and lowest low reached since the realtime bar's beginning. Pine Script™'s built-in variables will only represent the realtime bar's final values on the bar's last update.

Let's follow our script example in the realtime bar.

When the script arrives on the realtime bar it executes a first time. It uses the current values of the built-in variables to produce a set of results and plots them if required. Before the script executes another time when the next update happens, its user-defined variables are reset to a known state corresponding to that of the last *commit* at the close of the previous bar. If no commit was made on the variables because they are initialized every bar, then they are reinitialized. In both cases their last calculated state is lost. The state of plotted labels and lines is also reset. This resetting of the script's user-defined variables and drawings prior to each new iteration of the script in the realtime bar is called *rollback*. Its effect is to reset the script to the same known state it was in when the realtime bar opened, so calculations in the realtime bar are always performed from a clean state.

The constant recalculation of a script's values as price or volume changes in the realtime bar can lead to a situation where variable `c` in our example becomes true because a cross has occurred, and so the red marker plotted by the script's last line would appear on the chart. If on the next price update the price has moved in such a way that the `close` value no

longer produces calculations making `c` true because there is no longer a cross, then the marker previously plotted will disappear.

When the realtime bar closes, the script executes a last time. As usual, variables are rolled back prior to execution. However, since this iteration is the last one on the realtime bar, variables are committed to their final values for the bar when calculations are completed.

To summarize the realtime bar process:

- A script executes **at the open of the realtime bar and then once per update**.
- Variables are rolled back **before every realtime update**.
- Variables are committed **once at the closing bar update**.

### 3.1.3 Events triggering the execution of a script

A script is executed on the complete set of bars on the chart when one of the following events occurs:

- A new symbol or timeframe is loaded on a chart.
- A script is saved or added to the chart, from the Pine Script™ Editor or the chart's "Indicators & strategies" dialog box.
- A value is modified in the script's "Settings/Inputs" dialog box.
- A value is modified in a strategy's "Settings/Properties" dialog box.
- A browser refresh event is detected.

A script is executed on the realtime bar when trading is active and:

- One of the above conditions occurs, causing the script to execute on the open of the realtime bar, or
- The realtime bar updates because a price or volume change was detected.

Note that when a chart is left untouched when the market is active, a succession of realtime bars which have been opened and then closed will trail the current realtime bar. While these *elapsed realtime bars* will have been *confirmed* because their variables have all been committed, the script will not yet have executed on them in their *historical* state, since they did not exist when the script was last run on the chart's dataset.

When an event triggers the execution of the script on the chart and causes it to run on those bars which have now become historical bars, the script's calculation can sometimes vary from what they were when calculated on the last closing update of the same bars when they were realtime bars. This can be caused by slight variations between the OHLCV values saved at the close of realtime bars and those fetched from data feeds when the same bars have become historical bars. This behavior is one of the possible causes of *repainting*.

### 3.1.4 More information

- The built-in `barstate.*` variables provide information on *the type of bar or the event* where the script is executing. The page where they are documented also contains a script that allows you to visualize the difference between elapsed realtime and historical bars, for example.
- The [Strategies](#) page explains the details of strategy calculations, which are not identical to those of indicators.

### 3.1.5 Historical values of functions

Every function call in Pine leaves a trail of historical values that a script can access on subsequent bars using the `[ ]` operator. The historical series of functions depend on successive calls to record the output on every bar. When a script does not call functions on each bar, it can produce an inconsistent history that may impact calculations and results, namely when it depends on the continuity of their historical series to operate as expected. The compiler warns users in these cases to make them aware that the values from a function, whether built-in or user-defined, might be misleading.

To demonstrate, let's write a script that calculates the index of the current bar and outputs that value on every second bar. In the following script, we've defined a `calcBarIndex()` function that adds 1 to the previous value of its internal `index` variable on every bar. The script calls the function on each bar that the `condition` returns `true` on (every other bar) to update the `customIndex` value. It plots this value alongside the built-in `bar_index` to validate the output:



```

1 // @version=5
2 indicator("My script")
3
4 // @function Calculates the index of the current bar by adding 1 to its own value from
5 // the previous bar.
6 // The first bar will have an index of 0.
7 calcBarIndex() =>
8     int index = na
9     index := nz(index[1], replacement = -1) + 1
10
11 // @variable Returns `true` on every other bar.
12 condition = bar_index % 2 == 0
13
14 int customIndex = na
15
16 // Call `calcBarIndex()` when the `condition` is `true`. This prompts the compiler to
17 // raise a warning.
18 if condition
19     customIndex := calcBarIndex()
20
21 plot(bar_index, "Bar index", color = color.green)
22 plot(customIndex, "Custom index", color = color.red, style = plot.style_cross)

```

### Note that:

- The `nz()` function replaces `na` values with a specified replacement value (0 by default). On the first bar of the script, when the `index` series has no history, the `na` value is replaced with -1 before adding 1 to return an initial value of 0.

Upon inspecting the chart, we see that the two plots differ wildly. The reason for this behavior is that the script called `calcBarIndex()` within the scope of an `if` structure on every other bar, resulting in a historical output inconsistent with the `bar_index` series. When calling the function once every two bars, internally referencing the previous value of `index` gets the value from two bars ago, i.e., the last bar the function executed on. This behavior results in a `customIndex` value of half that of the built-in `bar_index`.

To align the `calcBarIndex()` output with the `bar_index`, we can move the function call to the script's global scope. That way, the function will execute on every bar, allowing its entire history to be recorded and referenced rather than only the results from every other bar. In the code below, we've defined a `globalScopeBarIndex` variable in the global scope and assigned it to the return from `calcBarIndex()` rather than calling the function locally. The script sets the `customIndex` to the value of `globalScopeBarIndex` on the occurrence of the condition:



```

1 // @version=5
2 indicator("My script")
3
4 //@function Calculates the index of the current bar by adding 1 to its own value from
5 // the previous bar.
6 // The first bar will have an index of 0.
7 calcBarIndex() =>
8     int index = na
9     index := nz(index[1], replacement = -1) + 1
10
11 // @variable Returns `true` on every second bar.
12 condition = bar_index % 2 == 0
13
14 globalScopeBarIndex = calcBarIndex()
15 int customIndex = na
16
17 // Assign `customIndex` to `globalScopeBarIndex` when the `condition` is `true`. This
18 // won't produce a warning.
19 if condition

```

(continues on next page)

(continued from previous page)

```

18 customIndex := globalScopeBarIndex
19
20 plot(bar_index, "Bar index", color = color.green)
21 plot(customIndex, "Custom index", color = color.red, style = plot.style_cross)

```

This behavior can also radically impact built-in functions that reference history internally. For example, the `ta.sma()` function references its past values “under the hood”. If a script calls this function conditionally rather than on every bar, the values within the calculation can change significantly. We can ensure calculation consistency by assigning `ta.sma()` to a variable in the global scope and referencing that variable’s history as needed.

The following example calculates three SMA series: `controlSMA`, `localsMA`, and `globalsMA`. The script calculates `controlSMA` in the global scope and `localsMA` within the local scope of an `if` structure. Within the `if` structure, it also updates the value of `globalsMA` using the `controlSMA` value. As we can see, the values from the `globalsMA` and `controlSMA` series align, whereas the `localsMA` series diverges from the other two because it uses an incomplete history, which affects its calculations:



```

1 // @version=5
2 indicator("My script")
3
4 // @variable Returns `true` on every second bar.
5 condition = bar_index % 2 == 0
6
7 controlSMA = ta.sma(close, 20)
8 float globalsMA = na
9 float localsMA = na
10
11 // Update `globalsMA` and `localsMA` when `condition` is `true`.
12 if condition
13     globalsMA := controlSMA      // No warning.
14     localsMA := ta.sma(close, 20) // Raises warning. This function depends on its_
15     ↪history to work as intended.
16
17 plot(controlSMA, "Control SMA", color = color.green)
18 plot(globalsMA, "Global SMA", color = color.blue, style = plot.style_cross)
19 plot(localsMA, "Local SMA", color = color.red, style = plot.style_cross)

```

## Why this behavior?

This behavior is required because forcing the execution of functions on each bar would lead to unexpected results in those functions that produce side effects, i.e., the ones that do something aside from returning the value. For example, the `label.new()` function creates a label on the chart, so forcing it to be called on every bar even when it is inside of an `if` structure would create labels where they should not logically appear.

## Exceptions

Not all built-in functions use their previous values in their calculations, meaning not all require execution on every bar. For example, `math.max()` compares all arguments passed into it to return the highest value. Such functions that do not interact with their history in any way do not require special treatment.

If the usage of a function within a conditional block does not cause a compiler warning, it's safe to use without impacting calculations. Otherwise, move the function call to the global scope to force consistent execution. When keeping a function call within a conditional block despite the warning, ensure the output is correct at the very least to avoid unexpected results.



## 3.2 Time series

Much of the power of Pine Script™ stems from the fact that it is designed to process *time series* efficiently. Time series are not a qualified type; they are the fundamental structure Pine Script™ uses to store the successive values of a variable over time, where each value is tethered to a point in time. Since charts are composed of bars, each representing a particular point in time, time series are the ideal data structure to work with values that may change with time.

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

Take the built-in `open` variable, which contains the “open” price of each bar in the dataset, the *dataset* being all the bars on any given chart. If your script is running on a 5min chart, then each value in the `open` time series is the “open” price of the consecutive 5min chart bars. When your script refers to `open`, it is referring to the “open” price of the bar the script is executing on. To refer to past values in a time series, we use the `[]` history-referencing operator. When a script is executing on a given bar, `open[1]` refers to the value of the `open` time series on the previous bar.

While time series may remind programmers of arrays, they are totally different. Pine Script™ does use an array data structure, but it is a completely different concept than a time series.

Time series in Pine Script™, combined with its special type of runtime engine and built-in functions, are what makes it easy to compute the cumulative total of `close` values without using a `for` loop, with only `ta.cum(close)`. This is possible because although `ta.cum(close)` appears rather static in a script, it is in fact executed on each bar, so its value becomes increasingly larger as the `close` value of each new bar is added to it. When the script reaches the rightmost bar of the chart, `ta.cum(close)` returns the sum of the `close` value from all bars on the chart.

Similarly, the mean of the difference between the last 14 `high` and `low` values can be expressed as `ta.sma(high - low, 14)`, or the distance in bars since the last time the chart made five consecutive higher highs as `barssince(rising(high, 5))`.

Even the result of function calls on successive bars leaves a trace of values in a time series that can be referenced using the `[]` history-referencing operator. This can be useful, for example, when testing the `close` of the current bar for a breach of the highest `high` in the last 10 bars, but excluding the current bar, which we could write as `breach = close > highest(close, 10) [1]`. The same statement could also be written as `breach = close > highest(close[1], 10)`.

The same looping logic on all bars is applied to function calls such as `plot(open)` which will repeat on each bar, successively plotting on the chart the value of `open` for each bar.

Do not confuse “time series” with the “series” qualifier. The *time series* concept explains how consecutive values of variables are stored in Pine Script™; the “series” qualifier denotes variables whose values can change bar to bar. Consider, for example, the `timeframe.period` built-in variable which has the “simple” qualifier and “string” type, meaning it is of the “simple string” qualified type. The “simple” qualifier entails that the variable’s value is established on bar zero (the first bar where the script executes) and will not change during the script’s execution on any of the chart’s bars. The variable’s value is the chart’s timeframe in string format, so “D” for a 1D chart, for example. Even though its value cannot change during the script, it would be syntactically correct in Pine Script™ (though not very useful) to refer to its value 10 bars ago using `timeframe.period[10]`. This is possible because the successive values of `timeframe.period` for each bar are stored in a time series, even though all the values in that particular time series are the same. Note, however, that when the `[]` operator is used to access past values of a variable, it yields a “series” qualified value, even when the variable without an offset uses a different qualifier, such as “simple” in the case of `timeframe.period`.

When you grasp how time series can be efficiently handled using Pine Script™’s syntax and its *execution model*, you can define complex calculations using little code.



### 3.3 Script structure

- *Version*
- *Declaration statement*
- *Code*
- *Comments*
- *Line wrapping*
- *Compiler annotations*

A Pine script follows this general structure:

```
<version>
<declaration_statement>
<code>
```

### 3.3.1 Version

A *compiler annotation* in the following form tells the compiler which of the versions of Pine Script™ the script is written in:

```
1 //@version=5
```

- The version number can be 1 to 5.
- The compiler annotation is not mandatory. When omitted, version 1 is assumed. It is strongly recommended to always use the latest version of the language.
- While it is syntactically correct to place the version compiler annotation anywhere in the script, it is much more useful to readers when it appears at the top of the script.

Notable changes to the current version of Pine Script™ are documented in the *Release notes*.

### 3.3.2 Declaration statement

All Pine scripts must contain one declaration statement, which is a call to one of these functions:

- `indicator()`
- `strategy()`
- `library()`

The declaration statement:

- Identifies the type of the script, which in turn dictates which content is allowed in it, and how it can be used and executed.
- Sets key properties of the script such as its name, where it will appear when it is added to a chart, the precision and format of the values it displays, and certain values that govern its runtime behavior, such as the maximum number of drawing objects it will display on the chart. With strategies, the properties include parameters that control backtesting, such as initial capital, commission, slippage, etc.

Each type of script has distinct requirements:

- Indicators must contain at least one function call which produces output on the chart (e.g., `plot()`, `plotshape()`, `barcolor()`, `line.new()`, etc.).
- Strategies must contain at least one `strategy.*()` call, e.g., `strategy.entry()`.
- Libraries must contain at least one exported *function* or *user-defined type*.

### 3.3.3 Code

Lines in a script that are not *comments* or *compiler annotations* are *statements*, which implement the script's algorithm. A statement can be one of these:

- variable declaration
- variable reassignment
- function declaration
- built-in function call, *user-defined function call* or *a library function call*
- `if`, `for`, `while`, `switch` or `type structure`.

Statements can be arranged in multiple ways:

- Some statements can be expressed in one line, like most variable declarations, lines containing only a function call or single-line function declarations. Lines can also be *wrapped* (continued on multiple lines). Multiple one-line statements can be concatenated on a single line by using the comma as a separator.
- Others statements such as structures or multi-line function declarations always require multiple lines because they require a *local block*. A local block must be indented by a tab or four spaces. Each local block defines a distinct *local scope*.
- Statements in the *global scope* of the script (i.e., which are not part of local blocks) cannot begin with white space (a space or a tab). Their first character must also be the line's first character. Lines beginning in a line's first position become by definition part of the script's *global scope*.

A simple valid Pine Script™ v5 indicator can be generated in the Pine Script™ Editor by using the “Open” button and choosing “New blank indicator”:

```
1 //@version=5
2 indicator("My Script")
3 plot(close)
```

This indicator includes three local blocks, one in the `f()` function declaration, and two in the variable declaration using an `if` structure:

```
1 //@version=5
2
3 indicator("", "", true)      // Declaration statement (global scope)
4
5 barIsUp() =>      // Function declaration (global scope)
6     close > open    // Local block (local scope)
7
8 plotColor = if barIsUp() // Variable declaration (global scope)
9     color.green      // Local block (local scope)
10 else
11     color.red       // Local block (local scope)
12
13 bgcolor(color.new(plotColor, 70)) // Call to a built-in function (global scope)
```

You can bring up a simple Pine Script™ v5 strategy by selecting “New blank strategy” instead:

```
1 //@version=5
2 strategy("My Strategy", overlay=true, margin_long=100, margin_short=100)
3
4 longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
5 if (longCondition)
6     strategy.entry("My Long Entry Id", strategy.long)
7
8 shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))
9 if (shortCondition)
10    strategy.entry("My Short Entry Id", strategy.short)
```

### 3.3.4 Comments

Double slashes (//) define comments in Pine Script™. Comments can begin anywhere on the line. They can also follow Pine Script™ code on the same line:

```

1 // @version=5
2 indicator("")
3 // This line is a comment
4 a = close // This is also a comment
5 plot(a)

```

The Pine Editor has a keyboard shortcut to comment/uncomment lines: **ctrl + /**. You can use it on multiple lines by highlighting them first.

### 3.3.5 Line wrapping

Long lines can be split on multiple lines, or “wrapped”. Wrapped lines must be indented with any number of spaces, provided it's not a multiple of four (those boundaries are used to indent local blocks):

```
a = open + high + low + close
```

may be wrapped as:

```

a = open +
    high +
    low +
    close

```

A long `plot()` call may be wrapped as:

```

plot(ta.correlation(src, ovr, length),
      color = color.new(color.purple, 40),
      style = plot.style_area,
      trackprice = true)

```

Statements inside user-defined function declarations can also be wrapped. However, since a local block must syntactically begin with an indentation (4 spaces or 1 tab), when splitting it onto the following line, the continuation of the statement must start with more than one indentation (not equal to a multiple of four spaces). For example:

```

updown(s) =>
    isEqual = s == s[1]
    isGrowing = s > s[1]
    ud = isEqual ?
        0 :
        isGrowing ?
            (nz(ud[1]) <= 0 ?
                1 :
                nz(ud[1])+1) :
            (nz(ud[1]) >= 0 ?
                -1 :
                nz(ud[1])-1)

```

You can use comments in wrapped lines:

```

1 //@version=5
2 indicator("")
3 c = open > close ? color.red :
4     high > high[1] ? color.lime : // A comment
5     low < low[1] ? color.blue : color.black
6 bgcolor(c)

```

### 3.3.6 Compiler annotations

Compiler annotations are *comments* that issue special instructions for a script:

- `//@version`= specifies the PineScript™ version that the compiler will use. The number in this annotation should not be confused with the script's revision number, which updates on every saved change to the code.
- `//@description` sets a custom description for scripts that use the `library()` declaration statement.
- `//@function`, `//@param` and `//@returns` add custom descriptions for a user-defined function, its parameters, and its result when placed above the function declaration.
- `//@type` and `//@field` add custom descriptions for a *user-defined type (UDT)* and its fields when placed above the type declaration.
- `//@variable` adds a custom description for a variable when placed above its declaration.
- `//@strategy_alert_message` provides a default message for strategy scripts to pre-fill the "Message" field in the alert creation dialogue.
- `//#region` and `//#endregion` create collapsible code regions in the Pine Editor. Clicking the dropdown arrow next to `//#region` collapses the lines of code between the two annotations.

This script draws a rectangle using three interactively selected points on the chart. It illustrates how compiler annotations can be used:



```

1 //@version=5
2 indicator("Triangle", "", true)
3
4 int    TIME_DEFAULT  = 0
5 float PRICE_DEFAULT = 0.0
6

```

(continues on next page)

(continued from previous page)

```

7  x1Input = input.time(TIME_DEFAULT,    "Point 1", inline = "1", confirm = true)
8  y1Input = input.price(PRICE_DEFAULT, "",           inline = "1", tooltip = "Pick point 1
   ↵", confirm = true)
9  x2Input = input.time(TIME_DEFAULT,    "Point 2", inline = "2", confirm = true)
10 y2Input = input.price(PRICE_DEFAULT, "",           inline = "2", tooltip = "Pick point 2
   ↵", confirm = true)
11 x3Input = input.time(TIME_DEFAULT,    "Point 3", inline = "3", confirm = true)
12 y3Input = input.price(PRICE_DEFAULT, "",           inline = "3", tooltip = "Pick point 3
   ↵", confirm = true)
13
14 // @type          Used to represent the coordinates and color to draw a triangle.
15 // @field time1  Time of first point.
16 // @field time2  Time of second point.
17 // @field time3  Time of third point.
18 // @field price1 Price of first point.
19 // @field price2 Price of second point.
20 // @field price3 Price of third point.
21 // @field lineColor Color to be used to draw the triangle lines.
22 type Triangle
23     int  time1
24     int  time2
25     int  time3
26     float price1
27     float price2
28     float price3
29     color lineColor
30
31 // @function Draws a triangle using the coordinates of the `t` object.
32 // @param t  (Triangle) Object representing the triangle to be drawn.
33 // @returns The ID of the last line drawn.
34 drawTriangle(Triangle t) =>
35     line.new(t.time1, t.price1, t.time2, t.price2, xloc = xloc.bar_time, color = t.
   ↵lineColor)
36     line.new(t.time2, t.price2, t.time3, t.price3, xloc = xloc.bar_time, color = t.
   ↵lineColor)
37     line.new(t.time1, t.price1, t.time3, t.price3, xloc = xloc.bar_time, color = t.
   ↵lineColor)
38
39 // Draw the triangle only once on the last historical bar.
40 if barstate.islastconfirmedhistory
41     // @variable Used to hold the Triangle object to be drawn.
42     Triangle triangle = Triangle.new()
43
44     triangle.time1  := x1Input
45     triangle.time2  := x2Input
46     triangle.time3  := x3Input
47     triangle.price1 := y1Input
48     triangle.price2 := y2Input
49     triangle.price3 := y3Input
50     triangle.lineColor := color.purple
51
52     drawTriangle(triangle)

```





## 3.4 Identifiers

Identifiers are names used for user-defined variables and functions:

- They must begin with an uppercase (A–Z) or lowercase (a–z) letter, or an underscore (\_).
- The next characters can be letters, underscores or digits (0–9).
- They are case-sensitive.

Here are some examples:

```
myVar
_myVar
my123Var
functionName
MAX_LEN
max_len
maxLen
3barsDown // NOT VALID!
```

The Pine Script™ *Style Guide* recommends using uppercase SNAKE\_CASE for constants, and camelCase for other identifiers:

```
GREEN_COLOR = #4CAF50
MAX_LOOKBACK = 100
int fastLength = 7
// Returns 1 if the argument is `true`, 0 if it is `false` or `na`.
zeroOne(boolValue) => boolValue ? 1 : 0
```



## 3.5 Operators

- *Introduction*
- *Arithmetic operators*
- *Comparison operators*
- *Logical operators*
- *`?:` ternary operator*
- *[ ] history-referencing operator*
- *Operator precedence*
- *`=` assignment operator*
- *`:=` reassignment operator*

### 3.5.1 Introduction

Some operators are used to build *expressions* returning a result:

- Arithmetic operators
- Comparison operators
- Logical operators
- The `?:` ternary operator
- The `[ ]` history-referencing operator

Other operators are used to assign values to variables:

- `=` is used to assign a value to a variable, **but only when you declare the variable** (the first time you use it)
- `:=` is used to assign a value to a **previously declared variable**. The following operators can also be used in such a way: `+=`, `-=`, `*=`, `/=`, `%=`

As is explained in the [Type system](#) page, *qualifiers* and *types* play a critical role in determining the type of results that expressions yield. This, in turn, has an impact on how and with what functions you will be allowed to use those results. Expressions always return a value with the strongest qualifier used in the expression, e.g., if you multiply an “input int” with a “series int”, the expression will produce a “series int” result, which you will not be able to use as the argument to `length` in `ta.ema()`.

This script will produce a compilation error:

```
1 // @version=5
2 indicator("")
3 lenInput = input.int(14, "Length")
4 factor = year > 2020 ? 3 : 1
5 adjustedLength = lenInput * factor
6 ma = ta.ema(close, adjustedLength) // Compilation error!
7 plot(ma)
```

The compiler will complain: *Cannot call ‘ta.ema’ with argument ‘length’=‘adjustedLength’. An argument of ‘series int’ type was used but a ‘simple int’ is expected.*: This is happening because `lenInput` is an “input int” but `factor` is a “series int” (it can only be determined by looking at the value of `year` on each bar). The `adjustedLength` variable is

thus assigned a “series int” value. Our problem is that the Reference Manual entry for `ta.ema()` tells us that its `length` parameter requires a “simple” value, which is a weaker qualifier than “series”, so a “series int” value is not allowed.

The solution to our conundrum requires:

- Using another moving average function that supports a “series int” length, such as `ta.sma()`, or
- Not using a calculation producing a “series int” value for our length.

### 3.5.2 Arithmetic operators

There are five arithmetic operators in Pine Script™:

+	Addition and string concatenation
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder after division)

The arithmetic operators above are all binary (means they need two *operands* — or values — to work on, like in `1 + 2`). The `+` and `-` also serve as unary operators (means they work on one operand, like `-1` or `+1`).

If both operands are numbers but at least one of these is of `float` type, the result will also be a `float`. If both operands are of `int` type, the result will also be an `int`. If at least one operand is `na`, the result is also `na`.

The `+` operator also serves as the concatenation operator for strings. `"EUR"+"USD"` yields the `"EURUSD"` string.

The `%` operator calculates the modulo by rounding down the quotient to the lowest possible value. Here is an easy example that helps illustrate how the modulo is calculated behind the scenes:

### 3.5.3 Comparison operators

There are six comparison operators in Pine Script™:

<	Less Than
<=	Less Than or Equal To
!=	Not Equal
==	Equal
>	Greater Than
>=	Greater Than or Equal To

Comparison operations are binary. If both operands have a numerical value, the result will be of type `bool`, i.e., `true`, `false` or `na`.

Examples:

```
1 > 2 // false
1 != 1 // false
close >= open // Depends on values of `close` and `open`
```

### 3.5.4 Logical operators

There are three logical operators in Pine Script™:

not	Negation
and	Logical Conjunction
or	Logical Disjunction

The operator `not` is unary. When applied to a `true`, operand the result will be `false`, and vice versa.

and operator truth table:

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

or operator truth table:

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

### 3.5.5 `?:` ternary operator

The `?:` ternary operator is used to create expressions of the form:

```
condition ? valueWhenConditionIsTrue : valueWhenConditionIsFalse
```

The ternary operator returns a result that depends on the value of `condition`. If it is `true`, then `valueWhenConditionIsTrue` is returned. If `condition` is `false` or `na`, then `valueWhenConditionIsFalse` is returned.

A combination of ternary expressions can be used to achieve the same effect as a `switch` structure, e.g.:

```
timeframe.isintraday ? color.red : timeframe.isdaily ? color.green : timeframe.  
↳ ismonthly ? color.blue : na
```

The example is calculated from left to right:

- If `timeframe.isintraday` is `true`, then `color.red` is returned. If it is `false`, then `timeframe.isdaily` is evaluated.
- If `timeframe.isdaily` is `true`, then `color.green` is returned. If it is `false`, then `timeframe.ismonthly` is evaluated.
- If `timeframe.ismonthly` is `true`, then `color.blue` is returned, otherwise `na` is returned.

Note that the return values on each side of the `:` are expressions — not local blocks, so they will not affect the limit of 500 local blocks per scope.

### 3.5.6 `[]` history-referencing operator

It is possible to refer to past values of *time series* using the `[]` history-referencing operator. Past values are values a variable had on bars preceding the bar where the script is currently executing — the *current bar*. See the [Execution model](#) page for more information about the way scripts are executed on bars.

The `[]` operator is used after a variable, expression or function call. The value used inside the square brackets of the operator is the offset in the past we want to refer to. To refer to the value of the `volume` built-in variable two bars away from the current bar, one would use `volume[2]`.

Because series grow dynamically, as the script moves on successive bars, the offset used with the operator will refer to different bars. Let's see how the value returned by the same offset is dynamic, and why series are very different from arrays. In Pine Script™, the `close` variable, or `close[0]` which is equivalent, holds the value of the current bar's "close". If your code is now executing on the **third** bar of the *dataset* (the set of all bars on your chart), `close` will contain the price at the close of that bar, `close[1]` will contain the price at the close of the preceding bar (the dataset's second bar), and `close[2]`, the first bar. `close[3]` will return `na` because no bar exists in that position, and thus its value is *not available*.

When the same code is executed on the next bar, the **fourth** in the dataset, `close` will now contain the closing price of that bar, and the same `close[1]` used in your code will now refer to the "close" of the third bar in the dataset. The close of the first bar in the dataset will now be `close[3]`, and this time `close[4]` will return `na`.

In the Pine Script™ runtime environment, as your code is executed once for each historical bar in the dataset, starting from the left of the chart, Pine Script™ is adding a new element in the series at index 0 and pushing the pre-existing elements in the series one index further away. Arrays, in comparison, can have constant or variable sizes, and their content or indexing structure is not modified by the runtime environment. Pine Script™ series are thus very different from arrays and only share familiarity with them through their indexing syntax.

When the market for the chart's symbol is open and the script is executing on the chart's last bar, the *realtime bar*, `close` returns the value of the current price. It will only contain the actual closing price of the realtime bar the last time the script is executed on that bar, when it closes.

Pine Script™ has a variable that contains the number of the bar the script is executing on: `bar_index`. On the first bar, `bar_index` is equal to 0 and it increases by 1 on each successive bar the script executes on. On the last bar, `bar_index` is equal to the number of bars in the dataset minus one.

There is another important consideration to keep in mind when using the `[]` operator in Pine Script™. We have seen cases when a history reference may return the `na` value. `na` represents a value which is not a number and using it in any expression will produce a result that is also `na` (similar to `NaN`). Such cases often happen during the script's calculations in the early bars of the dataset, but can also occur in later bars under certain conditions. If your code does not explicitly provide for handling these special cases, they can introduce invalid results in your script's calculations which can ripple through all the way to the realtime bar. The `na` and `nz` functions are designed to allow for handling such cases.

These are all valid uses of the `[]` operator:

```
high[10]
ta.sma(close, 10)[1]
ta.highest(high, 10)[20]
close > nz(close[1], open)
```

Note that the `[]` operator can only be used once on the same value. This is not allowed:

```
close[1][2] // Error: incorrect use of [] operator
```

### 3.5.7 Operator precedence

The order of calculations is determined by the operators' precedence. Operators with greater precedence are calculated first. Below is a list of operators sorted by decreasing precedence:

Precedence	Operator
9	[ ]
8	unary +, unary -, not
7	*, /, %
6	+, -
5	>, <, >=, <=
4	==, !=
3	and
2	or
1	? :

If in one expression there are several operators with the same precedence, then they are calculated left to right.

If the expression must be calculated in a different order than precedence would dictate, then parts of the expression can be grouped together with parentheses.

### 3.5.8 `=` assignment operator

The = operator is used to assign a variable when it is initialized — or declared —, i.e., the first time you use it. It says *this is a new variable that I will be using, and I want it to start on each bar with this value.*

These are all valid variable declarations:

```
i = 1
MS_IN_ONE_MINUTE = 1000 * 60
showPlotInput = input.bool(true, "Show plots")
pHi = pivothigh(5, 5)
plotColor = color.green
```

See the [Variable declarations](#) page for more information on how to declare variables.

### 3.5.9 `:=` reassignment operator

The := is used to *reassign* a value to an existing variable. It says *use this variable that was declared earlier in my script, and give it a new value.*

Variables which have been first declared, then reassigned using :=, are called *mutable* variables. All the following examples are valid variable reassessments. You will find more information on how var works in the section on the [`var` declaration mode](#):

```
1 // @version=5
2 indicator("", "", true)
3 // Declare `pHi` and initialize it on the first bar only.
4 var float pHi = na
5 // Reassign a value to `pHi`
6 pHi := nz(ta.pivothigh(5, 5), pHi)
7 plot(pHi)
```

Note that:

- We declare pHi with this code: `var float pHi = na.` The `var` keyword tells Pine Script™ that we only want that variable initialized with `na` on the dataset's first bar. The `float` keyword tells the compiler we are declaring a variable of type “float”. This is necessary because, contrary to most cases, the compiler cannot automatically determine the type of the value on the right side of the `=` sign.
- While the variable declaration will only be executed on the first bar because it uses `var`, the `pHi := nz(ta.pivothigh(5, 5), pHi)` line will be executed on all the chart's bars. On each bar, it evaluates if the `pivothigh()` call returns `na` because that is what the function does when it hasn't found a new pivot. The `nz()` function is the one doing the “checking for `na`” part. When its first argument (`ta.pivothigh(5, 5)`) is `na`, it returns the second argument (`pHi`) instead of the first. When `pivothigh()` returns the price point of a newly found pivot, that value is assigned to `pHi`. When it returns `na` because no new pivot was found, we assign the previous value of `pHi` to itself, in effect preserving its previous value.

The output of our script looks like this:



Note that:

- The line preserves its previous value until a new pivot is found.
- Pivots are detected five bars after the pivot actually occurs because our `ta.pivothigh(5, 5)` call says that we require five lower highs on both sides of a high point for it to be detected as a pivot.

See the [Variable reassignment](#) section for more information on how to reassign values to variables.



## 3.6 Variable declarations

- *Introduction*
- *Variable reassignment*
- *Declaration modes*

### 3.6.1 Introduction

Variables are *identifiers* that hold values. They must be *declared* in your code before you use them. The syntax of variable declarations is:

```
[<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
```

or

```
<tuple_declaration> = <function_call> | <structure>
```

where:

- | means “or”, and parts enclosed in square brackets ([]) can appear zero or one time.
- <declaration\_mode> is the variable’s *declaration mode*. It can be `var` or `varip`, or nothing.
- <type> is optional, as in almost all Pine Script™ variable declarations (see *types*).
- <identifier> is the variable’s *name*.
- <expression> can be a literal, a variable, an expression or a function call.
- <structure> can be an `if`, `for`, `while` or `switch` *structure*.
- <tuple\_declaration> is a comma-separated list of variable names enclosed in square brackets ([]), e.g., `[ma, upperBand, lowerBand]`.

These are all valid variable declarations. The last one requires four lines:

```
BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)
var barRange = float(na)
var firstBarOpen = open
varip float lastClose = na
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red
```

---

**Note:** The above statements all contain the = assignment operator because they are **variable declarations**. When you see similar lines using the := reassignment operator, the code is **reassigning** a value to a variable that was **already**

**declared.** Those are **variable reassignments**. Be sure you understand the distinction as this is a common stumbling block for newcomers to Pine Script™. See the next [Variable reassignment](#) section for details.

The formal syntax of a variable declaration is:

```
<variable_declarator>
  [<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
  |
  <tuple_declarator> = <function_call> | <structure>

<declaration_mode>
  var | varip

<type>
  int | float | bool | color | string | line | linefill | label | box | table |_
  ↵array<type> | matrix<type> | UDF
```

### Initialization with `na`

In most cases, an explicit type declaration is redundant because type is automatically inferred from the value on the right of the = at compile time, so the decision to use them is often a matter of preference. For example:

```
baseLine0 = na          // compile time error!
float baseLine1 = na    // OK
baseLine2 = float(na)   // OK
```

In the first line of the example, the compiler cannot determine the type of the `baseLine0` variable because `na` is a generic value of no particular type. The declaration of the `baseLine1` variable is correct because its `float` type is declared explicitly. The declaration of the `baseLine2` variable is also correct because its type can be derived from the expression `float(na)`, which is an explicit cast of the `na` value to the `float` type. The declarations of `baseLine1` and `baseLine2` are equivalent.

### Tuple declarations

Function calls or structures are allowed to return multiple values. When we call them and want to store the values they return, a *tuple declaration* must be used, which is a comma-separated set of one or more values enclosed in brackets. This allows us to declare multiple variables simultaneously. As an example, the `ta.bb()` built-in function for Bollinger bands returns three values:

```
[bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 4)
```

## 3.6.2 Variable reassignment

A variable reassignment is done using the `:=` reassignment operator. It can only be done after a variable has been first declared and given an initial value. Reassigning a new value to a variable is often necessary in calculations, and it is always necessary when a variable from the global scope must be assigned a new value from within a structure's local block, e.g.:

```
1 // @version=5
2 indicator("", "", true)
3 sensitivityInput = input.int(2, "Sensitivity", minval = 1, tooltip = "Higher values_
  ↵make color changes less sensitive.")
4 ma = ta.sma(close, 20)
```

(continues on next page)

(continued from previous page)

```

5 maUp = ta.rising(ma, sensitivityInput)
6 maDn = ta.falling(ma, sensitivityInput)
7
8 // On first bar only, initialize color to gray
9 var maColor = color.gray
10 if maUp
11     // MA has risen for two bars in a row; make it lime.
12     maColor := color.lime
13 else if maDn
14     // MA has fallen for two bars in a row; make it fuchsia.
15     maColor := color.fuchsia
16
17 plot(ma, "MA", maColor, 2)

```

Note that:

- We initialize `maColor` on the first bar only, so it preserves its value across bars.
- On every bar, the `if` statement checks if the MA has been rising or falling for the user-specified number of bars (the default is 2). When that happens, the value of `maColor` must be reassigned a new value from within the `if` local blocks. To do this, we use the `:=` reassignment operator.
- If we did not use the `:=` reassignment operator, the effect would be to initialize a new `maColor` local variable which would have the same name as that of the global scope, but actually be a very confusing independent entity that would persist only for the length of the local block, and then disappear without a trace.

All user-defined variables in Pine Script™ are *mutable*, which means their value can be changed using the `:=` reassignment operator. Assigning a new value to a variable may change its *type qualifier* (see the page on Pine Script™'s *type system* for more information). A variable can be assigned a new value as many times as needed during the script's execution on one bar, so a script can contain any number of reassignments of one variable. A variable's *declaration mode* determines how new values assigned to a variable will be saved.

### 3.6.3 Declaration modes

Understanding the impact that declaration modes have on the behavior of variables requires prior knowledge of Pine Script™'s *execution model*.

When you declare a variable, if a declaration mode is specified, it must come first. Three modes can be used:

- “On each bar”, when none is specified
- `var`
- `varip`

#### On each bar

When no explicit declaration mode is specified, i.e. no `var` or `varip` keyword is used, the variable is declared and initialized on each bar, e.g., the following declarations from our first set of examples in this page's introduction:

```

BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)

```

(continues on next page)

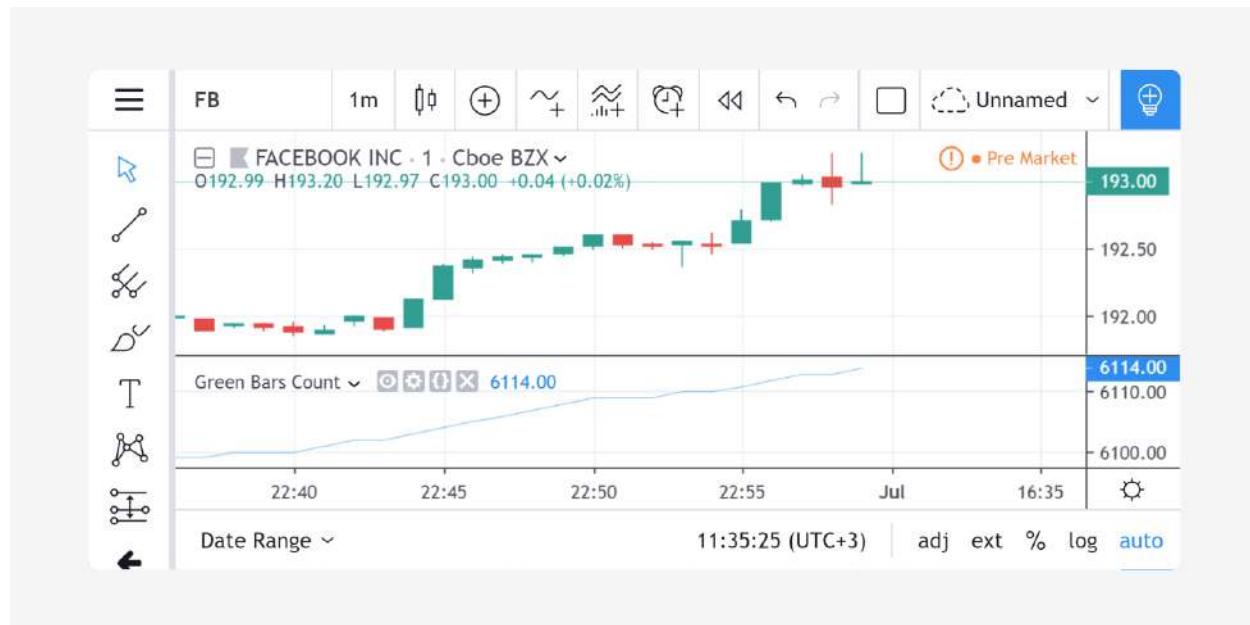
(continued from previous page)

```
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red
```

**'var'**

When the `var` keyword is used, the variable is only initialized once, on the first bar if the declaration is in the global scope, or the first time the local block is executed if the declaration is inside a local block. After that, it will preserve its last value on successive bars, until we reassign a new value to it. This behavior is very useful in many cases where a variable's value must persist through the iterations of a script across successive bars. For example, suppose we'd like to count the number of green bars on the chart:

```
1 // @version=5
2 indicator("Green Bars Count")
3 var count = 0
4 isGreen = close >= open
5 if isGreen
6     count := count + 1
7 plot(count)
```



Without the `var` modifier, variable `count` would be reset to zero (thus losing its value) every time a new bar update triggered a script recalculation.

Declaring variables on the first bar only is often useful to manage drawings more efficiently. Suppose we want to extend the last bar's `close` line to the right of the right chart. We could write:

```
1 // @version=5
2 indicator("Inefficient version", "", true)
3 closeLine = line.new(bar_index - 1, close, bar_index, close, extend = extend.right,
4 width = 3)
4 line.delete(closeLine[1])
```

but this is inefficient because we are creating and deleting the line on each historical bar and on each update in the realtime bar. It is more efficient to use:

```

1 // @version=5
2 indicator("Efficient version", "", true)
3 var closeLine = line.new(bar_index - 1, close, bar_index, close, extend = extend,
4     ↪ right, width = 3)
5 if barstate.islast
6     line.set_xy1(closeLine, bar_index - 1, close)
    line.set_xy2(closeLine, bar_index, close)

```

Note that:

- We initialize `closeLine` on the first bar only, using the `var` declaration mode
- We restrict the execution of the rest of our code to the chart's last bar by enclosing our code that updates the line in an `if barstate.islast` structure.

There is a very slight penalty performance for using the `var` declaration mode. For that reason, when declaring constants, it is preferable not to use `var` if performance is a concern, unless the initialization involves calculations that take longer than the maintenance penalty, e.g., functions with complex code or string manipulations.

### `'varip'`

Understanding the behavior of variables using the `varip` declaration mode requires prior knowledge of Pine Script™'s *execution model* and *bar states*.

The `varip` keyword can be used to declare variables that escape the *rollback process*, which is explained in the page on Pine Script™'s *execution model*.

Whereas scripts only execute once at the close of historical bars, when a script is running in realtime, it executes every time the chart's feed detects a price or volume update. At every realtime update, Pine Script™'s runtime normally resets the values of a script's variables to their last committed value, i.e., the value they held when the previous bar closed. This is generally handy, as each realtime script execution starts from a known state, which simplifies script logic.

Sometimes, however, script logic requires code to be able to save variable values **between different executions** in the realtime bar. Declaring variables with `varip` makes that possible. The "ip" in `varip` stands for *intrabar persist*.

Let's look at the following code, which does not use `varip`:

```

1 // @version=5
2 indicator("")
3 int updateNo = na
4 if barstate.isnew
5     updateNo := 1
6 else
7     updateNo := updateNo + 1
8
9 plot(updateNo, style = plot.style_circles)

```

On historical bars, `barstate.isnew` is always true, so the plot shows a value of "1" because the `else` part of the `if` structure is never executed. On realtime bars, `barstate.isnew` is only `true` when the script first executes on the bar's "open". The plot will then briefly display "1" until subsequent executions occur. On the next executions during the realtime bar, the second branch of the `if` statement is executed because `barstate.isnew` is no longer true. Since `updateNo` is initialized to `na` at each execution, the `updateNo + 1` expression yields `na`, so nothing is plotted on further realtime executions of the script.

If we now use `varip` to declare the `updateNo` variable, the script behaves very differently:

```

1 //@version=5
2 indicator("")
3 varip int updateNo = na
4 if barstate.isnew
5     updateNo := 1
6 else
7     updateNo := updateNo + 1
8
9 plot(updateNo, style = plot.style_circles)

```

The difference now is that `updateNo` tracks the number of realtime updates that occur on each realtime bar. This can happen because the `varip` declaration allows the value of `updateNo` to be preserved between realtime updates; it is no longer rolled back at each realtime execution of the script. The test on `barstate.isnew` allows us to reset the update count when a new realtime bar comes in.

Because `varip` only affects the behavior of your code in the realtime bar, it follows that backtest results on strategies designed using logic based on `varip` variables will not be able to reproduce that behavior on historical bars, which will invalidate test results on them. This also entails that plots on historical bars will not be able to reproduce the script's behavior in realtime.



## 3.7 Conditional structures

- *Introduction*
- *`if` structure*
- *`switch` structure*
- *Matching local block type requirement*

### 3.7.1 Introduction

The conditional structures in Pine Script™ are `if` and `switch`. They can be used:

- For their side effects, i.e., when they don't return a value but do things, like reassign values to variables or call functions.
- To return a value or a tuple which can then be assigned to one (or more, in the case of tuples) variable.

Conditional structures, like the `for` and `while` structures, can be embedded; you can use an `if` or `switch` inside another structure.

Some Pine Script™ built-in functions cannot be called from within the local blocks of conditional structures. They are: `alertcondition()`, `barcolor()`, `fill()`, `hline()`, `indicator()`, `library()`, `plot()`, `plotbar()`, `plotcandle()`, `plotchar()`, `plotshape()`,

`strategy()`. This does not entail their functionality cannot be controlled by conditions evaluated by your script — only that it cannot be done by including them in conditional structures. Note that while `input *.` () function calls are allowed in local blocks, their functionality is the same as if they were in the script's global scope.

The local blocks in conditional structures must be indented by four spaces or a tab.

### 3.7.2 `if` structure

#### 'if' used for its side effects

An `if` structure used for its side effects has the following syntax:

```
if <expression>
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

where:

- Parts enclosed in square brackets ([ ]) can appear zero or one time, and those enclosed in curly braces ({} ) can appear zero or more times.
- `<expression>` must be of “bool” type or be auto-castable to that type, which is only possible for “int” or “float” values (see the [Type system](#) page).
- `<local_block>` consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- There can be zero or more `else if` clauses.
- There can be zero or one `else` clause.

When the `<expression>` following the `if` evaluates to `true`, the first local block is executed, the `if` structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When the `<expression>` following the `if` evaluates to `false`, the successive `else if` clauses are evaluated, if there are any. When the `<expression>` of one evaluates to `true`, its local block is executed, the `if` structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When no `<expression>` has evaluated to `true` and an `else` clause exists, its local block is executed, the `if` structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When no `<expression>` has evaluated to `true` and no `else` clause exists, `na` is returned.

Using `if` structures for their side effects can be useful to manage the order flow in strategies, for example. While the same functionality can often be achieved using the `when` parameter in `strategy.*()` calls, code using `if` structures is easier to read:

```
if (ta.crossover(source, lower))
    strategy.entry("BBandLE", strategy.long, stop=lower,
                  oca_name="BollingerBands",
                  oca_type=strategy.oca.cancel, comment="BBandLE")
else
    strategy.cancel(id="BBandLE")
```

Restricting the execution of your code to specific bars can be done using `if` structures, as we do here to restrict updates to our label to the chart's last bar:

```

1 //@version=5
2 indicator("", "", true)
3 var ourLabel = label.new(bar_index, na, na, color = color(na), textcolor = color.
4   ↪orange)
5 if barstate.islast
6   label.set_xy(ourLabel, bar_index + 2, h12[1])
    label.set_text(ourLabel, str.tostring(bar_index + 1, "# bars in chart"))

```

Note that:

- We initialize the `ourLabel` variable on the script's first bar only, as we use the `var` declaration mode. The value used to initialize the variable is provided by the `label.new()` function call, which returns a label ID pointing to the label it creates. We use that call to set the label's properties because once set, they will persist until we change them.
- What happens next is that on each successive bar the Pine Script™ runtime will skip the initialization of `ourLabel`, and the `if` structure's condition (`barstate.islast`) is evaluated. It returns `false` on all bars until the last one, so the script does nothing on most historical bars after bar zero.
- On the last bar, `barstate.islast` becomes true and the structure's local block executes, modifying on each chart update the properties of our label, which displays the number of bars in the dataset.
- We want to display the label's text without a background, so we make the label's background `na` in the `label.new()` function call, and we use `h12[1]` for the label's `y` position because we don't want it to move all the time. By using the average of the **previous** bar's `high` and `low` values, the label doesn't move until the moment when the next realtime bar opens.
- We use `bar_index + 2` in our `label.set_xy()` call to offset the label to the right by two bars.

### `'if' used to return a value`

An `if` structure used to return one or more values has the following syntax:

```

[<declaration_mode>] [<type>] <identifier> = if <expression>
  <local_block>
{else if <expression>
  <local_block>}
[else
  <local_block>]

```

where:

- Parts enclosed in square brackets ([ ]) can appear zero or one time, and those enclosed in curly braces ({} ) can appear zero or more times.
- `<declaration_mode>` is the variable's *declaration mode*
- `<type>` is optional, as in almost all Pine Script™ variable declarations (see [types](#))
- `<identifier>` is the variable's *name*
- `<expression>` can be a literal, a variable, an expression or a function call.
- `<local_block>` consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the `<local_block>`, or `na` if no local block is executed.

This is an example:

```

1 //@version=5
2 indicator("", "", true)
3 string barState = if barstate.islastconfirmedhistory
4     "islastconfirmedhistory"
5 else if barstate.isnew
6     "isnew"
7 else if barstate.isrealtime
8     "isrealtime"
9 else
10    "other"
11
12 f_print(_text) =>
13     var table _t = table.new(position.middle_right, 1, 1)
14     table.cell(_t, 0, 0, _text, bgcolor = color.yellow)
15 f_print(barState)

```

It is possible to omit the `else` block. In this case, if the condition is false, an *empty* value (`na`, `false`, or `" "`) will be assigned to the `var_declarationX` variable.

This is an example showing how `na` is returned when no local block is executed. If `close > open` is `false` in here, `na` is returned:

```
x = if close > open
    close
```

Scripts can contain `if` structures with nested `if` and other conditional structures. For example:

```

if condition1
    if condition2
        if condition3
            expression

```

However, nesting these structures is not recommended from a performance perspective. When possible, it is typically more optimal to compose a single `if` statement with multiple logical operators rather than several nested `if` blocks:

```
if condition1 and condition2 and condition3
    expression
```

### 3.7.3 `switch` structure

The `switch` structure exists in two forms. One switches on the different values of a key expression:

```

[[<declaration_mode>] [<type>] <identifier> = ]switch <expression>
    {<expression> => <local_block>}
    => <local_block>

```

The other form does not use an expression as a key; it switches on the evaluation of different expressions:

```

[[<declaration_mode>] [<type>] <identifier> = ]switch
    {<expression> => <local_block>}
    => <local_block>

```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{ }`) can appear zero or more times.

- <declaration\_mode> is the variable's *declaration mode*
- <type> is optional, as in almost all Pine Script™ variable declarations (see [types](#))
- <identifier> is the variable's *name*
- <expression> can be a literal, a variable, an expression or a function call.
- <local\_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the <local\_block>, or `na` if no local block is executed.
- The => <local\_block> at the end allows you to specify a return value which acts as a default to be used when no other case in the structure is executed.

Only one local block of a `switch` structure is executed. It is thus a *structured switch* that doesn't *fall through* cases. Consequently, `break` statements are unnecessary.

Both forms are allowed as the value used to initialize a variable.

As with the `if` structure, if no local block is executed, `na` is returned.

### `'switch` with an expression`

Let's look at an example of a `switch` using an expression:

```

1 // @version=5
2 indicator("Switch using an expression", "", true)
3
4 string maType = input.string("EMA", "MA type", options = ["EMA", "SMA", "RMA", "WMA"])
5 int maLength = input.int(10, "MA length", minval = 2)
6
7 float ma = switch maType
8     "EMA" => ta.ema(close, maLength)
9     "SMA" => ta.sma(close, maLength)
10    "RMA" => ta.rma(close, maLength)
11    "WMA" => ta.wma(close, maLength)
12    =>
13        runtime.error("No matching MA type found.")
14        float(na)
15
16 plot(ma)

```

Note that:

- The expression we are switching on is the variable `maType`, which is of “`input int`” type (see here for an explanation of what the “`input`” qualifier is). Since it cannot change during the execution of the script, this guarantees that whichever MA type the user selects will be executing on each bar, which is a requirement for functions like `ta.ema()` which require a “simple int” argument for their `length` parameter.
- If no matching value is found for `maType`, the `switch` executes the last local block introduced by =>, which acts as a catch-all. We generate a runtime error in that block. We also end it with `float(na)` so the local block returns a value whose type is compatible with that of the other local blocks in the structure, to avoid a compilation error.

## `switch` without an expression

This is an example of a `switch` structure which does not use an expression:

```

1 // @version=5
2 strategy("Switch without an expression", "", true)
3
4 bool longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
5 bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))
6
7 switch
8     longCondition => strategy.entry("Long ID", strategy.long)
9     shortCondition => strategy.entry("Short ID", strategy.short)

```

Note that:

- We are using the `switch` to select the appropriate strategy order to emit, depending on whether the `longCondition` or `shortCondition` “bool” variables are true.
- The building conditions of `longCondition` and `shortCondition` are exclusive. While they can both be `false` simultaneously, they cannot be `true` at the same time. The fact that only **one** local block of the `switch` structure is ever executed is thus not an issue for us.
- We evaluate the calls to `ta.crossover()` and `ta.crossunder()` **prior** to entry in the `switch` structure. Not doing so, as in the following example, would prevent the functions to be executed on each bar, which would result in a compiler warning and erratic behavior:

```

1 // @version=5
2 strategy("Switch without an expression", "", true)
3
4 switch
5     // Compiler warning! Will not calculate correctly!
6     ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Long ID", ↵
7     ↵strategy.long)
8     ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Short ID", ↵
9     ↵strategy.short)

```

### 3.7.4 Matching local block type requirement

When multiple local blocks are used in structures, the type of the return value of all its local blocks must match. This applies only if the structure is used to assign a value to a variable in a declaration, because a variable can only have one type, and if the statement returns two incompatible types in its branches, the variable type cannot be properly determined. If the structure is not assigned anywhere, its branches can return different values.

This code compiles fine because `close` and `open` are both of the `float` type:

```

x = if close > open
    close
else
    open

```

This code does not compile because the first local block returns a `float` value, while the second one returns a `string`, and the result of the `if`-statement is assigned to the `x` variable:

```

// Compilation error!
x = if close > open

```

(continues on next page)

(continued from previous page)

```

close
else
  "open"

```



## 3.8 Loops

- *Introduction*
- `for`
- `while`

### 3.8.1 Introduction

#### When loops are not needed

Pine Script™'s runtime and its built-in functions make loops unnecessary in many situations. Budding Pine Script™ programmers not yet familiar with the Pine Script™ runtime and built-ins who want to calculate the average of the last 10 `close` values will often write code such as:

```

1 // @version=5
2 indicator("Inefficient MA", "", true)
3 MA_LENGTH = 10
4 sumOfCloses = 0.0
5 for offset = 0 to MA_LENGTH - 1
6   sumOfCloses := sumOfCloses + close[offset]
7 inefficientMA = sumOfCloses / MA_LENGTH
8 plot(inefficientMA)

```

A `for` loop is unnecessary and inefficient to accomplish tasks like this in Pine. This is how it should be done. This code is shorter *and* will run much faster because it does not use a loop and uses the `ta.sma()` built-in function to accomplish the task:

```

1 // @version=5
2 indicator("Efficient MA", "", true)
3 thePineMA = ta.sma(close, 10)
4 plot(thePineMA)

```

Counting the occurrences of a condition in the last bars is also a task which beginning Pine Script™ programmers often think must be done with a loop. To count the number of up bars in the last 10 bars, they will use:

```

1 //@version=5
2 indicator("Inefficient sum")
3 MA_LENGTH = 10
4 upBars = 0.0
5 for offset = 0 to MA_LENGTH - 1
6     if close[offset] > open[offset]
7         upBars := upBars + 1
8 plot (upBars)

```

The efficient way to write this in Pine (for the programmer because it saves time, to achieve the fastest-loading charts, and to share our common resources most equitably), is to use the [math.sum\(\)](#) built-in function to accomplish the task:

```

1 //@version=5
2 indicator("Efficient sum")
3 upBars = math.sum(close > open ? 1 : 0, 10)
4 plot (upBars)

```

What's happening in there is:

- We use the ?: ternary operator to build an expression that yields 1 on up bars and 0 on other bars.
- We use the [math.sum\(\)](#) built-in function to keep a running sum of that value for the last 10 bars.

### When loops are necessary

Loops exist for good reason because even in Pine Script™, they are necessary in some cases. These cases typically include:

- The manipulation of collections ([arrays](#), [matrices](#), and [maps](#)).
- Looking back in history to analyze bars using a reference value that can only be known on the current bar, e.g., to find how many past highs are higher than the [high](#) of the current bar. Since the current bar's [high](#) is only known on the bar the script is running on, a loop is necessary to go back in time and analyze past bars.
- Performing calculations on past bars that cannot be accomplished using built-in functions.

### 3.8.2 `for`

The `for` structure allows the repetitive execution of statements using a counter. Its syntax is:

```

[ [<declaration_mode>] [<type>] <identifier> = ]for <identifier> = <expression> to
    ↵<expression>[ by <expression>]
        <local_block_loop>

```

where:

- Parts enclosed in square brackets ([ ]) can appear zero or one time, and those enclosed in curly braces ({} ) can appear zero or more times.
- `<declaration_mode>` is the variable's [declaration mode](#)
- `<type>` is optional, as in almost all Pine Script™ variable declarations (see [types](#))
- `<identifier>` is a variable's [name](#)
- `<expression>` can be a literal, a variable, an expression or a function call.

- <local\_block\_loop> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab. It can contain the `break` statement to exit the loop, or the `continue` statement to exit the current iteration and continue on with the next.
- The value assigned to the variable is the return value of the <local\_block\_loop>, i.e., the last value calculated on the loop's last iteration, or `na` if the loop is not executed.
- The identifier in `for <identifier>` is the loop's counter *initial value*.
- The expression in `= <expression>` is the *start value* of the counter.
- The expression in `to <expression>` is the *end value* of the counter. **It is only evaluated upon entry in the loop.**
- The expression in `by <expression>` is optional. It is the step by which the loop counter is increased or decreased on each iteration of the loop. Its default value is 1 when `start value < end value`. It is -1 when `start value > end value`. The step (+1 or -1) used as the default is determined by the start and end values.

This example uses a `for` statement to look back a user-defined amount of bars to determine how many bars have a `high` that is higher or lower than the `high` of the last bar on the chart. A `for` loop is necessary here, since the script only has access to the reference value on the chart's last bar. Pine Script™'s runtime cannot, here, be used to calculate on the fly, as the script is executing bar to bar:

```

1 //@version=5
2 indicator(`for` loop")
3 lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
4 higherBars = 0
5 lowerBars = 0
6 if barstate.islast
7     var label lbl = label.new(na, na, "", style = label.style_label_left)
8     for i = 1 to lookbackInput
9         if high[i] > high
10            higherBars += 1
11        else if high[i] < high
12            lowerBars += 1
13    label.set_xy(lbl, bar_index, high)
14    label.set_text(lbl, str.tostring(higherBars, "# higher bars\n") + str.
→tostring(lowerBars, "# lower bars"))

```

This example uses a loop in its `checkLinesForBreaches()` function to go through an array of pivot lines and delete them when price crosses them. A loop is necessary here because all the lines in each of the `hiPivotLines` and `loPivotLines` arrays must be checked on each bar, and there is no built-in that can do this for us:

```

1 //@version=5
2 MAX_LINES_COUNT = 100
3 indicator("Pivot line breaches", "", true, max_lines_count = MAX_LINES_COUNT)
4
5 color hiPivotColorInput = input(color.new(color.lime, 0), "High pivots")
6 color loPivotColorInput = input(color.new(color.fuchsia, 0), "Low pivots")
7 int pivotLegsInput = input.int(5, "Pivot legs")
8 int qtyOfPivotsInput = input.int(50, "Quantity of last pivots to remember",_
→minval = 0, maxval = MAX_LINES_COUNT / 2)
9 int maxLineLengthInput = input.int(400, "Maximum line length in bars", minval = 2)
10
11 // ----- Queues a new element in an array and de-queues its first element.
12 qDq(array, qtyOfElements, arrayElement) =>
13     array.push(array, arrayElement)
14     if array.size(array) > qtyOfElements

```

(continues on next page)

(continued from previous page)

```

15     // Only dequeue if array has reached capacity.
16     array.shift(array)
17
18 // ----- Loop through an array of lines, extending those that price has not crossed
19 // and deleting those crossed.
20 checkLinesForBreaches(arrayOfLines) =>
21     int qtyOfLines = array.size(arrayOfLines)
22     // Don't loop in case there are no lines to check because "to" value will be `na`_
23     // then`.
24     for lineNo = 0 to (qtyOfLines > 0 ? qtyOfLines - 1 : na)
25         // Need to check that array size still warrants a loop because we may have_
26         // deleted array elements in the loop.
27         if lineNo < array.size(arrayOfLines)
28             line currentLine      = array.get(arrayOfLines, lineNo)
29             float lineLevel       = line.get_price(currentLine, bar_index)
30             bool lineWasCrossed = math.sign(close[1] - lineLevel) != math.sign(close_
31             // lineLevel)
32             bool lineIsTooLong   = bar_index - line.get_x1(currentLine) >_
33             maxLineLengthInput
34             if lineWasCrossed or lineIsTooLong
35                 // Line stays on the chart but will no longer be extend on further_
36             // bars.
37                 array.remove(arrayOfLines, lineNo)
38                 // Force type of both local blocks to same type.
39                 int(na)
40             else
41                 line.set_x2(currentLine, bar_index)
42                 int(na)
43
44 // Arrays of lines containing non-crossed pivot lines.
45 var array<line> hiPivotLines = array.new_line(qtyOfPivotsInput)
46 var array<line> loPivotLines = array.new_line(qtyOfPivotsInput)
47
48 // Detect new pivots.
49 float hiPivot = ta.pivothigh(pivotLegsInput, pivotLegsInput)
50 float loPivot = ta.pivotlow(pivotLegsInput, pivotLegsInput)
51
52 // Create new lines on new pivots.
53 if not na(hiPivot)
54     line.newLine = line.new(bar_index[pivotLegsInput], hiPivot, bar_index, hiPivot,_
55     //color = hiPivotColorInput)
56     line.delete(qDq(hiPivotLines, qtyOfPivotsInput, newLine))
57 else if not na(loPivot)
58     line.newLine = line.new(bar_index[pivotLegsInput], loPivot, bar_index, loPivot,_
59     //color = loPivotColorInput)
60     line.delete(qDq(loPivotLines, qtyOfPivotsInput, newLine))
61
62 // Extend lines if they haven't been crossed by price.
63 checkLinesForBreaches(hiPivotLines)
64 checkLinesForBreaches(loPivotLines)

```

### 3.8.3 `while`

The `while` structure allows the repetitive execution of statements until a condition is false. Its syntax is:

```
[ [<declaration_mode>] [<type>] <identifier> = ]while <expression>
    <local_block_loop>
```

where:

- Parts enclosed in square brackets ([]) can appear zero or one time.
- `<declaration_mode>` is the variable's *declaration mode*
- `<type>` is optional, as in almost all Pine Script™ variable declarations (see [types](#))
- `<identifier>` is a variable's *name*
- `<expression>` can be a literal, a variable, an expression or a function call. It is evaluated at each iteration of the loop. When it evaluates to `true`, the loop executes. When it evaluates to `false` the loop stops. Note that evaluation of the expression is done before each iteration only. Changes to the expression's value inside the loop will only have an impact on the next iteration.
- `<local_block_loop>` consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab. It can contain the `break` statement to exit the loop, or the `continue` statement to exit the current iteration and continue on with the next.
- The value assigned to the `<identifier>` variable is the return value of the `<local_block_loop>`, i.e., the last value calculated on the loop's last iteration, or `na` if the loop is not executed.

This is the first code example of the `for` section written using a `while` structure instead of a `for` one:

```

1 // @version=5
2 indicator(`for` loop")
3 lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
4 higherBars = 0
5 lowerBars = 0
6 if barstate.islast
7     var label lbl = label.new(na, na, "", style = label.style_label_left)
8     // Initialize the loop counter to its start value.
9     i = 1
10    // Loop until the `i` counter's value is <= the `lookbackInput` value.
11    while i <= lookbackInput
12        if high[i] > high
13            higherBars += 1
14        else if high[i] < high
15            lowerBars += 1
16            // Counter must be managed "manually".
17            i += 1
18        label.set_xy(lbl, bar_index, high)
19        label.set_text(lbl, str.tostring(higherBars, "# higher bars\n") + str.
→tostring(lowerBars, "# lower bars"))

```

Note that:

- The `i` counter must be incremented by one explicitly inside the `while`'s local block.
- We use the `+=` operator to add one to the counter. `lowerBars += 1` is equivalent to `lowerBars := lowerBars + 1`.

Let's calculate the factorial function using a `while` structure:

```
1 // @version=5
2 indicator("")
3 int n = input.int(10, "Factorial of", minval=0)
4
5 factorial(int val = na) =>
6     int counter = val
7     int fact = 1
8     result = while counter > 0
9         fact := fact * counter
10        counter := counter - 1
11    fact
12
13 // Only evaluate the function on the first bar.
14 var answer = factorial(n)
15 plot(answer)
```

Note that:

- We use `input.int()` for our input because we need to specify a `minval` value to protect our code. While `input()` also supports the input of “int” type values, it does not support the `minval` parameter.
- We have packaged our script’s functionality in a `factorial()` function which accepts as an argument the value whose factorial it must calculate. We have used `int val = na` to declare our function’s parameter, which says that if the function is called without an argument, as in `factorial()`, then the `val` parameter will initialize to `na`, which will prevent the execution of the `while` loop because its `counter > 0` expression will return `na`. The `while` structure will thus initialize the `result` variable to `na`. In turn, because the initialization of `result` is the return value of the our function’s local block, the function will return `na`.
- Note the last line of the `while`’s local block: `fact`. It is the local block’s return value, so the value it had on the `while` structure’s last iteration.
- Our initialization of `result` is not required; we do it for readability. We could just as well have used:

```
while counter > 0
    fact := fact * counter
    counter := counter - 1
    fact
```



## 3.9 Type system

- *Introduction*
- *Qualifiers*
- *Types*
- *`na` value*
- *Type templates*
- *Type casting*
- *Tuples*

### 3.9.1 Introduction

The Pine Script™ type system determines the compatibility of a script's values with various functions and operations. While it's possible to write simple scripts without knowing anything about the type system, a reasonable understanding of it is necessary to achieve any degree of proficiency with the language, and an in-depth knowledge of its subtleties will allow you to harness its full potential.

Pine Script™ uses *types* to classify all values, and it uses *qualifiers* to determine whether values are constant, established on the first script iteration, or dynamic throughout a script's execution. This system applies to all Pine values, including those from literals, variables, expressions, function returns, and function arguments.

The type system closely intertwines with Pine's *execution model* and *time series* concepts. Understanding all three is essential for making the most of the power of Pine Script™.

---

**Note:** For the sake of brevity, we often use “type” to refer to a “qualified type”.

---

### 3.9.2 Qualifiers

Pine Script™ *qualifiers* identify when a value is accessible in the script's execution:

- Values qualified as *const* are established at compile time (i.e., when saving the script in the Pine Editor or adding it to the chart).
- Values qualified as *input* are available at input time (i.e., when changing values in the script's “Settings/Inputs” tab).
- Values qualified as *simple* are established at bar zero (the first bar of the script's execution).
- Values qualified as *series* can change throughout the script's execution.

Pine Script™ bases the dominance of type qualifiers on the following hierarchy: **const < input < simple < series**, where “const” is the *weakest* qualifier and “series” is the *strongest*. The qualifier hierarchy translates into this rule: whenever a variable, function, or operation is compatible with a specific qualified type, values with *weaker* qualifiers are also allowed.

Scripts always qualify their expressions' returned values based on the dominant qualifier in their calculations. For example, evaluating an expression that involves “const” and “series” values will return a value qualified as “series”. Furthermore, scripts cannot change a value's qualifier to one that's lower on the hierarchy. If a value acquires a stronger qualifier (e.g., a value initially inferred as “simple” becomes “series” later in the script's execution), that state is irreversible.

Note that only values qualified as “series” can change throughout the execution of a script, which include those from various built-ins, such as `close` and `volume`, as well as the results of any operations that involve “series” values. Values qualified as “const”, “input”, or “simple” are consistent throughout a script’s execution.

### **const**

Values qualified as “const” are established at *compile time*, before the script starts its execution. Compilation initially occurs when saving a script in the Pine Editor, which does not require it to run on a chart. Values with the “const” qualifier never change between script iterations, not even on the initial bar of its execution.

Scripts can qualify values as “const” by using a *literal* value or calculating values from expressions that only use literal values or other variables qualified as “const”.

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 color*: #FF55C6, #FF55C6ff
- *literal string*: "A text literal", "Embedded single quotes 'text'", 'Embedded double quotes "text"'

Users can explicitly define variables and parameters that only accept “const” values by including the `const` keyword in their declaration.

Our [Style guide](#) recommends using uppercase SNAKE\_CASE to name “const” variables for readability. While it is not a requirement, one can also use the `var` keyword when declaring “const” variables so the script only initializes them on the *first bar* of the dataset. See [this section](#) of our User Manual for more information.

Below is an example that uses “const” values within `indicator()` and `plot()` functions, which both require a value of the “const string” qualified type as their `title` argument:

```
1 // @version=5
2
3 // The following global variables are all of the "const string" qualified type:
4
5 //@variable The title of the indicator.
6 INDICATOR_TITLE = "const demo"
7 //@variable The title of the first plot.
8 var PLOT1_TITLE = "High"
9 //@variable The title of the second plot.
10 const string PLOT2_TITLE = "Low"
11 //@variable The title of the third plot.
12 PLOT3_TITLE = "Midpoint between " + PLOT1_TITLE + " and " + PLOT2_TITLE
13
14 indicator(INDICATOR_TITLE, overlay = true)
15
16 plot(high, PLOT1_TITLE)
17 plot(low, PLOT2_TITLE)
18 plot(hl2, PLOT3_TITLE)
```

The following example will raise a compilation error since it uses `syminfo.ticker`, which returns a “simple” value because it depends on chart information that’s only accessible once the script starts its execution:

```

1 // @version=5
2
3 //@variable The title in the `indicator()` call.
4 var NAME = "My indicator for " + syminfo.ticker
5
6 indicator(NAME, "", true) // Causes an error because `NAME` is qualified as a "simple_
7 plot(close)

```

## input

Values qualified as “input” are established after initialization via the `input.*()` functions. These functions produce values that users can modify within the “Inputs” tab of the script’s settings. When one changes any of the values in this tab, the script re-executes from the beginning of the chart’s history to ensure its input values are consistent throughout its execution.

---

**Note:** The `input.source()` function is an exception in the `input.*()` namespace, as it returns “series” qualified values rather than “input” since built-in variables such as `open`, `close`, etc., as well as the values from another script’s plots, are qualified as “series”.

---

The following script plots the value of a `sourceInput` from the `symbolInput` and `timeframeInput` context. The `request.security()` call is valid in this script since its `symbol` and `timeframe` parameters allow “simple string” arguments, meaning they can also accept “input string” values because the “input” qualifier is *lower* on the hierarchy:

```

1 // @version=5
2 indicator("input demo", overlay = true)
3
4 //@variable The symbol to request data from. Qualified as "input string".
5 symbolInput = input.symbol("AAPL", "Symbol")
6 //@variable The timeframe of the data request. Qualified as "input string".
7 timeframeInput = input.timeframe("D", "Timeframe")
8 //@variable The source of the calculation. Qualified as "series float".
9 sourceInput = input.source(close, "Source")
10
11 //@variable The `sourceInput` value from the requested context. Qualified as "series_
12 //float".
13 requestedSource = request.security(symbolInput, timeframeInput, sourceInput)
14 plot(requestedSource)

```

## simple

Values qualified as “simple” are available only once the script begins execution on the *first* chart bar of its history, and they remain consistent during the script’s execution.

Users can explicitly define variables and parameters that accept “simple” values by including the `simple` keyword in their declaration.

Many built-in variables return “simple” qualified values because they depend on information that a script can only obtain once it starts its execution. Additionally, many built-in functions require “simple” arguments that do not change over time. Wherever a script allows “simple” values, it can also accept values qualified as “input” or “const”.

This script highlights the background to warn users that they're using a non-standard chart type. It uses the value of `chart.is_standard` to calculate the `isNonStandard` variable, then uses that variable's value to calculate a `warningColor` that also references a "simple" value. The `color` parameter of `bcolor()` allows a "series color" argument, meaning it can also accept a "simple color" value since "simple" is lower on the hierarchy:

```
1 // @version=5
2 indicator("simple demo", overlay = true)
3
4 // @variable Is `true` when the current chart is non-standard. Qualified as "simple
5 // → bool".
6 isNonStandard = not chart.is_standard
7 // @variable Is orange when the the current chart is non-standard. Qualified as
8 // → "simple color".
9 simple color warningColor = isNonStandard ? color.new(color.orange, 70) : na
10 bcolor(warningColor, title = "Non-standard chart color")
```

### series

Values qualified as "series" provide the most flexibility in scripts since they can change on any bar, even multiple times on the same bar.

Users can explicitly define variables and parameters that accept "series" values by including the `series` keyword in their declaration.

Built-in variables such as `open`, `high`, `low`, `close`, `volume`, `time`, and `bar_index`, and the result from any expression using such built-ins, are qualified as "series". The result of any function or operation that returns a dynamic value will always be a "series", as will the results from using the history-referencing operator `[]` to access historical values. Wherever a script allows "series" values, it will also accept values with any other qualifier, as "series" is the *highest* qualifier on the hierarchy.

This script displays the `highest` and `lowest` value of a `sourceInput` over `lengthInput` bars. The values assigned to the `highest` and `lowest` variables are of the "series float" qualified type, as they can change throughout the script's execution:

```
1 // @version=5
2 indicator("series demo", overlay = true)
3
4 // @variable The source value to calculate on. Qualified as "series float".
5 series float sourceInput = input.source(close, "Source")
6 // @variable The number of bars in the calculation. Qualified as "input int".
7 lengthInput = input.int(20, "Length")
8
9 // @variable The highest `sourceInput` value over `lengthInput` bars. Qualified as
10 // → "series float".
11 series float highest = ta.highest(sourceInput, lengthInput)
12 // @variable The lowest `sourceInput` value over `lengthInput` bars. Qualified as
13 // → "series float".
14 lowest = ta.lowest(sourceInput, lengthInput)
15 plot(highest, "Highest source", color.green)
16 plot(lowest, "Lowest source", color.red)
```

### 3.9.3 Types

Pine Script™ *types* classify values and determine the functions and operations they're compatible with. They include:

- The fundamental types: `int`, `float`, `bool`, `color`, and `string`
- The special types: `plot`, `hline`, `line`, `linefill`, `box`, `polyline`, `label`, `table`, `chart.point`, `array`, `matrix`, and `map`
- *User-defined types (UDTs)*
- `void`

Fundamental types refer to the underlying nature of a value, e.g., a value of 1 is of the “int” type, 1.0 is of the “float” type, “AAPL” is of the “string” type, etc. Special types and user-defined types utilize *IDs* that refer to objects of a specific class. For example, a value of the “label” type contains an ID that acts as a *pointer* referring to a “label” object. The “void” type refers to the output from a function or *method* that does not return a usable value.

Pine Script™ can automatically convert values from some types into others. The auto-casting rules are: **int → float → bool**. See the *Type casting* section of this page for more information.

In most cases, Pine Script™ can automatically determine a value's type. However, we can also use type keywords to *explicitly* specify types for readability and for code that requires explicit definitions (e.g., declaring a variable assigned to `na`). For example:

```

1 // @version=5
2 indicator("Types demo", overlay = true)
3
4 //@variable A value of the "const string" type for the `ma` plot's title.
5 string MA_TITLE = "MA"
6
7 //@variable A value of the "input int" type. Controls the length of the average.
8 int lengthInput = input.int(100, "Length", minval = 2)
9
10 //@variable A "series float" value representing the last `close` that crossed over_
11 //the `ma`.
11 var float crossValue = na
12
13 //@variable A "series float" value representing the moving average of `close`.
14 float ma = ta.sma(close, lengthInput)
15 //@variable A "series bool" value that's `true` when the `close` crosses over the_
16 //`ma`.
16 bool crossUp = ta.crossover(close, ma)
17 //@variable A "series color" value based on whether `close` is above or below its_
18 //`ma`.
18 color maColor = close > ma ? color.lime : color.fuchsia
19
20 // Update the `crossValue`.
21 if crossUp
22     crossValue := close
23
24 plot(ma, MA_TITLE, maColor)
25 plot(crossValue, "Cross value", style = plot.style_circles)
26 plotchar(crossUp, "Cross Up", "▲", location.belowbar, size = size.small)

```

### int

Values of the “int” type represent integers, i.e., whole numbers without any fractional quantities.

Integer literals are numeric values written in *decimal* notation. For example:

```
1  
-1  
750
```

Built-in variables such as `bar_index`, `time`, `timenow`, `dayofmonth`, and `strategy.wintrades` all return values of the “int” type.

### float

Values of the “float” type represent floating-point numbers, i.e., numbers that can contain whole and fractional quantities.

Floating-point literals are numeric values written with a `.` delimiter. They may also contain the symbol `e` or `E` (which means “10 raised to the power of X”, where X is the number after the `e` or `E` symbol). For example:

```
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 “float” values in Pine Script™ is 1e-16.

Built-in variables such as `close`, `hlcc4`, `volume`, `ta.vwap`, and `strategy.position_size` all return values of the “float” type.

### bool

Values of the “bool” type represent the truth value of a comparison or condition, which scripts can use in *conditional structures* and other expressions.

There are only two literals that represent boolean values:

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

When an expression of the “bool” type returns `na`, scripts treat its value as `false` when evaluating conditional statements and operators.

Built-in variables such as `barstate.isfirst`, `chart.is_heikinashi`, `session.ismarket`, and `timeframe.isdaily` all return values of the “bool” type.

### color

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

- RR, GG and BB pairs respectively represent the values for the color’s red, green and blue components.
- AA is an optional value for the color’s opacity (or *alpha* component) where 00 is invisible and FF opaque. When the literal does not include an AA pair, the script treats it as fully opaque (the same as using FF).
- The hexadecimal letters in the literals can be uppercase or lowercase.

These are examples of “color” literals:

```
#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 red color
```

Pine Script™ also has *built-in color constants*, including `color.green`, `color.red`, `color.orange`, `color.blue` (the default color in `plot*` () functions and many of the default color-related properties in *drawing types*), etc.

When using built-in color constants, it is possible to add transparency information to them via the `color.new()` function.

Note that when specifying red, green or blue components in `color.*()` functions, we use “int” or “float” arguments with values between 0 and 255. When specifying transparency, we use a value between 0 and 100, where 0 means fully opaque and 100 means completely transparent. For example:

```
1 // @version=5
2 indicator("Shading the chart's background", overlay = true)
3
4 // @variable A "const color" value representing the base for each day's color.
5 color BASE_COLOR = color.rgb(0, 99, 165)
6
7 // @variable A "series int" value that modifies the transparency of the `BASE_COLOR` ↴
8 // in `color.new()` .
9 int transparency = 50 + int(40 * dayofweek / 7)
10
11 // Color the background using the modified `BASE_COLOR` .
12 bgcolor(color.new(BASE_COLOR, transparency))
```

See the User Manual’s page on *colors* for more information on using colors in scripts.

## string

Values of the “string” type represent sequences of letters, numbers, symbols, spaces, and other characters.

String literals in Pine are characters enclosed in single or double quotation marks. For example:

```
"This is a string literal using double quotes."
'This is a string literal using single quotes.'
```

Single and double quotation marks are functionally equivalent in Pine Script™. A “string” enclosed within double quotation marks can contain any number of single quotation marks and vice versa:

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

Scripts can *escape* the enclosing delimiter in a “string” using the backslash character (\). For example:

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

We can create “string” values containing the new line escape character (\n) for displaying multi-line text with `plot()` and `log.*()` functions and objects of *drawing types*. For example:

```
"This\nString\nHas\nOne\nWord\nPer\nLine"
```

We can use the `+` operator to concatenate “string” values:

```
"This is a " + "concatenated string."
```

The built-ins in the `str.*()` namespace create “string” values using specialized operations. For instance, this script creates a *formatted string* to represent “float” price values and displays the result using a label:

```
1 // @version=5
2 indicator("Formatted string demo", overlay = true)
3
4 //@variable A "series string" value representing the bar's OHLC data.
5 string ohlcString = str.format("Open: {0}\nHigh: {1}\nLow: {2}\nClose: {3}", open,
6   high, low, close)
7
8 // Draw a label containing the `ohlcString`.
9 label.new(bar_index, high, ohlcString, textcolor = color.white)
```

See our User Manual’s page on [Text and shapes](#) for more information about displaying “string” values from a script.

Built-in variables such as `syminfo.tickerid`, `syminfo.currency`, and `timeframe.period` return values of the “string” type.

### plot and hline

Pine Script™’s `plot()` and `hline()` functions return IDs that respectively reference instances of the “plot” and “hline” types. These types display calculated values and horizontal levels on the chart, and one can assign their IDs to variables for use with the built-in `fill()` function.

For example, this script plots two EMAs on the chart and fills the space between them using the `fill()` function:

```
1 // @version=5
2 indicator("plot fill demo", overlay = true)
3
4 //@variable A "series float" value representing a 10-bar EMA of `close`.
5 float emaFast = ta.ema(close, 10)
6 //@variable A "series float" value representing a 20-bar EMA of `close`.
7 float emaSlow = ta.ema(close, 20)
8
9 //@variable The plot of the `emaFast` value.
10 emaFastPlot = plot(emaFast, "Fast EMA", color.orange, 3)
11 //@variable The plot of the `emaSlow` value.
12 emaSlowPlot = plot(emaSlow, "Slow EMA", color.gray, 3)
13
14 // Fill the space between the `emaFastPlot` and `emaSlowPlot`.
15 fill(emaFastPlot, emaSlowPlot, color.new(color.purple, 50), "EMA Fill")
```

It’s important to note that unlike other special types, there is no `plot` or `hline` keyword in Pine to explicitly declare a variable’s type as “plot” or “hline”.

Users can control where their scripts’ plots display via the variables in the `display.*` namespace. Additionally, one script can use the values from another script’s plots as *external inputs* via the `input.source()` function (see our User Manual’s section on [source inputs](#)).

## Drawing types

Pine Script™ drawing types allow scripts to create custom drawings on charts. They include the following: `line`, `linefill`, `box`, `polyline`, `label`, and `table`.

Each type also has a namespace containing all the built-ins that create and manage drawing instances. For example, the following `*.new()` constructors create new objects of these types in a script: `line.new()`, `linefill.new()`, `box.new()`, `polyline.new()`, `label.new()`, and `table.new()`.

Each of these functions returns an *ID* which is a reference that uniquely identifies a drawing object. IDs are always qualified as “series”, meaning their qualified types are “series line”, “series label”, etc. Drawing IDs act like pointers, as each ID references a specific instance of a drawing in all the functions from that drawing’s namespace. For instance, the ID of a line returned by a `line.new()` call is used later to refer to that specific object once it’s time to delete it with `line.delete()`.

## Chart points

Chart points are special types that represent coordinates on the chart. Scripts use the information from `chart.point` objects to determine the chart locations of `lines`, `boxes`, `polylines`, and `labels`.

Objects of this type contain three *fields*: `time`, `index`, and `price`. Whether a drawing instance uses the `time` or `price` field from a `chart.point` as an x-coordinate depends on the drawing’s `xloc` property.

We can use any of the following functions to create chart points in a script:

- `chart.point.new()` - Creates a new `chart.point` with a specified `time`, `index`, and `price`.
- `chart.point.now()` - Creates a new `chart.point` with a specified `price` y-coordinate. The `time` and `index` fields contain the `time` and `bar_index` of the bar the function executes on.
- `chart.point_from_index()` - Creates a new `chart.point` with an `index` x-coordinate and `price` y-coordinate. The `time` field of the resulting instance is `na`, meaning it will not work with drawing objects that use an `xloc` value of `xloc.bar_time`.
- `chart.point.from_time()` - Creates a new `chart.point` with a `time` x-coordinate and `price` y-coordinate. The `index` field of the resulting instance is `na`, meaning it will not work with drawing objects that use an `xloc` value of `xloc.bar_index`.
- `chart.point.copy()` - Creates a new `chart.point` containing the same `time`, `index`, and `price` information as the `id` in the function call.

This example draws lines connecting the previous bar’s `high` to the current bar’s `low` on each chart bar. It also displays labels at both points of each line. The line and labels get their information from the `firstPoint` and `secondPoint` variables, which reference chart points created using `chart.point_from_index()` and `chart.point.now()`:

```

1 // @version=5
2 indicator("Chart points demo", overlay = true)
3
4 // @variable A new `chart.point` at the previous `bar_index` and `high`.
5 firstPoint = chart.point.from_index(bar_index - 1, high[1])
6 // @variable A new `chart.point` at the current bar's `low`.
7 secondPoint = chart.point.now(low)
8
9 // Draw a new line connecting coordinates from the `firstPoint` and `secondPoint`.
10 // This line uses the `index` fields from the points as x-coordinates.
11 line.new(firstPoint, secondPoint, color = color.purple, width = 3)
12 // Draw a label at the `firstPoint`. Uses the point's `index` field as its x-
13 // coordinate.
14 label.new(
    firstPoint, str.tostring(firstPoint.price), color = color.green,

```

(continues on next page)

(continued from previous page)

```

15     style = label.style_label_down, textcolor = color.white
16   )
17 // Draw a label at the `secondPoint`. Uses the point's `index` field as its x-
18 // coordinate.
18 label.new(
19   secondPoint, str.tostring(secondPoint.price), color = color.red,
20   style = label.style_label_up, textcolor = color.white
21 )

```

## Collections

Collections in Pine Script™ ([arrays](#), [matrices](#), and [maps](#)) utilize reference IDs, much like other special types (e.g., labels). The type of the ID defines the type of *elements* the collection will contain. In Pine, we specify array, matrix, and map types by appending a [type template](#) to the [array](#), [matrix](#), or [map](#) keywords:

- `array<int>` defines an array containing “int” elements.
- `array<label>` defines an array containing “label” IDs.
- `array<UDT>` defines an array containing IDs referencing objects of a [user-defined type \(UDT\)](#).
- `matrix<float>` defines a matrix containing “float” elements.
- `matrix<UDT>` defines a matrix containing IDs referencing objects of a [user-defined type \(UDT\)](#).
- `map<string, float>` defines a map containing “string” keys and “float” values.
- `map<int, UDT>` defines a map containing “int” keys and IDs of [user-defined type \(UDT\)](#) instances as values.

For example, one can declare an “int” array with a single element value of 10 in any of 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 specify an array of “int” elements, but its use is discouraged. No equivalent exists to specify the types of matrices or maps in that way.
- Type-specific built-ins exist for arrays, such as `array.new_int()`, 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 scripts can create [objects](#). UDTs are composite types; they contain an arbitrary number of *fields* that can be of any type, including other user-defined types. The syntax to define a user-defined type is:

```

[export] type <UDT_identifier>
  <field_type> <field_name> [= <value>]
  ...

```

where:

- `export` is the keyword that a [library](#) script uses to export the user-defined type. To learn more about exporting UDTs, see our User Manual’s [Libraries](#) page.

- <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 the script will assign to it when creating new objects of that UDT. If one does not provide a value, the field's default is `na`. The same rules as those governing the default values of parameters in function signatures apply to the default values of fields. For example, a UDT's default values cannot use results from the history-referencing operator `[]` or expressions.

This example declares a `pivotPoint` UDT with an “int” `pivotTime` field and a “float” `priceLevel` field that will respectively hold time and price information about a calculated pivot:

```
//@type          A user-defined type containing pivot information.
//@field pivotTime Contains time information about the pivot.
//@field priceLevel Contains price information about the pivot.
type pivotPoint
    int      pivotTime
    float   priceLevel
```

User-defined types support *type recursion*, i.e., the fields of a UDT can reference objects of the same UDT. Here, we've added a `nextPivot` field to our previous `pivotPoint` type that references another `pivotPoint` instance:

```
//@type          A user-defined type containing pivot information.
//@field pivotTime Contains time information about the pivot.
//@field priceLevel Contains price information about the pivot.
//@field nextPivot A `pivotPoint` instance containing additional pivot information.
type pivotPoint
    int      pivotTime
    float   priceLevel
    pivotPoint nextPivot
```

Scripts can use two built-in methods to create and copy UDTs: `new()` and `copy()`. See our User Manual's page on [Objects](#) to learn more about working with UDTs.

## 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 usable value.

Scripts cannot use “void” results in expressions or assign them to variables. No `void` keyword exists in Pine Script™ since one cannot declare a variable of the “void” type.

### 3.9.4 `na` value

There is a special value in Pine Script™ called `na`, which is an acronym for *not available*. We use `na` to represent an undefined value from a variable or expression. It is similar to `null` in Java and `None` in Python.

Scripts can automatically cast `na` values to almost any type. However, in some cases, the compiler cannot infer the type associated with an `na` value because more than one type-casting rule may apply. For example:

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

The above line of code causes a compilation error because the compiler cannot determine the nature of the `myVar` variable, i.e., whether the variable will reference numeric values for plotting, string values for setting text in a label, or other values for some other purpose later in the script's execution.

To resolve such errors, we must explicitly declare the type associated with the variable. Suppose the `myVar` variable will reference “float” values in subsequent script iterations. We can resolve the error by declaring the variable with the `float` keyword:

```
float myVar = na
```

or by explicitly casting the `na` value to the “float” type via the `float()` function:

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

To test if the value from a variable or expression is `na`, we call the `na()` function, which returns `true` if the value is undefined. For example:

```
//@variable Is 0 if the `myVar` is `na`, `close` otherwise.
float myClose = na(myVar) ? 0 : close
```

Do not use the `==` comparison operator to test for `na` values, as scripts cannot determine the equality of an undefined value:

```
//@variable Returns the `close` value. The script cannot compare the equality of `na` ↴
values, as they're undefined.
float myClose = myVar == na ? 0 : close
```

Best coding practices often involve handling `na` values to prevent undefined values in calculations.

For example, this line of code checks if the `close` value on the current bar is greater than the previous bar's value:

```
//@variable Is `true` when the `close` exceeds the last bar's `close`, `false` ↴
otherwise.
bool risingClose = close > close[1]
```

On the first chart bar, the value of `risingClose` is `na` since there is no past `close` value to reference.

We can ensure the expression also returns an actionable value on the first bar by replacing the undefined past value with a value from the current bar. This line of code uses the `nz()` function to replace the past bar's `close` with the current bar's `open` when the value is `na`:

```
//@variable Is `true` when the `close` exceeds the last bar's `close` (or the current ↴
`open` if the value is `na`).
bool risingClose = close > nz(close[1], open)
```

Protecting scripts against `na` instances helps to prevent undefined values from propagating in a calculation's results. For example, this script declares an `allTimeHigh` variable on the first bar. It then uses the `math.max()` between the `allTimeHigh` and the bar's `high` to update the `allTimeHigh` throughout its execution:

```
1 // @version=5
2 indicator("na protection demo", overlay = true)
3
4 //@variable The result of calculating the all-time high price with an initial value ↴
5 // of `na`.
6 var float allTimeHigh = na
7
8 // Reassign the value of the `allTimeHigh`.
9 // Returns `na` on all bars because `math.max()` can't compare the `high` to an ↴
```

(continues on next page)

(continued from previous page)

```

9 ↵undefined value.
10 allTimeHigh := math.max(allTimeHigh, high)
11 plot(allTimeHigh) // Plots `na` on all bars.

```

This script plots a value of `na` on all bars, as we have not included any `na` protection in the code. To fix the behavior and plot the intended result (i.e., the all-time high of the chart's prices), we can use `nz()` to replace `na` values in the `allTimeHigh` series:

```

1 //@version=5
2 indicator("na protection demo", overlay = true)
3
4 //@variable The result of calculating the all-time high price with an initial value_
5 ↵of `na`.
6 var float allTimeHigh = na
7
8 // Reassign the value of the `allTimeHigh`.
9 // We've used `nz()` to prevent the initial `na` value from persisting throughout the_
10 ↵calculation.
11 allTimeHigh := math.max(nz(allTimeHigh), high)
12 plot(allTimeHigh)

```

### 3.9.5 Type templates

Type templates specify the data types that collections (`arrays`, `matrices`, and `maps`) can contain.

Templates for `arrays` and `matrices` consist of a single type identifier surrounded by angle brackets, e.g., `<int>`, `<label>`, and `<PivotPoint>` (where `PivotPoint` is a *user-defined type (UDT)*).

Templates for `maps` consist of two type identifiers enclosed in angle brackets, where the first specifies the type of *keys* in each key-value pair, and the second specifies the *value* type. For example, `<string, float>` is a type template for a map that holds `string` keys and `float` values.

Users can construct type templates from:

- Fundamental types: `int`, `float`, `bool`, `color`, and `string`
- The following special types: `line`, `linefill`, `box`, `polyline`, `label`, `table`, and `chart.point`
- *User-defined types (UDTs)*

**Note that:**

- *Maps* can use any of these types as *values*, but they can only accept fundamental types as *keys*.

Scripts use type templates to declare variables that point to collections, and when creating new collection instances. For example:

```

1 //@version=5
2 indicator("Type templates demo")
3
4 //@variable A variable initially assigned to `na` that accepts arrays of "int" values.
5 array<int> intArray = na
6 //@variable An empty matrix that holds "float" values.
7 floatMatrix = matrix.new<float>()
8 //@variable An empty map that holds "string" keys and "color" values.
9 stringColorMap = map.new<string, color>()

```

### 3.9.6 Type casting

Pine Script™ includes an automatic type-casting mechanism that *casts* (converts) “int” values to “float” when necessary. Variables or expressions requiring “float” values can also use “int” values because any integer can be represented as a floating point number with its fractional part equal to 0.

For the sake of backward compatibility, Pine Script™ also automatically casts “int” and “float” values to “bool” when necessary. When passing numeric values to the parameters of functions and operations that expect “bool” types, Pine auto-casts them to “bool”. However, we do not recommend relying on this behavior. Most scripts that automatically cast numeric values to the “bool” type will produce a *compiler warning*. One can avoid the compiler warning and promote code readability by using the `bool()` function, which explicitly casts a numeric value to the “bool” type.

When casting an “int” or “float” to “bool”, a value of 0 converts to `false` and any other numeric value always converts to `true`.

This code below demonstrates deprecated auto-casting behavior in Pine. It creates a `randomValue` variable with a “series float” value on every bar, which it passes to the `condition` parameter in an `if` structure and the `series` parameter in a `plotchar()` function call. Since both parameters accept “bool” values, the script automatically casts the `randomValue` to “bool” when evaluating them:

```

1 // @version=5
2 indicator("Auto-casting demo", overlay = true)
3
4 // @variable A random rounded value between -1 and 1.
5 float randomValue = math.round(math.random(-1, 1))
6 // @variable The color of the chart background.
7 color bgColor = na
8
9 // This raises a compiler warning since `randomValue` is a "float", but `if` expects_
10 // a "bool".
11 if randomValue
12     bgColor := color.new(color.blue, 60)
13 // This does not raise a warning, as the `bool()`` function explicitly casts the_
14 // `randomValue` to "bool".
15 if bool(randomValue)
16     bgColor := color.new(color.blue, 60)
17
18 // Display unicode characters on the chart based on the `randomValue`.
19 // Whenever `math.random()` returns 0, no character will appear on the chart because_
20 // 0 converts to `false`.
21 plotchar(randomValue)
22 // We recommend explicitly casting the number with the `bool()`` function to make the_
23 // type transformation more obvious.
24 plotchar(bool(randomValue))
25
26 // Highlight the background with the `bgColor`.
27 bgcolor(bgColor)

```

It's sometimes necessary to cast one type to another when auto-casting rules do not suffice. For such cases, the following type-casting functions are available: `int()`, `float()`, `bool()`, `color()`, `string()`, `line()`, `linefill()`, `label()`, `box()`, and `table()`.

The example below shows a code that tries to use a “const float” value as the `length` argument in the `ta.sma()` function call. The script will fail to compile, as it cannot automatically convert the “float” value to the required “int” type:

```

1 // @version=5
2 indicator("Explicit casting demo", overlay = true)
3
4 // @variable The length of the SMA calculation. Qualified as "const float".

```

(continues on next page)

(continued from previous page)

```

5 float LENGTH = 10.0
6
7 float sma = ta.sma(close, LENGTH) // Compilation error. The `length` parameter
  ↪ requires an "int" value.
8
9 plot(sma)

```

The code raises the following error: “*Cannot call ‘ta.sma’ with argument ‘length’=‘LENGTH’. An argument of ‘const float’ type was used but a ‘series int’ is expected.*”

The compiler is telling us that the code is using a “float” value where an “int” is required. There is no auto-casting rule to cast a “float” to an “int”, so we must do the job ourselves. In this version of the code, we’ve used the `int()` function to explicitly convert our “float” `LENGTH` value to the “int” type within the `ta.sma()` call:

```

1 // @version=5
2 indicator("explicit casting demo")
3
4 // @variable The length of the SMA calculation. Qualified as "const float".
5 float LENGTH = 10.0
6
7 float sma = ta.sma(close, int(LENGTH)) // Compiles successfully since we've converted
  ↪ the `LENGTH` to "int".
8
9 plot(sma)

```

Explicit type casting is also handy when declaring variables assigned to `na`, as explained in the [previous section](#).

For example, once could explicitly declare a variable with a value of `na` as a “label” type in either of the following, equivalent ways:

```

// Explicitly specify that the variable references "label" objects:
label myLabel = na

// Explicitly cast the `na` value to the "label" type:
myLabel = label(na)

```

### 3.9.7 Tuples

A *tuple* is a comma-separated set of expressions enclosed in brackets. When a function, *method*, or other local block returns more than one value, scripts return those values in the form of a tuple.

For example, the following *user-defined function* returns the sum and product of two “float” values:

```

// @function Calculates the sum and product of two values.
calcSumAndProduct(float a, float b) =>
    // @variable The sum of `a` and `b`.
    float sum = a + b
    // @variable The product of `a` and `b`.
    float product = a * b
    // Return a tuple containing the `sum` and `product`.
    [sum, product]

```

When we call this function later in the script, we use a *tuple declaration* to declare multiple variables corresponding to the values returned by the function call:

```
// Declare a tuple containing the sum and product of the `high` and `low`,  
// respectively.  
[hlSum, hlProduct] = calcSumAndProduct(high, low)
```

Keep in mind that unlike declaring single variables, we cannot explicitly define the types the tuple's variables (`hlSum` and `hlProduct` in this case), will contain. The compiler automatically infers the types associated with the variables in a tuple.

In the above example, the resulting tuple contains values of the same type (“float”). However, it’s important to note that tuples can contain values of *multiple types*. For example, the `chartInfo()` function below returns a tuple containing “int”, “float”, “bool”, “color”, and “string” values:

```
//@function Returns information about the current chart.  
chartInfo() =>  
    //@variable The first visible bar's UNIX time value.  
    int firstVisibleTime = chart.left_visible_bar_time  
    //@variable The `close` value at the `firstVisibleTime`.  
    float firstVisibleClose = ta.valuewhen(ta.cross(time, firstVisibleTime), close, 0)  
    //@variable Is `true` when using a standard chart type, `false` otherwise.  
    bool isStandard = chart.is_standard  
    //@variable The foreground color of the chart.  
    color fgColor = chart.fg_color  
    //@variable The ticker ID of the current chart.  
    string symbol = syminfo.tickerid  
    // Return a tuple containing the values.  
    [firstVisibleTime, firstVisibleClose, isStandard, fgColor, symbol]
```

Tuples are especially handy for requesting multiple values in one `request.security()` call.

For instance, this `roundedOHLC()` function returns a tuple containing OHLC values rounded to the nearest prices that are divisible by the symbol’s `minimum tick` value. We call this function as the `expression` argument in `request.security()` to request a tuple containing daily OHLC values:

```
//@function Returns a tuple of OHLC values, rounded to the nearest tick.  
roundedOHLC() =>  
    [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_  
     mintick(low), math.round_to_mintick(close)]  
  
[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", roundedOHLC())
```

We can also achieve the same result by directly passing a tuple of rounded values as the `expression` in the `request.security()` call:

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

Local blocks of *conditional structures*, including `if` and `switch` statements, can return tuples. For example:

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

and:

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

However, ternaries cannot contain tuples, as the return values in a ternary statement are not considered local blocks:

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

Note that all items within a tuple returned from a function are qualified as “simple” or “series”, depending on its contents. If a tuple contains a “series” value, all other elements within the tuple will also adopt the “series” qualifier. For example:

```
1 // @version=5
2 indicator("Qualified types in tuples demo")
3
4 makeTicker(simple string prefix, simple string ticker) =>
5     tId = prefix + ":" + ticker // simple string
6     source = close // series float
7     [tId, source]
8
9 // Both variables are series now.
10 [tId, source] = makeTicker("BATS", "AAPL")
11
12 // Error cannot call 'request.security' with 'series string' tId.
13 r = request.security(tId, "", source)
14
15 plot(r)
```



## 3.10 Built-ins

- *Introduction*
- *Built-in variables*
- *Built-in functions*

### 3.10.1 Introduction

Pine Script™ has hundreds of *built-in* variables and functions. They provide your scripts with valuable information and make calculations for you, dispensing you from coding them. The better you know the built-ins, the more you will be able to do with your Pine scripts.

In this page we present an overview of some of Pine Script™'s built-in variables and functions. They will be covered in more detail in the pages of this manual covering specific themes.

All built-in variables and functions are defined in the Pine Script™ [v5 Reference Manual](#). It is called a “Reference Manual” because it is the definitive reference on the Pine Script™ language. It is an essential tool that will accompany you anytime you code in Pine, whether you are a beginner or an expert. If you are learning your first programming language, make the [Reference Manual](#) your friend. Ignoring it will make your programming experience with Pine Script™ difficult and frustrating — as it would with any other programming language.

Variables and functions in the same family share the same *namespace*, which is a prefix to the function's name. The `ta.sma()` function, for example, is in the `ta` namespace, which stands for “technical analysis”. A namespace can contain both variables and functions.

Some variables have function versions as well, e.g.:

- The `ta.tr` variable returns the “True Range” of the current bar. The `ta.tr(true)` function call also returns the “True Range”, but when the previous `close` value which is normally needed to calculate it is `na`, it calculates using `high - low` instead.
- The `time` variable gives the time at the `open` of the current bar. The `time(timeframe)` function returns the time of the bar's `open` from the `timeframe` specified, even if the chart's timeframe is different. The `time(timeframe, session)` function returns the time of the bar's `open` from the `timeframe` specified, but only if it is within the `session` time. The `time(timeframe, session, timezone)` function returns the time of the bar's `open` from the `timeframe` specified, but only if it is within the `session` time in the specified `timezone`.

### 3.10.2 Built-in variables

Built-in variables exist for different purposes. These are a few examples:

- Price- and volume-related variables: `open`, `high`, `low`, `close`, `h12`, `hlc3`, `ohlc4`, and `volume`.
- Symbol-related information in the `syminfo` namespace: `syminfo.basecurrency`, `syminfo.currency`, `syminfo.description`, `syminfo.mintick`, `syminfo.pointvalue`, `syminfo.prefix`, `syminfo.root`, `syminfo.session`, `syminfo.ticker`, `syminfo.tickerid`, `syminfo.timezone`, and `syminfo.type`.
- Timeframe (a.k.a. “interval” or “resolution”, e.g., `15sec`, `30min`, `60min`, `1D`, `3M`) variables in the `timeframe` namespace: `timeframe.isseconds`, `timeframe.isminutes`, `timeframe.isintraday`, `timeframe.isdaily`, `timeframe.isweekly`, `timeframe.ismonthly`, `timeframe.isdwm`, `timeframe.multiplier`, and `timeframe.period`.
- Bar states in the `barstate` namespace (see the [Bar states](#) page): `barstate.isconfirmed`, `barstate.isfirst`, `barstate.ishistory`, `barstate.islast`, `barstate.islastconfirmedhistory`, `barstate.isnew`, and `barstate.isrealtime`.
- Strategy-related information in the `strategy` namespace: `strategy.equity`, `strategy.initial_capital`, `strategy.grossloss`, `strategy.grossprofit`, `strategy.wintrades`, `strategy.losstrades`, `strategy.position_size`, `strategy.position_avg_price`, `strategy.wintrades`, etc.

### 3.10.3 Built-in functions

Many functions are used for the result(s) they return. These are a few examples:

- Math-related functions in the `math` namespace: `math.abs()`, `math.log()`, `math.max()`, `math.random()`, `math.round_to_mintick()`, etc.
- Technical indicators in the `ta` namespace: `ta.sma()`, `ta.ema()`, `ta.macd()`, `ta.rsi()`, `ta.supertrend()`, etc.
- Support functions often used to calculate technical indicators in the `ta` namespace: `ta.barssince()`, `ta.crossover()`, `ta.highest()`, etc.
- Functions to request data from other symbols or timeframes in the `request` namespace: `request.dividends()`, `request.earnings()`, `request.financial()`, `request.quandl()`, `request.security()`, `request.splits()`.
- Functions to manipulate strings in the `str` namespace: `str.format()`, `str.length()`, `str.tonumber()`, `str.tostring()`, etc.
- Functions used to define the input values that script users can modify in the script's "Settings/Inputs" tab, in the `input` namespace: `input()`, `input.color()`, `input.int()`, `input.session()`, `input.symbol()`, etc.
- Functions used to manipulate colors in the `color` namespace: `color.from_gradient()`, `color.new()`, `color.rgb()`, etc.

Some functions do not return a result but are used for their side effects, which means they do something, even if they don't return a result:

- Functions used as a declaration statement defining one of three types of Pine scripts, and its properties. Each script must begin with a call to one of these functions: `indicator()`, `strategy()` or `library()`.
- Plotting or coloring functions: `bgcolor()`, `plotbar()`, `plotcandle()`, `plotchar()`, `plotshape()`, `fill()`.
- Strategy functions placing orders, in the `strategy` namespace: `strategy.cancel()`, `strategy.close()`, `strategy.entry()`, `strategy.exit()`, `strategy.order()`, etc.
- Strategy functions returning information on individual past trades, in the `strategy` namespace: `strategy.closedtrades.entry_bar_index()`, `strategy.closedtrades.entry_price()`, `strategy.closedtrades.entry_time()`, `strategy.closedtrades.exit_bar_index()`, `strategy.closedtrades.max_drawdown()`, `strategy.closedtrades.max_runup()`, `strategy.closedtrades.profit()`, etc.
- Functions to generate alert events: `alert()` and `alertcondition()`.

Other functions return a result, but we don't always use it, e.g.: `hline()`, `plot()`, `array.pop()`, `label.new()`, etc.

All built-in functions are defined in the Pine Script™ v5 Reference Manual. You can click on any of the function names listed here to go to its entry in the Reference Manual, which documents the function's signature, i.e., the list of *parameters* it accepts and the qualified type of the value(s) it returns (a function can return more than one result). The Reference Manual entry will also list, for each parameter:

- Its name.
- The qualified type of the value it requires (we use *argument* to name the values passed to a function when calling it).
- If the parameter is required or not.

All built-in functions have one or more parameters defined in their signature. Not all parameters are required for every function.

Let's look at the `ta.vwma()` function, which returns the volume-weighted moving average of a source value. This is its entry in the Reference Manual:

**ta.vwma**

The `vwma` function returns volume-weighted moving average of `'source'` for `'length'` bars back. It is the same as:  $\text{sma}(\text{source} * \text{volume}, \text{length}) / \text{sma}(\text{volume}, \text{length})$ .

```
ta.vwma(source, length) → series float
```

**EXAMPLE**

```
plot(ta.vwma(close, 15))

// same on pine, but less efficient
pine_vwma(x, y) =>
    ta.sma(x * volume, y) / ta.sma(volume, y)
plot(pine_vwma(close, 15))
```

**RETURNS**  
Volume-weighted moving average of `'source'` for `'length'` bars back.

**ARGUMENTS**

- source** (series int/float) Series of values to process.
- length** (series int) Number of bars (length).

**SEE ALSO**

[ta.sma](#) [ta.ema](#) [ta.rma](#) [ta.wma](#) [ta.swma](#) [ta.alma](#)

The entry gives us the information we need to use it:

- What the function does.
- Its signature (or definition):

`ta.vwma(source, length) → series float`

- The parameters it includes: `source` and `length`
- The qualified type of the result it returns: “series float”.
- An example showing it in use: `plot(ta.vwma(close, 15))`.
- An example showing what it does, but in long form, so you can better understand its calculations. Note that this is meant to explain — not as usable code, because it is more complicated and takes longer to execute. There are only disadvantages to using the long form.
- The “RETURNS” section explains exactly what value the function returns.
- The “ARGUMENTS” section lists each parameter and gives the critical information concerning what qualified type is required for arguments used when calling the function.
- The “SEE ALSO” section refers you to related Reference Manual entries.

This is a call to the function in a line of code that declares a `myVwma` variable and assigns the result of `ta.vwma(close, 20)` to it:

`myVwma = ta.vwma(close, 20)`

Note that:

- We use the built-in variable `close` as the argument for the `source` parameter.
- We use `20` as the argument for the `length` parameter.

- If placed in the global scope (i.e., starting in a line's first position), it will be executed by the Pine Script™ runtime on each bar of the chart.

We can also use the parameter names when calling the function. Parameter names are called *keyword arguments* when used in a function call:

```
myVwma = ta.vwma(source = close, length = 20)
```

You can change the position of arguments when using keyword arguments, but only if you use them for all your arguments. When calling functions with many parameters such as `indicator()`, you can also forego keyword arguments for the first arguments, as long as you don't skip any. If you skip some, you must then use keyword arguments so the Pine Script™ compiler can figure out which parameter they correspond to, e.g.:

```
indicator("Example", "Ex", true, max_bars_back = 100)
```

Mixing things up this way is not allowed:

```
indicator(precision = 3, "Example") // Compilation error!
```

**When calling built-ins, it is critical to ensure that the arguments you use are of the required qualified type, which will vary for each parameter.**

To learn how to do this, one needs to understand Pine Script™'s *type system*. The Reference Manual entry for each built-in function includes an “ARGUMENTS” section which lists the qualified type required for the argument supplied to each of the function's parameters.



## 3.11 User-defined functions

- *Introduction*
- *Single-line functions*
- *Multi-line functions*
- *Scopes in the script*
- *Functions that return multiple results*
- *Limitations*

### 3.11.1 Introduction

User-defined functions are functions that you write, as opposed to the built-in functions in Pine Script™. They are useful to define calculations that you must do repetitively, or that you want to isolate from your script's main section of calculations. Think of user-defined functions as a way to extend the capabilities of Pine Script™, when no built-in function will do what you need.

You can write your functions in two ways:

- In a single line, when they are simple, or
- On multiple lines

Functions can be located in two places:

- If a function is only used in one script, you can include it in the script where it is used. See our [Style guide](#) for recommendations on where to place functions in your script.
- You can create a Pine Script™ *library* to include your functions, which makes them reusable in other scripts without having to copy their code. Distinct requirements exist for library functions. They are explained in the page on [libraries](#).

Whether they use one line or multiple lines, user-defined functions have the following characteristics:

- They cannot be embedded. All functions are defined in the script's global scope.
- They do not support recursion. It is **not allowed** for a function to call itself from within its own code.
- The type of the value returned by a function is determined automatically and depends on the type of arguments used in each particular function call.
- A function's returned value is that of the last value in the function's body.
- Each instance of a function call in a script maintains its own, independent history.

### 3.11.2 Single-line functions

Simple functions can often be written in one line. This is the formal definition of single-line functions:

```
<function_declarator>
    <identifier>(<parameter_list>) => <return_value>

<parameter_list>
    {<parameter_definition>, <parameter_definition>} }

<parameter_definition>
    [<identifier> = <default_value>]

<return_value>
    <statement> | <expression> | <tuple>
```

Here is an example:

```
f(x, y) => x + y
```

After the function `f()` has been declared, it's possible to call it using different types of arguments:

```
a = f(open, close)
b = f(2, 2)
c = f(open, 2)
```

In the example above, the type of variable `a` is *series* because the arguments are both *series*. The type of variable `b` is *integer* because arguments are both *literal integers*. The type of variable `c` is *series* because the addition of a *series* and *literal integer* produces a *series* result.

### 3.11.3 Multi-line functions

Pine Script™ also supports multi-line functions with the following syntax:

```
<identifier>(<parameter_list>) =>
    <local_block>

<identifier>(<list of parameters>) =>
    <variable declaration>
    ...
    <variable declaration or expression>
```

where:

```
<parameter_list>
    {<parameter_definition>{, <parameter_definition>}}

<parameter_definition>
    [<identifier> = <default_value>]
```

The body of a multi-line function consists of several statements. Each statement is placed on a separate line and must be preceded by 1 indentation (4 spaces or 1 tab). The indentation before the statement indicates that it is a part of the body of the function and not part of the script's global scope. After the function's code, the first statement without an indent indicates the body of the function has ended.

Either an expression or a declared variable should be the last statement of the function's body. The result of this expression (or variable) will be the result of the function's call. For example:

```
geom_average(x, y) =>
    a = x*x
    b = y*y
    math.sqrt(a + b)
```

The function `geom_average` has two arguments and creates two variables in the body: `a` and `b`. The last statement calls the function `math.sqrt` (an extraction of the square root). The `geom_average` call will return the value of the last expression: `(math.sqrt(a + b))`.

### 3.11.4 Scopes in the script

Variables declared outside the body of a function or of other local blocks belong to the *global* scope. User-declared and built-in functions, as well as built-in variables also belong to the global scope.

Each function has its own *local* scope. All the variables declared within the function, as well as the function's arguments, belong to the scope of that function, meaning that it is impossible to reference them from outside — e.g., from the global scope or the local scope of another function.

On the other hand, since it is possible to refer to any variable or function declared in the global scope from the scope of a function (except for self-referencing recursive calls), one can say that the local scope is embedded into the global scope.

In Pine Script™, nested functions are not allowed, i.e., one cannot declare a function inside another one. All user functions are declared in the global scope. Local scopes cannot intersect with each other.

### 3.11.5 Functions that return multiple results

In most cases a function returns only one result, but it is possible to return a list of results (a *tuple*-like result):

```
fun(x, y) =>
    a = x+y
    b = x-y
    [a, b]
```

Special syntax is required for calling such functions:

```
[res0, res1] = fun(open, close)
plot(res0)
plot(res1)
```

### 3.11.6 Limitations

User-defined functions can use any of the Pine Script™ built-ins, except: `barcolor()`, `fill()`, `hline()`, `indicator()`, `library()`, `plot()`, `plotbar()`, `plotcandle()`, `plotchar()`, `plotshape()` and `strategy()`.



■ Advanced



## 3.12 Objects

- *Introduction*
- *Creating objects*
- *Changing field values*
- *Collecting objects*
- *Copying objects*
- *Shadowing*

---

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

---

### 3.12.1 Introduction

Pine Script™ objects are instances of *user-defined types* (UDTs). They are the equivalent of variables containing parts called *fields*, each able to hold independent values that can be of various types.

Experienced programmers can think of UDTs as methodless classes. They allow users to create custom types that organize different values under one logical entity.

### 3.12.2 Creating objects

Before an object can be created, its type must be defined. The [User-defined types](#) section of the [Type system](#) page explains how to do so.

Let's define a `pivotPoint` type to hold pivot information:

```
type pivotPoint
    int x
    float y
    string xloc = xloc.bar_time
```

Note that:

- We use the `type` keyword to declare the creation of a UDT.
- We name our new UDT `pivotPoint`.
- After the first line, we create a local block containing the type and name of each field.
- The `x` field will hold the x-coordinate of the pivot. It is declared as an “int” because it will hold either a timestamp or a bar index of “int” type.
- `y` is a “float” because it will hold the pivot’s price.
- `xloc` is a field that will specify the units of `x`: `xloc.bar_index` or `xloc.bar_time`. We set its default value to `xloc.bar_time` by using the `=` operator. When an object is created from that UDT, its `xloc` field will thus be set to that value.

Now that our `pivotPoint` UDT is defined, we can proceed to create objects from it. We create objects using the UDT's `new()` built-in method. To create a new `foundPoint` object from our `pivotPoint` UDT, we use:

```
foundPoint = pivotPoint.new()
```

We can also specify field values for the created object using the following:

```
foundPoint = pivotPoint.new(time, high)
```

Or the equivalent:

```
foundPoint = pivotPoint.new(x = time, y = high)
```

At this point, the `foundPoint` object's `x` field will contain the value of the `time` built-in when it is created, `y` will contain the value of `high` and the `xloc` field will contain its default value of `xloc.bar_time` because no value was defined for it when creating the object.

Object placeholders can also be created by declaring `na` object names using the following:

```
pivotPoint foundPoint = na
```

This example displays a label where high pivots are detected. The pivots are detected `legsInput` bars after they occur, so we must plot the label in the past for it to appear on the pivot:

```

1 // @version=5
2 indicator("Pivot labels", overlay = true)
3 int legsInput = input(10)
4
5 // Define the `pivotPoint` UDT.
6 type pivotPoint
7     int x
8     float y
9     string xloc = xloc.bar_time
10
11 // Detect high pivots.
12 pivotHighPrice = ta.pivothigh(legsInput, legsInput)
13 if not na(pivotHighPrice)
14     // A new high pivot was found; display a label where it occurred `legsInput` bars back.
15     foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
16     label.new(
17         foundPoint.x,
18         foundPoint.y,
19         str.tostring(foundPoint.y, format.mintick),
20         foundPoint.xloc,
21         textcolor = color.white)

```

Take note of this line from the above example:

```
foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
```

This could also be written using the following:

```
pivotPoint foundPoint = na
foundPoint := pivotPoint.new(time[legsInput], pivotHighPrice)
```

When using the `var` keyword while declaring a variable assigned to an object of a *user-defined type*, the keyword automatically applies to all the object's fields:

```

1 // @version=5
2 indicator("Objects using `var` demo")
3
4 // @type A custom type to hold index, price, and volume information.
5 type BarInfo
6     int index = bar_index
7     float price = close
8     float vol = volume
9
10 // @variable A `BarInfo` instance whose fields persist through all iterations, starting from the first bar.
11 var BarInfo firstBar = BarInfo.new()
12 // @variable A `BarInfo` instance declared on every bar.
13 BarInfo currentBar = BarInfo.new()
14
15 // Plot the `index` fields of both instances to compare the difference.
16 plot(firstBar.index)
17 plot(currentBar.index)

```

It's important to note that assigning an object to a variable that uses the `varip` keyword does *not* automatically allow the object's fields to persist without rolling back on each *intrabar* update. One must apply the keyword to each desired field in the type declaration to achieve this behavior. For example:

```

1 // @version=5
2 indicator("Objects using `varip` fields demo")
3
4 // @type A custom type that counts the bars and ticks in the script's execution.
5 type Counter
6     int      bars = 0
7     varip int ticks = 0
8
9 // @variable A `Counter` object whose reference persists throughout all bars.
10 var Counter counter = Counter.new()
11
12 // Add 1 to the `bars` and `ticks` fields. The `ticks` field is not subject to
13 // rollback on unconfirmed bars.
14 counter.bars += 1
15 counter.ticks += 1
16
17 // Plot both fields for comparison.
18 plot(counter.bars, "Bar counter", color.blue, 3)
19 plot(counter.ticks, "Tick counter", color.purple, 3)

```

**Note that:**

- We used the `var` keyword to specify that the `Counter` object assigned to the `counter` variable persists throughout the script's execution.
- The `bars` field rolls back on realtime bars, whereas the `ticks` field does not since we included `varip` in its declaration.

### 3.12.3 Changing field values

The value of an object's fields can be changed using the `:=` reassignment operator.

This line of our previous example:

```
foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
```

Could be written using the following:

```
foundPoint = pivotPoint.new()
foundPoint.x := time[legsInput]
foundPoint.y := pivotHighPrice
```

### 3.12.4 Collecting objects

Pine Script™ collections (`arrays`, `matrices`, and `maps`) can contain objects, allowing users to add virtual dimensions to their data structures. To declare a collection of objects, pass a UDT name into its `type template`.

This example declares an empty `array` that will hold objects of a `pivotPoint` user-defined type:

```
pivotHighArray = array.new<pivotPoint>()
```

To explicitly declare the type of a variable as an `array`, `matrix`, or `map` of a `user-defined type`, use the collection's type keyword followed by its `type template`. For example:

```
var array<pivotPoint> pivotHighArray = na
pivotHighArray := array.new<pivotPoint>()
```

Let's use what we have learned to create a script that detects high pivot points. The script first collects historical pivot information in an [array](#). It then loops through the array on the last historical bar, creating a label for each pivot and connecting the pivots with lines:



```

1 // @version=5
2 indicator("Pivot Points High", overlay = true)
3
4 int legsInput = input(10)
5
6 // Define the `pivotPoint` UDT containing the time and price of pivots.
7 type pivotPoint
8     int openTime
9     float level
10
11 // Create an empty `pivotPoint` array.
12 var pivotHighArray = array.new<pivotPoint>()
13
14 // Detect new pivots (`na` is returned when no pivot is found).
15 pivotHighPrice = ta.pivothigh(legsInput, legsInput)
16
17 // Add a new `pivotPoint` object to the end of the array for each detected pivot.
18 if not na(pivotHighPrice)
19     // A new pivot is found; create a new object of `pivotPoint` type, setting its
20     // `openTime` and `level` fields.
21     newPivot = pivotPoint.new(time[legsInput], pivotHighPrice)
22     // Add the new pivot object to the array.
23     array.push(pivotHighArray, newPivot)
24
25 // On the last historical bar, draw pivot labels and connecting lines.
26 if barstate.islastconfirmedhistory
27     var pivotPoint previousPoint = na
28     for eachPivot in pivotHighArray
29         // Display a label at the pivot point.
30         label.new(eachPivot.openTime, eachPivot.level, str.tostring(eachPivot.level,
31         //format.mintick), xloc.bar_time, textcolor = color.white)
32         // Create a line between pivots.
33         if not na(previousPoint)

```

(continues on next page)

(continued from previous page)

```

32         // Only create a line starting at the loop's second iteration because
33         // lines connect two pivots.
34         line.new(previousPoint.openTime, previousPoint.level, eachPivot.openTime,
35         //eachPivot.level, xloc = xloc.bar_time)
36         // Save the pivot for use in the next iteration.
37         previousPoint := eachPivot

```

### 3.12.5 Copying objects

In Pine, objects are assigned by reference. When an existing object is assigned to a new variable, both point to the same object.

In the example below, we create a `pivot1` object and set its `x` field to 1000. Then, we declare a `pivot2` variable containing the reference to the `pivot1` object, so both point to the same instance. Changing `pivot2.x` will thus also change `pivot1.x`, as both refer to the `x` field of the same object:

```

1 // @version=5
2 indicator("")
3 type pivotPoint
4     int x
5     float y
6 pivot1 = pivotPoint.new()
7 pivot1.x := 1000
8 pivot2 = pivot1
9 pivot2.x := 2000
10 // Both plot the value 2000.
11 plot(pivot1.x)
12 plot(pivot2.x)

```

To create a copy of an object that is independent of the original, we can use the built-in `copy()` method in this case.

In this example, we declare the `pivot2` variable referring to a copied instance of the `pivot1` object. Now, changing `pivot2.x` will not change `pivot1.x`, as it refers to the `x` field of a separate object:

```

1 // @version=5
2 indicator("")
3 type pivotPoint
4     int x
5     float y
6 pivot1 = pivotPoint.new()
7 pivot1.x := 1000
8 pivot2 = pivotPoint.copy(pivot1)
9 pivot2.x := 2000
10 // Plots 1000 and 2000.
11 plot(pivot1.x)
12 plot(pivot2.x)

```

It's important to note that the built-in `copy()` method produces a *shallow copy* of an object. If an object has fields with *special types* (`array`, `matrix`, `map`, `line`, `linefill`, `box`, `polyline`, `label`, `table`, or `chart.point`), those fields in a shallow copy of the object will point to the same instances as the original.

In the following example, we have defined an `InfoLabel` type with a `label` as one of its fields. The script instantiates a shallow copy of the parent object, then calls a user-defined `set()` *method* to update the `info` and `lbl` fields of each object. Since the `lbl` field of both objects points to the same `label` instance, changes to this field in either object affect the other:

```

1 //@version=5
2 indicator("Shallow Copy")
3
4 type InfoLabel
5     string info
6     label lbl
7
8 method set(InfoLabel this, int x = na, int y = na, string info = na) =>
9     if not na(x)
10        this.lbl.set_x(x)
11     if not na(y)
12        this.lbl.set_y(y)
13     if not na(info)
14        this.info := info
15        this.lbl.set_text(this.info)
16
17 var parent  = InfoLabel.new("", label.new(0, 0))
18 var shallow = parent.copy()
19
20 parent.set(bar_index, 0, "Parent")
21 shallow.set(bar_index, 1, "Shallow Copy")

```

To produce a *deep copy* of an object with all of its special type fields pointing to independent instances, we must explicitly copy those fields as well.

In this example, we have defined a `deepCopy()` method that instantiates a new `InfoLabel` object with its `lbl` field pointing to a copy of the original's field. Changes to the deep copy's `lbl` field will not affect the `parent` object, as it points to a separate instance:

```

1 //@version=5
2 indicator("Deep Copy")
3
4 type InfoLabel
5     string info
6     label lbl
7
8 method set(InfoLabel this, int x = na, int y = na, string info = na) =>
9     if not na(x)
10        this.lbl.set_x(x)
11     if not na(y)
12        this.lbl.set_y(y)
13     if not na(info)
14        this.info := info
15        this.lbl.set_text(this.info)
16
17 method deepCopy(InfoLabel this) =>
18     InfoLabel.new(this.info, this.lbl.copy())
19
20 var parent = InfoLabel.new("", label.new(0, 0))
21 var deep   = parent.deepCopy()
22
23 parent.set(bar_index, 0, "Parent")
24 deep.set(bar_index, 1, "Deep Copy")

```

### 3.12.6 Shadowing

To avoid potential conflicts in the eventuality where namespaces added to Pine Script™ in the future would collide with UDTs or object names in existing scripts; as a rule, UDTs and object names shadow the language's namespaces. For example, a UDT or object can use the name of built-in types, such as `line` or `table`.

Only the language's five primitive types cannot be used to name UDTs or objects: `int`, `float`, `string`, `bool`, and `color`.



Advanced



## 3.13 Methods

- *Introduction*
- *Built-in methods*
- *User-defined methods*
- *Method overloading*
- *Advanced example*

---

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

---

### 3.13.1 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*.

### 3.13.2 Built-in methods

Pine Script™ includes built-in methods for all *special types*, including `array`, `matrix`, `map`, `line`, `linefill`, `box`, `polyline`, `label`, and `table`. 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`:



```

1 // @version=5
2 indicator("Custom Sample BB", overlay = true)
3
4 float sourceInput = input.source(close, "Source")
5 int samplesInput = input.int(20, "Samples")
6 int n = input.int(10, "Bars")
7 float multiplier = input.float(2.0, "StdDev")
8
9 var array<float> sourceArray = array.new<float>(samplesInput)
10 var float sampleMean = na
11 var float sampleDev = na
12
13 // Identify if `n` bars have passed.
14 if bar_index % n == 0
15     // Update the queue.
16     array.push(sourceArray, sourceInput)
17     array.shift(sourceArray)

```

(continues on next page)

(continued from previous page)

```

18     // Update the mean and standard deviaiton values.
19     sampleMean := array.avg(sourceArray)
20     sampleDev  := array.stdev(sourceArray) * multiplier
21
22 // Calculate bands.
23 float highBand = sampleMean + sampleDev
24 float lowBand  = sampleMean - sampleDev
25
26 plot(sampleMean, "Basis", color.orange)
27 plot(highBand, "Upper", color.lime)
28 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:

```

1 //@version=5
2 indicator("Custom Sample BB", overlay = true)
3
4 float sourceInput  = input.source(close, "Source")
5 int   samplesInput = input.int(20, "Samples")
6 int   n            = input.int(10, "Bars")
7 float multiplier   = input.float(2.0, "StdDev")
8
9 var array<float> sourceArray = array.new<float>(samplesInput)
10 var float        sampleMean  = na
11 var float        sampleDev   = na
12
13 // Identify if `n` bars have passed.
14 if bar_index % n == 0
15     // Update the queue.
16     sourceArray.push(sourceInput)
17     sourceArray.shift()
18     // Update the mean and standard deviaiton values.
19     sampleMean := sourceArray.avg()
20     sampleDev  := sourceArray.stdev() * multiplier
21
22 // Calculate band values.
23 float highBand = sampleMean + sampleDev
24 float lowBand  = sampleMean - sampleDev
25
26 plot(sampleMean, "Basis", color.orange)
27 plot(highBand, "Upper", color.lime)
28 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.

### 3.13.3 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:

```
1 // Identify if `n` bars have passed.
2 if bar_index % n == 0
3     // Update the queue.
4     sourceArray.push(sourceInput)
5     sourceArray.shift()
6     // Update the mean and standard deviation values.
7     sampleMean := sourceArray.avg()
8     sampleDev  := sourceArray.stdev() * multiplier
9
10 // Calculate band values.
11 float highBand = sampleMean + sampleDev
12 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:

```
1 // @function      Maintains a queue of the size of `srcArray`.
2 //              It appends a `value` to the array and removes its oldest element.
3 //              ↪at position zero.
4 // @param srcArray (array<float>) The array where the queue is maintained.
5 // @param value    (float) The new value to be added to the queue.
6 //              The queue's oldest value is also removed, so its size is
7 //              ↪constant.
8 // @param takeSample (bool) A new `value` is only pushed into the queue if this is
9 //              ↪true.
10 // @returns        (array<float>) `srcArray` object.
11 method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
12     if takeSample
13         srcArray.push(value)
14         srcArray.shift()
15     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:

```

1 // Identify if `n` bars have passed.
2 if bar_index % n == 0
3     // Update the queue.
4     sourceArray.maintainQueue(sourceInput)
5     // Update the mean and standard deviation values.
6     sampleMean := sourceArray.avg()
7     sampleDev  := sourceArray.stdev() * multiplier
8
9 // Calculate band values.
10 float highBand = sampleMean + sampleDev
11 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:

```

1 // @function      Computes Bollinger Band values from an array of data.
2 // @param srcArray (array<float>) The array where the queue is maintained.
3 // @param multiplier (float) Standard deviation multiplier.
4 // @param calculate (bool) The method will only calculate new values when this is true.
5 // @returns        A tuple containing the basis, upper band, and lower band respectively.
6 method calcBB(array<float> srcArray, float mult, bool calculate = true) =>
7     var float mean = na
8     var float dev = na
9     if calculate
10         // Compute the mean and standard deviation of the array.
11         mean := srcArray.avg()
12         dev  := srcArray.stdev() * mult
13     [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).
calcBB(multiplier, newSample)

```

#### 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:

```

1 // @version=5
2 indicator("Custom Sample BB", overlay = true)
3

```

(continues on next page)

(continued from previous page)

```

4 float sourceInput = input.source(close, "Source")
5 int samplesInput = input.int(20, "Samples")
6 int n = input.int(10, "Bars")
7 float multiplier = input.float(2.0, "StdDev")
8
9 var array<float> sourceArray = array.new<float>(samplesInput)
10
11 // @function Maintains a queue of the size of `srcArray`.
12 // It appends a `value` to the array and removes its oldest element ↵
13 // at position zero.
14 // @param srcArray (array<float>) The array where the queue is maintained.
15 // @param value (float) The new value to be added to the queue.
16 // The queue's oldest value is also removed, so its size is ↵
17 // constant.
18 // @param takeSample (bool) A new `value` is only pushed into the queue if this is ↵
19 // true.
20 // @returns (array<float>) `srcArray` object.
21 method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
22     if takeSample
23         srcArray.push(value)
24         srcArray.shift()
25     srcArray
26
27 // @function Computes Bollinger Band values from an array of data.
28 // @param srcArray (array<float>) The array where the queue is maintained.
29 // @param multiplier (float) Standard deviation multiplier.
30 // @param calculate (bool) The method will only calculate new values when this is ↵
31 // true.
32 // @returns A tuple containing the basis, upper band, and lower band ↵
33 // respectively.
34 method calcBB(array<float> srcArray, float mult, bool calculate = true) =>
35     var float mean = na
36     var float dev = na
37     if calculate
38         // Compute the mean and standard deviation of the array.
39         mean := srcArray.avg()
40         dev := srcArray.stdev() * mult
41         [mean, mean + dev, mean - dev]
42
43 // Identify if `n` bars have passed.
44 bool newSample = bar_index % n == 0
45
46 // Update the queue and compute new BB values on each new sample.
47 [sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput, newSample). ↵
48 calcBB(multiplier, newSample)
49
50 plot(sampleMean, "Basis", color.orange)
51 plot(highBand, "Upper", color.lime)
52 plot(lowBand, "Lower", color.red)

```

### 3.13.4 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:

```

1 // @function Identifies an object's type.
2 // @param this Object to inspect.
3 // @returns (string) A string representation of the type.
4 method getType(int this) =>
5     na(this) ? "int(na)" : "int"
6
7 method getType(float this) =>
8     na(this) ? "float(na)" : "float"
9
10 method getType(bool this) =>
11     na(this) ? "bool(na)" : "bool"
12
13 method getType(color this) =>
14     na(this) ? "color(na)" : "color"
15
16 method getType(string this) =>
17     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:



```

1 // @version=5
2 indicator("Type Inspection")
3
4 // @function Identifies an object's type.
5 // @param this Object to inspect.
6 // @returns (string) A string representation of the type.
7 method getType(int this) =>
8     na(this) ? "int(na)" : "int"
9
10 method getType(float this) =>
```

(continues on next page)

(continued from previous page)

```

11     na(this) ? "float(na)" : "float"
12
13 method getType(bool this) =>
14     na(this) ? "bool(na)" : "bool"
15
16 method getType(color this) =>
17     na(this) ? "color(na)" : "color"
18
19 method getType(string this) =>
20     na(this) ? "string(na)" : "string"
21
22 a = 1
23 b = 1.0
24 c = true
25 d = color.white
26 e = "1"
27
28 // Inspect variables and format results.
29 results = str.format(
30     "a: {0}\nb: {1}\nc: {2}\nd: {3}\n\n: {4}",
31     a.getType(), b.getType(), c.getType(), d.getType(), e.getType()
32 )
33
34 var label lbl = label.new(0, 0)
35 lbl.set_x(bar_index)
36 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.

### 3.13.5 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`:

```

1 // @function      Replaces elements in a `srcArray` between `lowerBound` and_
2 //                `upperBound` with an `innerValue`,_
3 //                and replaces elements outside the range with an `outerValue`.
4 // @param srcArray (array<float>) Array to modify.
5 // @param innerValue (float) Value to replace elements within the range with.
6 // @param outerValue (float) Value to replace elements outside the range with.
7 // @param lowerBound (float) Lowest value to replace with `innerValue`.
8 // @param upperBound (float) Highest value to replace with `innerValue`.
9 // @returns        (array<float>) `srcArray` object.
method fill(array<float> srcArray, float innerValue, float outerValue, float_
    lowerBound, float upperBound) =>

```

(continues on next page)

(continued from previous page)

```

10   for [i, element] in srcArray
11     if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or_
12       ↪na(upperBound))
13       srcArray.set(i, innerValue)
14     else
15       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`:

```

1 // @function      Estimates the empirical CDF of a `srcArray`.
2 // @param srcArray (array<float>) Array to calculate on.
3 // @param steps    (int) Number of steps in the estimation.
4 // @returns        (array<float>) Array of estimated CDF ratios.
5 method eCDF(array<float> srcArray, int steps) =>
6   float min = srcArray.min()
7   float rng = srcArray.range() / steps
8   array<float> cdfArray = array.new<float>()
9   // Add averages of `srcArray` filtered by value region to the `cdfArray`.
10  float val = min
11  for i = 1 to steps
12    val += rng
13    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:

```

1 // @function      Rescales the elements within a `srcArray` to the interval [0, 1].
2 // @param srcArray (array<float>) Array to normalize.
3 // @returns        (array<float>) Normalized copy of the `srcArray`.
4 method featureScale(array<float> srcArray) =>
5   float min = srcArray.min()

```

(continues on next page)

(continued from previous page)

```

6   float rng = srcArray.range()
7   array<float> scaledArray = array.new<float>()
8   // Push normalized `element` values into the `scaledArray`.
9   for element in srcArray
10      scaledArray.push((element - min) / rng)
11

```

### 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:



```

1 // @version=5
2 indicator("Empirical Distribution", overlay = true)
3
4 float sourceInput = input.source(close, "Source")
5 int length      = input.int(20, "Length")
6 int n           = input.int(20, "Steps")
7
8 // @function      Maintains a queue of the size of `srcArray`.
9 //               It appends a `value` to the array and removes its oldest element.
10 // @param srcArray (array<float>) The array where the queue is maintained.
11 // @param value    (float) The new value to be added to the queue.
12 //               The queue's oldest value is also removed, so its size is
13 //               constant.
14 // @param takeSample (bool) A new `value` is only pushed into the queue if this is
15 //               true.
16 // @returns        (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
    if takeSample

```

(continues on next page)

(continued from previous page)

```

17     srcArray.push(value)
18     srcArray.shift()
19     srcArray
20
21 // @function      Replaces elements in a `srcArray` between `lowerBound` and_
22 //                  `upperBound` with an `innerValue`,_
23 //                  and replaces elements outside the range with an `outerValue`.
24 // @param srcArray (array<float>) Array to modify.
25 // @param innerValue (float) Value to replace elements within the range with.
26 // @param outerValue (float) Value to replace elements outside the range with.
27 // @param lowerBound (float) Lowest value to replace with `innerValue`.
28 // @param upperBound (float) Highest value to replace with `innerValue`.
29 // @returns         (array<float>) `srcArray` object.
30 method fill(array<float> srcArray, float innerValue, float outerValue, float_
31            ↵lowerBound, float upperBound) =>
32     for [i, element] in srcArray
33         if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or_
34            ↵na(upperBound))
35             srcArray.set(i, innerValue)
36         else
37             srcArray.set(i, outerValue)
38     srcArray
39
40 // @function      Estimates the empirical CDF of a `srcArray`.
41 // @param srcArray (array<float>) Array to calculate on.
42 // @param steps     (int) Number of steps in the estimation.
43 // @returns         (array<float>) Array of estimated CDF ratios.
44 method eCDF(array<float> srcArray, int steps) =>
45     float min = srcArray.min()
46     float rng = srcArray.range() / steps
47     array<float> cdfArray = array.new<float>()
48     // Add averages of `srcArray` filtered by value region to the `cdfArray`.
49     float val = min
50     for i = 1 to steps
51         val += rng
52         cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg())
53     cdfArray
54
55 // @function      Rescales the elements within a `srcArray` to the interval [0, 1].
56 // @param srcArray (array<float>) Array to normalize.
57 // @returns         (array<float>) Normalized copy of the `srcArray`.
58 method featureScale(array<float> srcArray) =>
59     float min = srcArray.min()
60     float rng = srcArray.range()
61     array<float> scaledArray = array.new<float>()
62     // Push normalized `element` values into the `scaledArray`.
63     for element in srcArray
64         scaledArray.push((element - min) / rng)
65     scaledArray
66
67 // @function      Draws a label containing eCDF estimates in the format "{price}:
68 //                  ↵{percent}%"
69 // @param srcArray (array<float>) Array of source values.
70 // @param cdfArray (array<float>) Array of CDF estimates.
71 // @returns         (void)
72 makeLabel(array<float> srcArray, array<float> cdfArray) =>
73     float max      = srcArray.max()

```

(continues on next page)

(continued from previous page)

```

70     float rng      = srcArray.range() / cdfArray.size()
71     string results = ""
72     var label lbl  = label.new(0, 0, "", style = label.style_label_left, text_font_
73     ↪family = font.family_monospace)
74     // Add percentage strings to `results` starting from the `max`.
75     cdfArray.reverse()
76     for [i, element] in cdfArray
77         results += str.format("{0}: {1}%", max - i * rng, element * 100)
78     // Update `lbl` attributes.
79     lbl.set_xy(bar_index + 1, srcArray.avg())
80     lbl.set_text(results)
81
82 var array<float> sourceArray = array.new<float>(length)
83
84 // Add background color for the last `length` bars.
85 bgcolor(bar_index > last_bar_index - length ? color.new(color.orange, 80) : na)
86
87 // Queue `sourceArray`, feature scale, then estimate the distribution over `n` steps.
88 array<float> distArray = sourceArray.maintainQueue(sourceInput).featureScale().eCDF(n)
89 // Draw label.
90 makeLabel(sourceArray, distArray)

```



Advanced



## 3.14 Arrays

- *Introduction*
- *Declaring arrays*
- *Reading and writing array elements*
- *Looping through array elements*
- *Scope*
- *History referencing*
- *Inserting and removing array elements*
- *Calculations on arrays*
- *Manipulating arrays*
- *Searching arrays*
- *Error handling*

---

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

---

### 3.14.1 Introduction

Pine Script™ Arrays are one-dimensional collections that can hold multiple value references. Think of them as a better way to handle cases where one would otherwise need to explicitly declare a set of similar variables (e.g., `price00`, `price01`, `price02`, ...).

All elements within an array must be of the same type, which can be a [built-in](#) or a [user-defined type](#), always qualified as “series”. Scripts reference arrays using an array ID similar to the IDs of lines, labels, and other special types. Pine Script™ does not use an indexing operator to reference individual array elements. Instead, functions including `array.get()` and `array.set()` read and write the values of array elements. We can use array values in expressions and functions that allow “series” values.

Scripts reference the elements of an array using an *index*, which starts at 0 and extends to the number of elements in the array minus one. Arrays in Pine Script™ can have a dynamic size that varies across bars, as one can change the number of elements in an array on each iteration of a script. Scripts can contain multiple array instances. The size of arrays is limited to 100,000 elements.

---

**Note:** We will use *beginning* of an array to designate index 0, and *end* of an array to designate the array’s element with the highest index value. We will also extend the meaning of *array* to include array IDs, for the sake of brevity.

---

### 3.14.2 Declaring arrays

Pine Script™ uses the following syntax to declare arrays:

```
[var/varip ][array<type>/<type[]> ]<identifier> = <expression>
```

Where `<type>` is a [type template](#) for the array that declares the type of values it will contain, and the `<expression>` returns either an array of the specified type or `na`.

When declaring a variable as an array, we can use the `array` keyword followed by a [type template](#). Alternatively, we can use the `type` name followed by the `[]` modifier (not to be confused with the [\[\] history-referencing operator](#)).

Since Pine always uses type-specific functions to create arrays, the `array<type>/type[]` part of the declaration is redundant, except when declaring an array variable assigned to `na`. Even when not required, explicitly declaring the array type helps clearly state the intention to readers.

This line of code declares an array variable named `prices` that points to `na`. In this case, we must specify the type to declare that the variable can reference arrays containing “float” values:

```
array<float> prices = na
```

We can also write the above example in this form:

```
float[] prices = na
```

When declaring an array and the `<expression>` is not `na`, use one of the following functions: `array.new<type>(size, initial_value)`, `array.from()`, or `array.copy()`. For `array.new<type>(size, initial_value)` functions, the arguments of the `size` and `initial_value` parameters can be “series” to allow dynamic sizing and initialization

of array elements. The following example creates an array containing zero “float” elements, and this time, the array ID returned by the `array.new<float>()` function call is assigned to `prices`:

```
prices = array.new<float>(0)
```

**Note:** The `array.*` namespace also contains type-specific functions for creating arrays, including `array.new_int()`, `array.new_float()`, `array.new_bool()`, `array.new_color()`, `array.new_string()`, `array.new_line()`, `array.new_linefill()`, `array.new_label()`, `array.new_box()` and `array.new_table()`. The `array.new<type>()` function can create an array of any type, including *user-defined types*.

---

The `initial_value` parameter of `array.new*` functions allows users to set all elements in the array to a specified value. If no argument is provided for `initial_value`, the array is filled with `na` values.

This line declares an array ID named `prices` pointing to an array containing two elements, each assigned to the bar’s `close` value:

```
prices = array.new<float>(2, close)
```

To create an array and initialize its elements with different values, use `array.from()`. This function infers the array’s size and the type of elements it will hold from the arguments in the function call. As with `array.new*` functions, it accepts “series” arguments. All values supplied to the function must be of the same type.

For example, all three of these lines of code will create identical “bool” arrays with the same two elements:

```
statesArray = array.from(close > open, high != close)
bool[] statesArray = array.from(close > open, high != close)
array<bool> statesArray = array.from(close > open, high != close)
```

### Using `var` and `varip` keywords

Users can utilize `var` and `varip` keywords to instruct a script to declare an array variable only once on the first iteration of the script on the first chart bar. Array variables declared using these keywords point to the same array instances until explicitly reassigned, allowing an array and its element references to persist across bars.

When declaring an array variable using these keywords and pushing a new value to the end of the referenced array on each bar, the array will grow by one on each bar and be of size `bar_index + 1` (`bar_index` starts at zero) by the time the script executes on the last bar, as this code demonstrates:

```
1 // @version=5
2 indicator("Using `var`")
3 // @variable An array that expands its size by 1 on each bar.
4 var a = array.new<float>(0)
5 array.push(a, close)
6
7 if barstate.islast
8     // @variable A string containing the size of `a` and the current `bar_index` value.
9     string labelText = "Array size: " + str.tostring(a.size()) + "\nbar_index: " +
10        str.tostring(bar_index)
11    // Display the `labelText`.
12    label.new(bar_index, 0, labelText, size = size.large)
```

The same code without the `var` keyword would re-declare the array on each bar. In this case, after execution of the `array.push()` call, the `a.size()` call would return a value of 1.

**Note:** Array variables declared using `varip` behave as ones using `var` on historical data, but they update their values for realtime bars (i.e., the bars since the script's last compilation) on each new price tick. Arrays assigned to `varip` variables can only hold `int`, `float`, `bool`, `color`, or `string` types or *user-defined types* that exclusively contain within their fields these types or collections (arrays, *matrices*, or *maps*) of these types.

### 3.14.3 Reading and writing array elements

Scripts can write values to existing individual array elements using `array.set(id, index, value)`, and read using `array.get(id, index)`. When using these functions, it is imperative that the `index` in the function call is always less than or equal to the array's size (because array indices start at zero). To get the size of an array, use the `array.size(id)` function.

The following example uses the `set()` method to populate a `fillColors` array with instances of one base color using different transparency levels. It then uses `array.get()` to retrieve one of the colors from the array based on the location of the bar with the highest price within the last `lookbackInput` bars:



```

1 // @version=5
2 indicator("Distance from high", "", true)
3 lookbackInput = input.int(100)
4 FILL_COLOR = color.green
5 // Declare array and set its values on the first bar only.
6 var fillColors = array.new<color>(5)
7 if barstate.isfirst
8     // Initialize the array elements with progressively lighter shades of the fill
9     ↪ color.
10    fillColors.set(0, color.new(FILL_COLOR, 70))
11    fillColors.set(1, color.new(FILL_COLOR, 75))
12    fillColors.set(2, color.new(FILL_COLOR, 80))
13    fillColors.set(3, color.new(FILL_COLOR, 85))
14    fillColors.set(4, color.new(FILL_COLOR, 90))
15
16 // Find the offset to highest high. Change its sign because the function returns a
17 ↪ negative value.
18 lastHiBar = - ta.highestbars(high, lookbackInput)
19 // Convert the offset to an array index, capping it to 4 to avoid a runtime error.
20 // The index used by `array.get()` will be the equivalent of `floor(fillNo)`.
21 fillNo = math.min(lastHiBar / (lookbackInput / 5), 4)
22 // Set background to a progressively lighter fill with increasing distance from
23 ↪ location of highest high.

```

(continues on next page)

(continued from previous page)

```

21 bgcolor(array.get(fillColors, fillNo))
22 // Plot key values to the Data Window for debugging.
23 plotchar(lastHiBar, "lastHiBar", "", location.top, size = size.tiny)
24 plotchar(fillNo, "fillNo", "", location.top, size = size.tiny)

```

Another technique for initializing the elements in an array is to create an *empty array* (an array with no elements), then use `array.push()` to append **new** elements to the end of the array, increasing the size of the array by one on each call. The following code is functionally identical to the initialization section from the preceding script:

```

// Declare array and set its values on the first bar only.
var fillColors = array.new<color>(0)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill
    ↵color.
    array.push(fillColors, color.new(FILL_COLOR, 70))
    array.push(fillColors, color.new(FILL_COLOR, 75))
    array.push(fillColors, color.new(FILL_COLOR, 80))
    array.push(fillColors, color.new(FILL_COLOR, 85))
    array.push(fillColors, color.new(FILL_COLOR, 90))

```

This code is equivalent to the one above, but it uses `array.unshift()` to insert new elements at the *beginning* of the `fillColors` array:

```

// Declare array and set its values on the first bar only.
var fillColors = array.new<color>(0)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill
    ↵color.
    array.unshift(fillColors, color.new(FILL_COLOR, 90))
    array.unshift(fillColors, color.new(FILL_COLOR, 85))
    array.unshift(fillColors, color.new(FILL_COLOR, 80))
    array.unshift(fillColors, color.new(FILL_COLOR, 75))
    array.unshift(fillColors, color.new(FILL_COLOR, 70))

```

We can also use `array.from()` to create the same `fillColors` array with a single function call:

```

1 // @version=5
2 indicator("Using `var`")
3 FILL_COLOR = color.green
4 var array<color> fillColors = array.from(
5     color.new(FILL_COLOR, 70),
6     color.new(FILL_COLOR, 75),
7     color.new(FILL_COLOR, 80),
8     color.new(FILL_COLOR, 85),
9     color.new(FILL_COLOR, 90)
10 )
11 // Cycle background through the array's colors.
12 bgcolor(array.get(fillColors, bar_index % (fillColors.size())))

```

The `array.fill(id, value, index_from, index_to)` function points all array elements, or the elements within the `index_from` to `index_to` range, to a specified value. Without the last two optional parameters, the function fills the whole array, so:

```
a = array.new<float>(10, close)
```

and:

```
a = array.new<float>(10)
a.fill(close)
```

are equivalent, but:

```
a = array.new<float>(10)
a.fill(close, 1, 3)
```

only fills the second and third elements (at index 1 and 2) of the array with `close`. Note how `array.fill()`'s last parameter, `index_to`, must be one greater than the last index the function will fill. The remaining elements will hold `na` values, as the `array.new()` function call does not contain an `initial_value` argument.

### 3.14.4 Looping through array elements

When looping through an array's element indices and the array's size is unknown, one can use the `array.size()` function to get the maximum index value. For example:

```
1 // @version=5
2 indicator("Protected `for` loop", overlay = true)
3 // @variable An array of `close` prices from the 1-minute timeframe.
4 array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)
5
6 // @variable A string representation of the elements in `a`.
7 string labelText = ""
8 for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
9     labelText += str.tostring(array.get(a, i)) + "\n"
10
11 label.new(bar_index, high, text = labelText)
```

#### Note that:

- We use the `request.security_lower_tf()` function which returns an array of `close` prices at the 1 minute timeframe.
- This code example will throw an error if you use it on a chart timeframe smaller than 1 minute.
- `for` loops do not execute if the `to` expression is `na`. Note that the `to` value is only evaluated once upon entry.

An alternative method to loop through an array is to use a `for...in` loop. This approach is a variation of the standard for loop that can iterate over the value references and indices in an array. Here is an example of how we can write the code example from above using a `for...in` loop:

```
1 // @version=5
2 indicator(`for...in` loop, overlay = true)
3 // @variable An array of `close` prices from the 1-minute timeframe.
4 array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)
5
6 // @variable A string representation of the elements in `a`.
7 string labelText = ""
8 for price in a
9     labelText += str.tostring(price) + "\n"
10
11 label.new(bar_index, high, text = labelText)
```

#### Note that:

- `for...in` loops can return a tuple containing each index and corresponding element. For example, `for [i, price] in a` returns the `i` index and `price` value for each element in `a`.

A `while` loop statement can also be used:

```

1 // @version=5
2 indicator(`while` loop", overlay = true)
3 array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)
4
5 string labelText = ""
6 int i = 0
7 while i < array.size(a)
8     labelText += str.tostring(array.get(a, i)) + "\n"
9     i += 1
10
11 label.new(bar_index, high, text = labelText)

```

### 3.14.5 Scope

Users can declare arrays within the global scope of a script, as well as the local scopes of *functions*, *methods*, and *conditional structures*. Unlike some of the other built-in types, namely *fundamental* types, scripts can modify globally-assigned arrays from within local scopes, allowing users to implement global variables that any function in the script can directly interact with. We use the functionality here to calculate progressively lower or higher price levels:



```

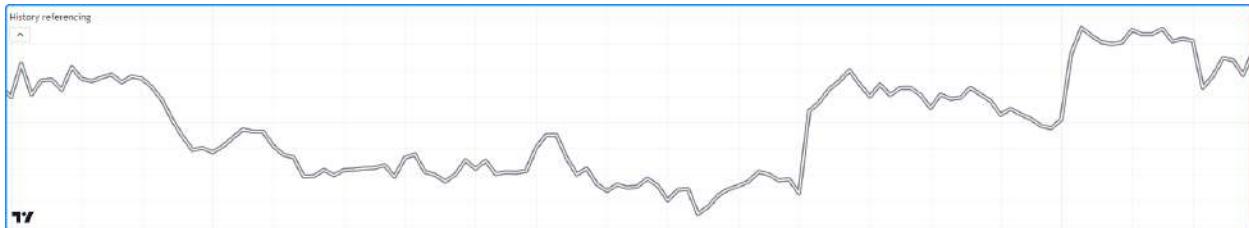
1 // @version=5
2 indicator("Bands", "", true)
3 // @variable The distance ratio between plotted price levels.
4 factorInput = 1 + (input.float(-2., "Step %") / 100)
5 // @variable A single-value array holding the lowest `ohlc4` value within a 50 bar
6 // window from 10 bars back.
7 level = array.new<float>(1, ta.lowest(ohlc4, 50)[10])
8
9 nextLevel(val) =>
10     newLevel = level.get(0) * val
11     // Write new level to the global `level` array so we can use it as the base in
12     // the next function call.
13     level.set(0, newLevel)
14     newLevel
15
16 plot(nextLevel(1))
17 plot(nextLevel(factorInput))
18 plot(nextLevel(factorInput))
19 plot(nextLevel(factorInput))

```

### 3.14.6 History referencing

Pine Script™'s history-referencing operator [ ] can access the history of array variables, allowing scripts to interact with past array instances previously assigned to a variable.

To illustrate this, let's create a simple example to show how one can fetch the previous bar's `close` value in two equivalent ways. This script uses the [ ] operator to get the array instance assigned to `a` on the previous bar, then uses the `get()` method to retrieve the value of the first element (`previousClose1`). For `previousClose2`, we use the history-referencing operator on the `close` variable directly to retrieve the value. As we see from the plots, `previousClose1` and `previousClose2` both return the same value:



```

1 //@version=5
2 indicator("History referencing")
3
4 //@variable A single-value array declared on each bar.
5 a = array.new<float>(1)
6 // Set the value of the only element in `a` to `close`.
7 array.set(a, 0, close)
8
9 //@variable The array instance assigned to `a` on the previous bar.
10 previous = a[1]
11
12 previousClose1 = na(previous) ? na : previous.get(0)
13 previousClose2 = close[1]
14
15 plot(previousClose1, "previousClose1", color.gray, 6)
16 plot(previousClose2, "previousClose2", color.white, 2)

```

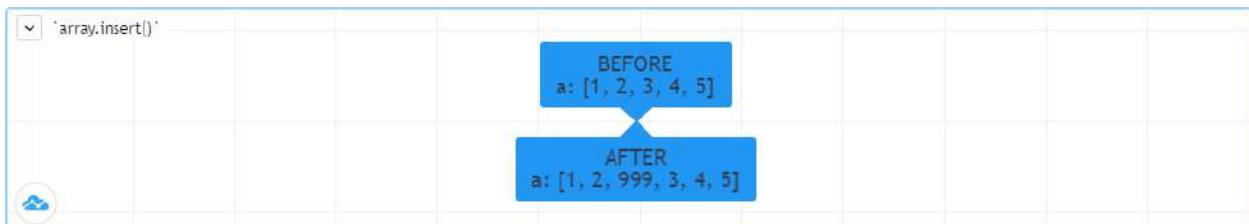
### 3.14.7 Inserting and removing array elements

#### Inserting

The following three functions can insert new elements into an array.

`array.unshift()` inserts a new element at the beginning of an array (index 0) and increases the index values of any existing elements by one.

`array.insert()` inserts a new element at the specified `index` and increases the index of existing elements at or after the `index` by one.



```

1 // @version=5
2 indicator(`array.insert()`)
3 a = array.new<float>(5, 0)
4 for i = 0 to 4
5     array.set(a, i, i + 1)
6 if barstate.islast
7     label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a), size = size.large)
8     array.insert(a, 2, 999)
9     label.new(bar_index, 0, "AFTER\na: " + str.tostring(a), style = label.style_label_
→up, size = size.large)

```

`array.push()` adds a new element at the end of an array.

### Removing

These four functions remove elements from an array. The first three also return the value of the removed element.

`array.remove()` removes the element at the specified `index` and returns that element's value.

`array.shift()` removes the first element from an array and returns its value.

`array.pop()` removes the last element of an array and returns its value.

`array.clear()` removes all elements from an array. Note that clearing an array won't delete any objects its elements referenced. See the example below that illustrates how this works:

```

1 // @version=5
2 indicator(`array.clear()` example, overlay = true)
3
4 // Create a label array and add a label to the array on each new bar.
5 var a = array.new<label>()
6 label lbl = label.new(bar_index, high, "Text", color = color.red)
7 array.push(a, lbl)
8
9 var table t = table.new(position.top_right, 1, 1)
10 // Clear the array on the last bar. This doesn't remove the labels from the chart.
11 if barstate.islast
12     array.clear(a)
13     table.cell(t, 0, 0, "Array elements count: " + str.tostring(array.size(a)), ←
→bgcolor = color.yellow)

```

### Using an array as a stack

Stacks are LIFO (last in, first out) constructions. They behave somewhat like a vertical pile of books to which books can only be added or removed one at a time, always from the top. Pine Script™ arrays can be used as a stack, in which case we use the `array.push()` and `array.pop()` functions to add and remove elements at the end of the array.

`array.push(prices, close)` will add a new element to the end of the `prices` array, increasing the array's size by one.

`array.pop(prices)` will remove the end element from the `prices` array, return its value and decrease the array's size by one.

See how the functions are used here to track successive lows in rallies:



```

1 // @version=5
2 indicator("Lows from new highs", "", true)
3 var lows = array.new<float>(0)
4 flushLows = false
5
6 // Remove last element from the stack when `_cond` is true.
7 array_pop(id, cond) => cond and array.size(id) > 0 ? array.pop(id) : float(na)
8
9 if ta.rising(high, 1)
10    // Rising highs; push a new low on the stack.
11    lows.push(low)
12    // Force the return type of this `if` block to be the same as that of the next
13    // block.
14    bool(na)
15 else if lows.size() >= 4 or low < array.min(lows)
16    // We have at least 4 lows or price has breached the lowest low;
17    // sort lows and set flag indicating we will plot and flush the levels.
18    array.sort(lows, order.ascending)
19    flushLows := true
20
21 // If needed, plot and flush lows.
22 lowLevel = array_pop(lows, flushLows)
23 plot(lowLevel, "Low 1", low > lowLevel ? color.silver : color.purple, 2, plot.style_
24    ↪ linebr)
25 lowLevel := array_pop(lows, flushLows)
26 plot(lowLevel, "Low 2", low > lowLevel ? color.silver : color.purple, 3, plot.style_
27    ↪ linebr)
28 lowLevel := array_pop(lows, flushLows)
29 plot(lowLevel, "Low 3", low > lowLevel ? color.silver : color.purple, 4, plot.style_
30    ↪ linebr)
31 lowLevel := array_pop(lows, flushLows)
32 plot(lowLevel, "Low 4", low > lowLevel ? color.silver : color.purple, 5, plot.style_
33    ↪ linebr)
34
35 if flushLows
36    // Clear remaining levels after the last 4 have been plotted.
37    lows.clear()

```

### Using an array as a queue

Queues are FIFO (first in, first out) constructions. They behave somewhat like cars arriving at a red light. New cars are queued at the end of the line, and the first car to leave will be the first one that arrived to the red light.

In the following code example, we let users decide through the script's inputs how many labels they want to have on their chart. We use that quantity to determine the size of the array of labels we then create, initializing the array's elements to na.

When a new pivot is detected, we create a label for it, saving the label's ID in the pLabel variable. We then queue the ID of that label by using `array.push()` to append the new label's ID to the end of the array, making our array size one greater than the maximum number of labels to keep on the chart.

Lastly, we de-queue the oldest label by removing the array's first element using `array.shift()` and deleting the label referenced by that array element's value. As we have now de-queued an element from our queue, the array contains `pivotCountInput` elements once again. Note that on the dataset's first bars we will be deleting na label IDs until the maximum number of labels has been created, but this does not cause runtime errors. Let's look at our code:



```

1 // @version=5
2 MAX_LABELS = 100
3 indicator("Show Last n High Pivots", "", true, max_labels_count = MAX_LABELS)
4
5 pivotCountInput = input.int(5, "How many pivots to show", minval = 0, maxval = MAX_
6 ↵LABELS)
7 pivotLegsInput = input.int(3, "Pivot legs", minval = 1, maxval = 5)
8
9 // Create an array containing the user-selected max count of label IDs.
10 var labelIds = array.new<label>(pivotCountInput)
11
12 pH = ta.pivothigh(pivotLegsInput, pivotLegsInput)
13 if not na(pH)
14     // New pivot found; plot its label `i_pivotLegs` bars back.
15     pLabel = label.new(bar_index[pivotLegsInput], pH, str.tostring(pH, format.
16 ↵mintick), textcolor = color.white)
17     // Queue the new label's ID by appending it to the end of the array.
18     array.push(labelIds, pLabel)
19     // De-queue the oldest label ID from the queue and delete the corresponding label.
20     label.delete(array.shift(labelIds))

```

### 3.14.8 Calculations on arrays

While series variables can be viewed as a horizontal set of values stretching back in time, Pine Script™'s one-dimensional arrays can be viewed as vertical structures residing on each bar. As an array's set of elements is not a time series, Pine Script™'s usual mathematical functions are not allowed on them. Special-purpose functions must be used to operate on all of an array's values. The available functions are: `array.abs()`, `array.avg()`, `array.covariance()`, `array.min()`, `array.max()`, `array.median()`, `array.mode()`, `array.percentile_linear_interpolation()`, `array.percentile_nearest_rank()`, `array.percentrank()`, `array.range()`, `array.standardize()`, `array.stdev()`, `array.sum()`, `array.variance()`.

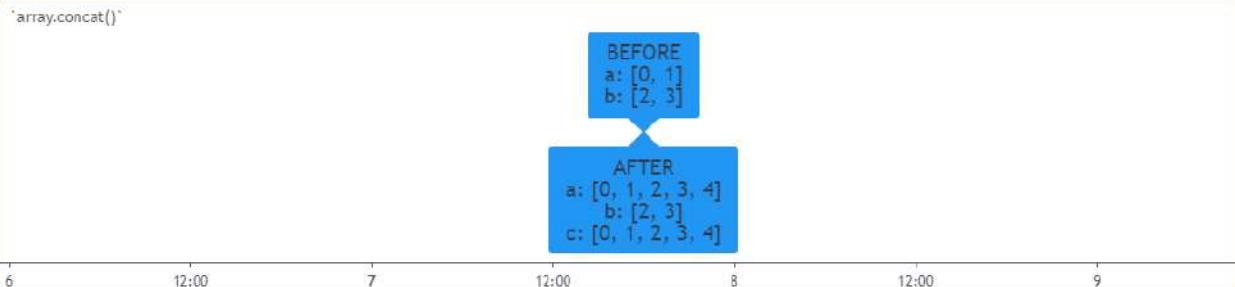
Note that contrary to the usual mathematical functions in Pine Script™, those used on arrays do not return `na` when some of the values they calculate on have `na` values. There are a few exceptions to this rule:

- When all array elements have `na` value or the array contains no elements, `na` is returned. `array.standardize()` however, will return an empty array.
- `array.mode()` will return `na` when no mode is found.

### 3.14.9 Manipulating arrays

#### Concatenation

Two arrays can be merged—or concatenated—using `array.concat()`. When arrays are concatenated, the second array is appended to the end of the first, so the first array is modified while the second one remains intact. The function returns the array ID of the first array:



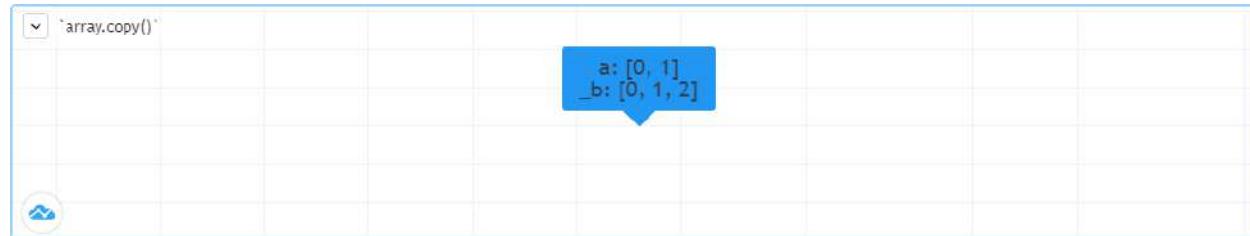
```

1 // @version=5
2 indicator(`array.concat()`)
3 a = array.new<float>(0)
4 b = array.new<float>(0)
5 array.push(a, 0)
6 array.push(a, 1)
7 array.push(b, 2)
8 array.push(b, 3)
9 if barstate.islast
10    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nb: " + str.
11      tostring(b), size = size.large)
12    c = array.concat(a, b)
13    array.push(c, 4)
14    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nb: " + str.
15      tostring(b) + "\nc: " + str.tostring(c), style = label.style_label_up, size = size.
16      large)

```

### Copying

You can copy an array using `array.copy()`. Here we copy the array `a` to a new array named `_b`:



```
1 // @version=5
2 indicator(`array.copy()`)
3 a = array.new<float>(0)
4 array.push(a, 0)
5 array.push(a, 1)
6 if barstate.islast
7     b = array.copy(a)
8     array.push(b, 2)
9     label.new(bar_index, 0, "a: " + str.tostring(a) + "\nb: " + str.tostring(b), size_
10    ↪= size.large)
```

Note that simply using `_b = a` in the previous example would not have copied the array, but only its ID. From thereon, both variables would point to the same array, so using either one would affect the same array.

### Joining

Use `array.join()` to concatenate all of the elements in the array into a string and separate these elements with the specified separator:

```
1 // @version=5
2 indicator("")
3 v1 = array.new<string>(10, "test")
4 v2 = array.new<string>(10, "test")
5 array.push(v2, "test1")
6 v3 = array.new_float(5, 5)
7 v4 = array.new_int(5, 5)
8 l1 = label.new(bar_index, close, array.join(v1))
9 l2 = label.new(bar_index, close, array.join(v2, ","))
10 l3 = label.new(bar_index, close, array.join(v3, ","))
11 l4 = label.new(bar_index, close, array.join(v4, ","))
```

### Sorting

Arrays containing “int” or “float” elements can be sorted in either ascending or descending order using `array.sort()`. The `order` parameter is optional and defaults to `order.ascending`. As all `array.*()` function arguments, it is qualified as “series”, so can be determined at runtime, as is done here. Note that in the example, which array is sorted is also determined at runtime:



```

1 //version=5
2 indicator(`array.sort()`)
3 a = array.new<float>(0)
4 b = array.new<float>(0)
5 array.push(a, 2)
6 array.push(a, 0)
7 array.push(a, 1)
8 array.push(b, 4)
9 array.push(b, 3)
10 array.push(b, 5)
11 if barstate.islast
12     barUp = close > open
13     array.sort(barUp ? a : b, barUp ? order.ascending : order.descending)
14     label.new(bar_index, 0,
15         "a " + (barUp ? "is sorted ▲: " : "is not sorted: ") + str.tostring(a) + "\n\n"
16         "b " + (barUp ? "is not sorted: " : "is sorted ▼: ") + str.tostring(b), size_
        ← = size.large)

```

Another useful option for sorting arrays is to use the `array.sort_indices()` function, which takes a reference to the original array and returns an array containing the indices from the original array. Please note that this function won't modify the original array. The `order` parameter is optional and defaults to `order.ascending`.

## Reversing

Use `array.reverse()` to reverse an array:

```

1 //version=5
2 indicator(`array.reverse()`)
3 a = array.new<float>(0)
4 array.push(a, 0)
5 array.push(a, 1)
6 array.push(a, 2)

```

(continues on next page)

(continued from previous page)

```

7 if barstate.islast
8     array.reverse(a)
9     label.new(bar_index, 0, "a: " + str.tostring(a))

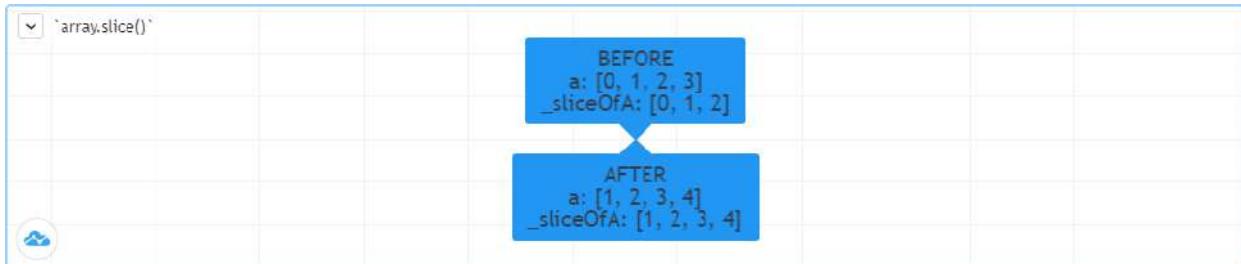
```

## Slicing

Slicing an array using `array.slice()` creates a shallow copy of a subset of the parent array. You determine the size of the subset to slice using the `index_from` and `index_to` parameters. The `index_to` argument must be one greater than the end of the subset you want to slice.

The shallow copy created by the slice acts like a window on the parent array's content. The indices used for the slice define the window's position and size over the parent array. If, as in the example below, a slice is created from the first three elements of an array (indices 0 to 2), then regardless of changes made to the parent array, and as long as it contains at least three elements, the shallow copy will always contain the parent array's first three elements.

Additionally, once the shallow copy is created, operations on the copy are mirrored on the parent array. Adding an element to the end of the shallow copy, as is done in the following example, will widen the window by one element and also insert that element in the parent array at index 3. In this example, to slice the subset from index 0 to index 2 of array `a`, we must use `_sliceOfA = array.slice(a, 0, 3)`:



```

1 //@version=5
2 indicator(`array.slice()`)
3 a = array.new<float>(0)
4 array.push(a, 0)
5 array.push(a, 1)
6 array.push(a, 2)
7 array.push(a, 3)
8 if barstate.islast
9     // Create a shadow of elements at index 1 and 2 from array `a`.
10    sliceOfA = array.slice(a, 0, 3)
11    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nsliceOfA: " + str.
12        tostring(sliceOfA))
13    // Remove first element of parent array `a`.
14    array.remove(a, 0)
15    // Add a new element at the end of the shallow copy, thus also affecting the
16    // original array `a`.
17    array.push(sliceOfA, 4)
18    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nsliceOfA: " + str.
19        tostring(sliceOfA), style = label.style_label_up)

```

### 3.14.10 Searching arrays

We can test if a value is part of an array with the `array.includes()` function, which returns true if the element is found. We can find the first occurrence of a value in an array by using the `array.indexof()` function. The first occurrence is the one with the lowest index. We can also find the last occurrence of a value with `array.lastindexof()`:

```

1 // @version=5
2 indicator("Searching in arrays")
3 valueInput = input.int(1)
4 a = array.new<float>(0)
5 array.push(a, 0)
6 array.push(a, 1)
7 array.push(a, 2)
8 array.push(a, 1)
9 if barstate.islast
10     valueFound      = array.includes(a, valueInput)
11     firstIndexFound = array.indexof(a, valueInput)
12     lastIndexFound = array.lastindexof(a, valueInput)
13     label.new(bar_index, 0, "a: " + str.tostring(a) +
14         "\nFirst " + str.tostring(valueInput) + (firstIndexFound != -1 ? " value was" +
15         "found at index: " + str.tostring(firstIndexFound) : " value was not found.") +
16         "\nLast " + str.tostring(valueInput) + (lastIndexFound != -1 ? " value was" +
17         "found at index: " + str.tostring(lastIndexFound) : " value was not found."))

```

We can also perform a binary search on an array but note that performing a binary search on an array means that the array will first need to be sorted in ascending order only. The `array.binary_search()` function will return the value's index if it was found or -1 if it wasn't. If we want to always return an existing index from the array even if our chosen value wasn't found, then we can use one of the other binary search functions available. The `array.binary_search_leftmost()` function, which returns an index if the value was found or the first index to the left where the value would be found. The `array.binary_search_rightmost()` function is almost identical and returns an index if the value was found or the first index to the right where the value would be found.

### 3.14.11 Error handling

Malformed `array.*()` call syntax in Pine scripts will cause the usual **compiler** error messages to appear in Pine Editor's console, at the bottom of the window, when you save a script. Refer to the Pine Script™ v5 Reference Manual when in doubt regarding the exact syntax of function calls.

Scripts using arrays can also throw **runtime** errors, which appear as an exclamation mark next to the indicator's name on the chart. We discuss those runtime errors in this section.

#### Index xx is out of bounds. Array size is yy

This will most probably be the most frequent error you encounter. It will happen when you reference an nonexistent array index. The “xx” value will be the value of the faulty index you tried to use, and “yy” will be the size of the array. Recall that array indices start at zero—not one—and end at the array’s size, minus one. An array of size 3’s last valid index is thus 2.

To avoid this error, you must make provisions in your code logic to prevent using an index lying outside of the array’s index boundaries. This code will generate the error because the last index we use in the loop is outside the valid index range for the array:

```

1 // @version=5
2 indicator("Out of bounds index")

```

(continues on next page)

(continued from previous page)

```

3 a = array.new<float>(3)
4 for i = 1 to 3
5     array.set(a, i, i)
6 plot(array.pop(a))

```

The correct `for` statement is:

```
for i = 0 to 2
```

To loop on all array elements in an array of unknown size, use:

```

1 //@version=5
2 indicator("Protected `for` loop")
3 sizeInput = input.int(0, "Array size", minval = 0, maxval = 100000)
4 a = array.new<float>(sizeInput)
5 for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
6     array.set(a, i, i)
7 plot(array.pop(a))

```

When you size arrays dynamically using a field in your script's *Settings/Inputs* tab, protect the boundaries of that value using `input.int()`'s `minval` and `maxval` parameters:

```

1 //@version=5
2 indicator("Protected array size")
3 sizeInput = input.int(10, "Array size", minval = 1, maxval = 100000)
4 a = array.new<float>(sizeInput)
5 for i = 0 to sizeInput - 1
6     array.set(a, i, i)
7 plot(array.size(a))

```

See the [Looping](#) section of this page for more information.

### Cannot call array methods when ID of array is 'na'

When an array ID is initialized to `na`, operations on it are not allowed, since no array exists. All that exists at that point is an array variable containing the `na` value rather than a valid array ID pointing to an existing array. Note that an array created with no elements in it, as you do when you use `a = array.new_int(0)`, has a valid ID nonetheless. This code will throw the error we are discussing:

```

1 //@version=5
2 indicator("Out of bounds index")
3 array<int> a = na
4 array.push(a, 111)
5 label.new(bar_index, 0, "a: " + str.tostring(a))

```

To avoid it, create an array with size zero using:

```
array<int> a = array.new_int(0)
```

or:

```
a = array.new_int(0)
```

## Array is too large. Maximum size is 100000

This error will appear if your code attempts to declare an array with a size greater than 100,000. It will also occur if, while dynamically appending elements to an array, a new element would increase the array's size past the maximum.

## Cannot create an array with a negative size

We haven't found any use for arrays of negative size yet, but if you ever do, we may allow them :)

## Cannot use shift() if array is empty.

This error will occur if `array.shift()` is called to remove the first element of an empty array.

## Cannot use pop() if array is empty.

This error will occur if `array.pop()` is called to remove the last element of an empty array.

## Index 'from' should be less than index 'to'

When two indices are used in functions such as `array.slice()`, the first index must always be smaller than the second one.

## Slice is out of bounds of the parent array

This message occurs whenever the parent array's size is modified in such a way that it makes the shallow copy created by a slice point outside the boundaries of the parent array. This code will reproduce it because after creating a slice from index 3 to 4 (the last two elements of our five-element parent array), we remove the parent's first element, making its size four and its last index 3. From that moment on, the shallow copy which is still pointing to the "window" at the parent array's indices 3 to 4, is pointing out of the parent array's boundaries:

```

1 // @version=5
2 indicator("Slice out of bounds")
3 a = array.new<float>(5, 0)
4 b = array.slice(a, 3, 5)
5 array.remove(a, 0)
6 c = array.indexof(b, 2)
7 plot(c)

```



## 3.15 Matrices

- *Introduction*
- *Declaring a matrix*
- *Reading and writing matrix elements*
- *Rows and columns*
- *Looping through a matrix*
- *Copying a matrix*
- *Scope and history*
- *Inspecting a matrix*
- *Manipulating a matrix*
- *Matrix calculations*
- *Error handling*

---

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

---

### 3.15.1 Introduction

Pine Script™ Matrices are collections that store value references in a rectangular format. They are essentially the equivalent of two-dimensional `array` objects with functions and methods for inspection, modification, and specialized calculations. As with `arrays`, all matrix elements must be of the same `type`, which can be a `built-in` or a `user-defined type`.

Matrices reference their elements using two indices: one index for their rows and the other for their columns. Each index starts at 0 and extends to the number of rows/columns in the matrix minus one. Matrices in Pine can have dynamic numbers of rows and columns that vary across bars. The `total number of elements` within a matrix is the product of the number of `rows` and `columns` (e.g., a 5x5 matrix has a total of 25). Like `arrays`, the total number of elements in a matrix cannot exceed 100,000.

### 3.15.2 Declaring a matrix

Pine Script™ uses the following syntax for matrix declaration:

```
[var/varip] [matrix<type>] <identifier> = <expression>
```

Where `<type>` is a `type template` for the matrix that declares the type of values it will contain, and the `<expression>` returns either a matrix instance of the type or `na`.

When declaring a matrix variable as `na`, users must specify that the identifier will reference matrices of a specific type by including the `matrix` keyword followed by a `type template`.

This line declares a new `myMatrix` variable with a value of `na`. It explicitly declares the variable as `matrix<float>`, which tells the compiler that the variable can only accept `matrix` objects containing `float` values:

```
matrix<float> myMatrix = na
```

When a matrix variable is not assigned to na, the `matrix` keyword and its type template are optional, as the compiler will use the type information from the object the variable references.

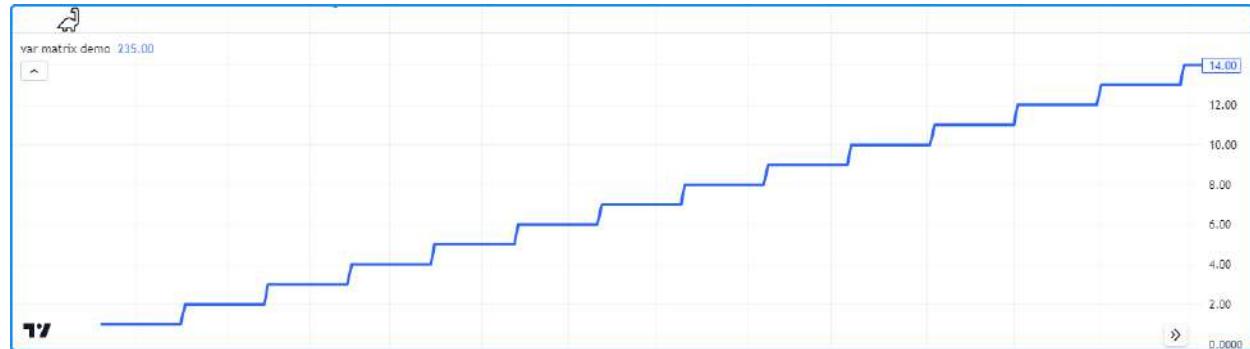
Here, we declare a `myMatrix` variable referencing a new `matrix<float>` instance with two rows, two columns, and an `initial_value` of 0. The variable gets its type information from the new object in this case, so it doesn't require an explicit type declaration:

```
myMatrix = matrix.new<float>(2, 2, 0.0)
```

## Using `var` and `varip` keywords

As with other variables, users can include the `var` or `varip` keywords to instruct a script to declare a matrix variable only once rather than on every bar. A matrix variable declared with this keyword will point to the same instance throughout the span of the chart unless the script explicitly assigns another matrix to it, allowing a matrix and its element references to persist between script iterations.

This script declares an `m` variable assigned to a matrix that holds a single row of two `int` elements using the `var` keyword. On every 20th bar, the script adds 1 to the first element on the first row of the `m` matrix. The `plot()` call displays this element on the chart. As we see from the plot, the value of `m.get(0, 0)` persists between bars, never returning to the initial value of 0:



```

1 // @version=5
2 indicator("var matrix demo")
3
4 // @variable A 1x2 rectangular matrix declared only at `bar_index == 0`, i.e., the_
5 // first bar.
6 var m = matrix.new<int>(1, 2, 0)
7
8 // @variable Is `true` on every 20th bar.
9 bool update = bar_index % 20 == 0
10
11 if update
12     int currentValue = m.get(0, 0) // Get the current value of the first row and_
13 // column.
14     m.set(0, 0, currentValue + 1) // Set the first row and column element value to_
15 // `currentValue + 1`.
16
17 plot(m.get(0, 0), linewidth = 3) // Plot the value from the first row and column.

```

**Note:** Matrix variables declared using `varip` behave as ones using `var` on historical data, but they update their values for realtime bars (i.e., the bars since the script's last compilation) on each new price tick. Matrices assigned to `varip` variables

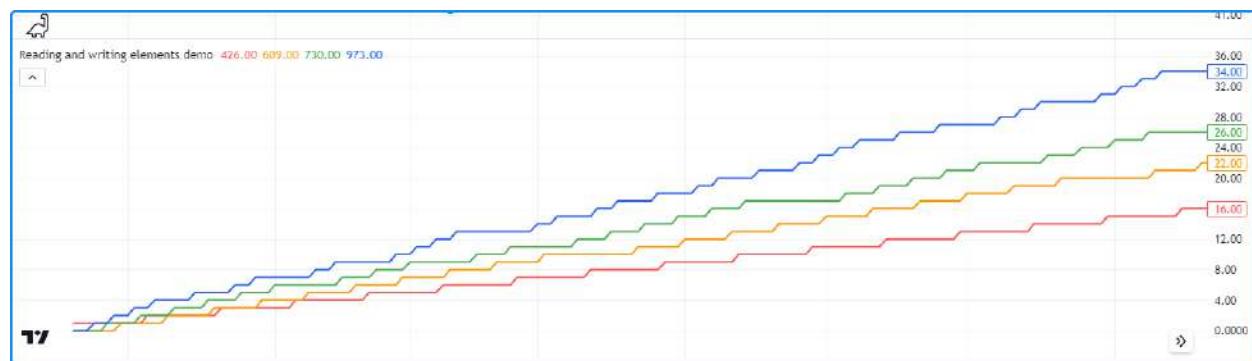
can only hold `int`, `float`, `bool`, `color`, or `string` types or *user-defined types* that exclusively contain within their fields these types or collections (*arrays*, *matrices*, or *maps*) of these types.

### 3.15.3 Reading and writing matrix elements

#### `'matrix.get()'` and `'matrix.set()'`

To retrieve the value from a matrix at a specified `row` and `column` index, use `matrix.get()`. This function locates the specified matrix element and returns its value. Similarly, to overwrite a specific element's value, use `matrix.set()` to assign the element at the specified `row` and `column` to a new value.

The example below defines a square matrix `m` with two rows and columns and an `initial_value` of 0 for all elements on the first bar. The script adds 1 to each element's value on different bars using the `m.get()` and `m.set()` methods. It updates the first row's first value once every 11 bars, the first row's second value once every seven bars, the second row's first value once every five bars, and the second row's second value once every three bars. The script plots each element's value on the chart:



```

1 // @version=5
2 indicator("Reading and writing elements demo")
3
4 // @variable A 2x2 square matrix of `float` values.
5 var m = matrix.new<float>(2, 2, 0.0)
6
7 switch
8     bar_index % 11 == 0 => m.set(0, 0, m.get(0, 0) + 1.0) // Adds 1 to the value at
9         // row 0, column 0 every 11th bar.
10    bar_index % 7 == 0 => m.set(0, 1, m.get(0, 1) + 1.0) // Adds 1 to the value at
11        // row 0, column 1 every 7th bar.
12    bar_index % 5 == 0 => m.set(1, 0, m.get(1, 0) + 1.0) // Adds 1 to the value at
13        // row 1, column 0 every 5th bar.
14    bar_index % 3 == 0 => m.set(1, 1, m.get(1, 1) + 1.0) // Adds 1 to the value at
15        // row 1, column 1 every 3rd bar.
16
17 plot(m.get(0, 0), "Row 0, Column 0 Value", color.red, 2)
18 plot(m.get(0, 1), "Row 0, Column 1 Value", color.orange, 2)
19 plot(m.get(1, 0), "Row 1, Column 0 Value", color.green, 2)
20 plot(m.get(1, 1), "Row 1, Column 1 Value", color.blue, 2)

```

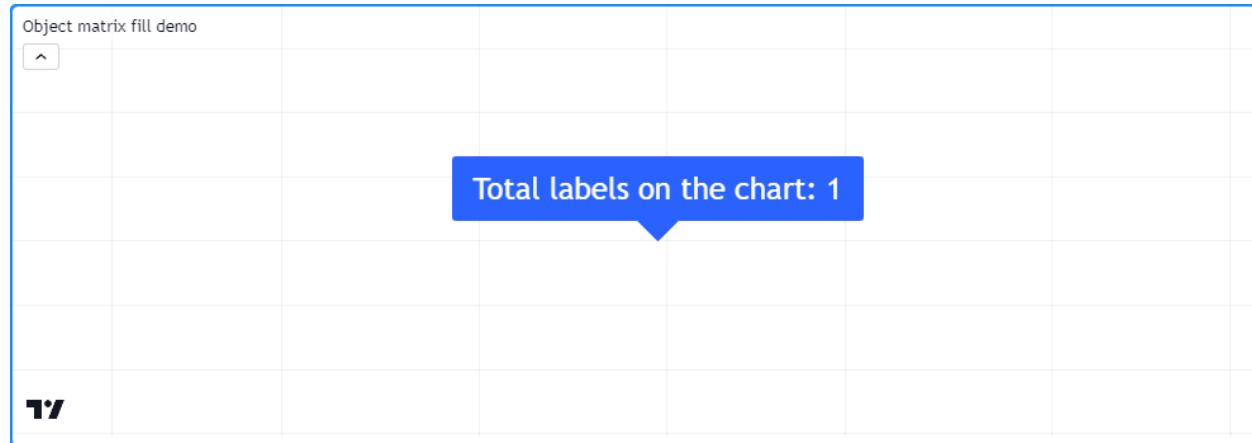
### `matrix.fill()`

To overwrite all matrix elements with a specific value, use `matrix.fill()`. This function points all items in the entire matrix or within the `from_row/column` and `to_row/column` index range to the `value` specified in the call. For example, this snippet declares a 4x4 square matrix, then fills its elements with a `random` value:

```
myMatrix = matrix.new<float>(4, 4)
myMatrix.fill(math.random())
```

Note when using `matrix.fill()` with matrices containing special types (`line`, `linefill`, `box`, `polyline`, `label`, `table`, or `chart.point`) or `UDTs`, all replaced elements will point to the same object passed in the function call.

This script declares a matrix with four rows and columns of `label` references, which it fills with a new `label` object on the first bar. On each bar, the script sets the `x` attribute of the label referenced at row 0, column 0 to `bar_index`, and the `text` attribute of the one referenced at row 3, column 3 to the number of labels on the chart. Although the matrix can reference 16 (4x4) labels, each element points to the *same* instance, resulting in only one label on the chart that updates its `x` and `text` attributes on each bar:



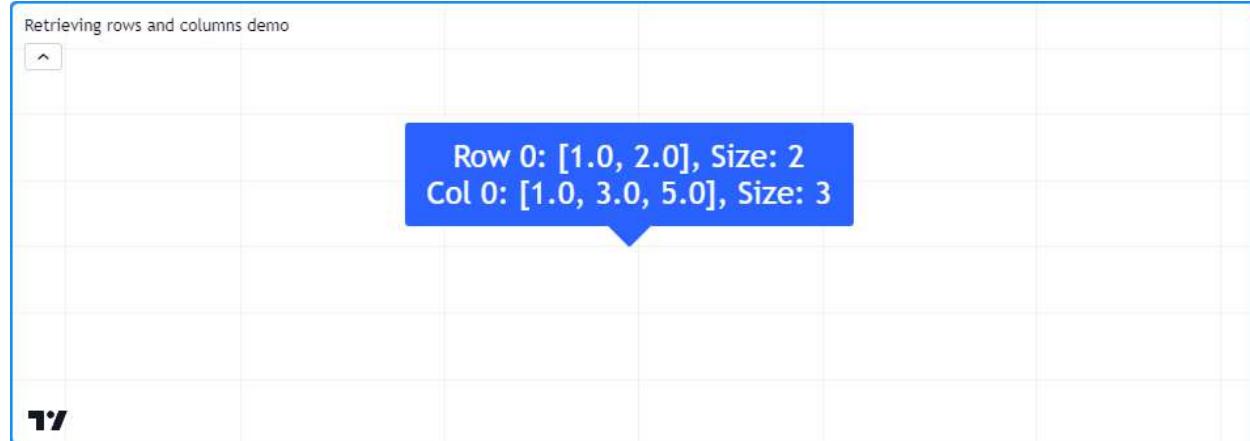
```
1 // @version=5
2 indicator("Object matrix fill demo")
3
4 // @variable A 4x4 label matrix.
5 var matrix<label> m = matrix.new<label>(4, 4)
6
7 // Fill `m` with a new label object on the first bar.
8 if bar_index == 0
9     m.fill(label.new(0, 0, textcolor = color.white, size = size.huge))
10
11 // @variable The number of label objects on the chart.
12 int numLabels = label.all.size()
13
14 // Set the `x` of the label from the first row and column to `bar_index`.
15 m.get(0, 0).set_x(bar_index)
16 // Set the `text` of the label at the last row and column to the number of labels.
17 m.get(3, 3).set_text(str.format("Total labels on the chart: {0}", numLabels))
```

### 3.15.4 Rows and columns

#### Retrieving

Matrices facilitate the retrieval of all values from a specific row or column via the `matrix.row()` and `matrix.col()` functions. These functions return the values as an `array` object sized according to the other dimension of the matrix, i.e., the size of a `matrix.row()` array equals the `number of columns` and the size of a `matrix.col()` array equals the `number of rows`.

The script below populates a  $3 \times 2$  m matrix with the values 1 - 6 on the first chart bar. It calls the `m.row()` and `m.col()` methods to access the first row and column arrays from the matrix and displays them on the chart in a label along with the array sizes:



```

1 // @version=5
2 indicator("Retrieving rows and columns demo")
3
4 //@variable A 3x2 rectangular matrix.
5 var matrix<float> m = matrix.new<float>(3, 2)
6
7 if bar_index == 0
8     m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
9     m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
10    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.
11    m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
12    m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.
13    m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.
14
15 //@variable The first row of the matrix.
16 array<float> row0 = m.row(0)
17 //@variable The first column of the matrix.
18 array<float> column0 = m.col(0)
19
20 //@variable Displays the first row and column of the matrix and their sizes in a
21 // label.
22 var label debugLabel = label.new(0, 0, color = color.blue, textcolor = color.white,
23 // size = size.huge)
24 debugLabel.set_x(bar_index)
25 debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0,
26 // m.columns(), column0, m.rows())))

```

#### Note that:

- To get the sizes of the arrays displayed in the label, we used the `rows()` and `columns()` methods rather than

`array.size()` to demonstrate that the size of the `row0` array equals the number of columns and the size of the `column0` array equals the number of rows.

`matrix.row()` and `matrix.col()` copy the references in a row/column to a new `array`. Modifications to the `arrays` returned by these functions do not directly affect the elements or the shape of a matrix.

Here, we've modified the previous script to set the first element of `row0` to 10 via the `array.set()` method before displaying the label. This script also plots the value from row 0, column 0. As we see, the label shows that the first element of the `row0` array is 10. However, the `plot` shows that the corresponding matrix element still has a value of 1:



```

1 // @version=5
2 indicator("Retrieving rows and columns demo")
3
4 //@variable A 3x2 rectangular matrix.
5 var matrix<float> m = matrix.new<float>(3, 2)
6
7 if bar_index == 0
8     m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
9     m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
10    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.
11    m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
12    m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.
13    m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.
14
15 //@variable The first row of the matrix.
16 array<float> row0 = m.row(0)
17 //@variable The first column of the matrix.
18 array<float> column0 = m.col(0)
19
20 // Set the first `row` element to 10.
21 row0.set(0, 10)
22
23 //@variable Displays the first row and column of the matrix and their sizes in a
24 // label.
25 var label debugLabel = label.new(0, m.get(0, 0), color = color.blue, textcolor =
26 //color.white, size = size.huge)
27 debugLabel.set_x(bar_index)
28 debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0,
29 //m.columns(), column0, m.rows()))
30
31 // Plot the first element of `m`.
32 plot(m.get(0, 0), linewidth = 3)

```

Although changes to an `array` returned by `matrix.row()` or `matrix.col()` do not directly affect a parent matrix, it's important to note the resulting array from a matrix containing *UDTs* or special types, including `line`, `linefill`, `box`, `polyline`, `label`, `table`, or `chart.point`, behaves as a *shallow copy* of a row/column, i.e., the elements within an array returned from these

functions point to the same objects as the corresponding matrix elements.

This script contains a custom `myUDT` type containing a `value` field with an initial value of 0. It declares a  $1 \times 1$  `m` matrix to hold a single `myUDT` instance on the first bar, then calls `m.row(0)` to copy the first row of the matrix as an `array`. On every chart bar, the script adds 1 to the `value` field of the first `row` array element. In this case, the `value` field of the matrix element increases on every bar as well since both elements reference the same object:

```

1 // @version=5
2 indicator("Row with reference types demo")
3
4 // @type A custom type that holds a float value.
5 type myUDT
6     float value = 0.0
7
8 // @variable A 1x1 matrix of `myUDT` type.
9 var matrix<myUDT> m = matrix.new<myUDT>(1, 1, myUDT.new())
10 // @variable A shallow copy of the first row of `m`.
11 array<myUDT> row = m.row(0)
12 // @variable The first element of the `row`.
13 myUDT firstElement = row.get(0)
14
15 firstElement.value += 1.0 // Add 1 to the `value` field of `firstElement`. Also
16 // affects the element in the matrix.
17 plot(m.get(0, 0).value, linewidth = 3) // Plot the `value` of the `myUDT` object from
18 // the first row and column of `m`.

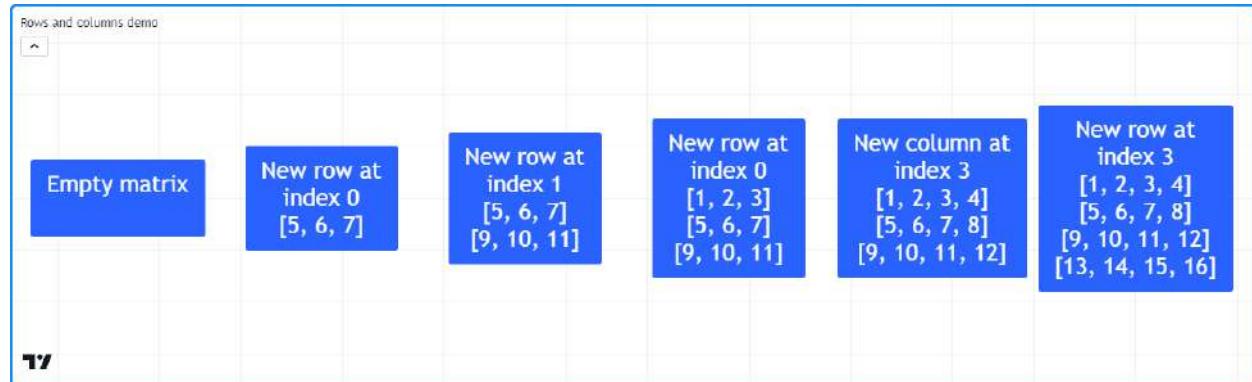
```

## Inserting

Scripts can add new rows and columns to a matrix via `matrix.add_row()` and `matrix.add_col()`. These functions insert the value references from an `array` into a matrix at the specified `row/column` index. If the `matrix` is empty (has no rows or columns), the `array_id` in the call can be of any size. If a row/column exists at the specified index, the matrix increases the index value for the existing row/column and all after it by 1.

The script below declares an empty `m` matrix and inserts rows and columns using the `m.add_row()` and `m.add_col()` methods. It first inserts an array with three elements at row 0, turning `m` into a  $1 \times 3$  matrix, then another at row 1, changing the shape to  $2 \times 3$ . After that, the script inserts another array at row 0, which changes the shape of `m` to  $3 \times 3$  and shifts the index of all rows previously at index 0 and higher. It inserts another array at the last column index, changing the shape to  $3 \times 4$ . Finally, it adds an array with four values at the end row index.

The resulting matrix has four rows and columns and contains values 1-16 in ascending order. The script displays the rows of `m` after each row/column insertion with a user-defined `debugLabel()` function to visualize the process:



```

1 // @version=5
2 indicator("Rows and columns demo")
3
4 // @function Displays the rows of a matrix in a label with a note.
5 // @param    this The matrix to display.
6 // @param    barIndex The `bar_index` to display the label at.
7 // @param    bgColor The background color of the label.
8 // @param    textColor The color of the label's text.
9 // @param    note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textcolor = textColor, size = size.huge
20         )
21
22 // Create an empty matrix.
23 var m = matrix.new<float>()
24
25 if bar_index == last_bar_index - 1
26     debugLabel(m, bar_index - 30, note = "Empty matrix")
27
28     // Insert an array at row 0. `m` will now have 1 row and 3 columns.
29     m.add_row(0, array.from(5, 6, 7))
30     debugLabel(m, bar_index - 20, note = "New row at\nindex 0")
31
32     // Insert an array at row 1. `m` will now have 2 rows and 3 columns.
33     m.add_row(1, array.from(9, 10, 11))
34     debugLabel(m, bar_index - 10, note = "New row at\nindex 1")
35
36     // Insert another array at row 0. `m` will now have 3 rows and 3 columns.
37     // The values previously on row 0 will now be on row 1, and the values from row 1
38     ↪will be on row 2.
39     m.add_row(0, array.from(1, 2, 3))
40     debugLabel(m, bar_index, note = "New row at\nindex 0")
41
42     // Insert an array at column 3. `m` will now have 3 rows and 4 columns.
43     m.add_col(3, array.from(4, 8, 12))
44     debugLabel(m, bar_index + 10, note = "New column at\nindex 3")
45
46     // Insert an array at row 3. `m` will now have 4 rows and 4 columns.
47     m.add_row(3, array.from(13, 14, 15, 16))
48     debugLabel(m, bar_index + 20, note = "New row at\nindex 3")

```

**Note:** Just as the row or column arrays *retrieved* from a matrix of `line`, `linefill`, `box`, `polyline`, `label`, `table`, `chart.point`, or `UDT` instances behave as shallow copies, the elements of matrices containing such types reference the same objects as the *arrays* inserted into them. Modifications to the element values in either object affect the other in such cases.

## Removing

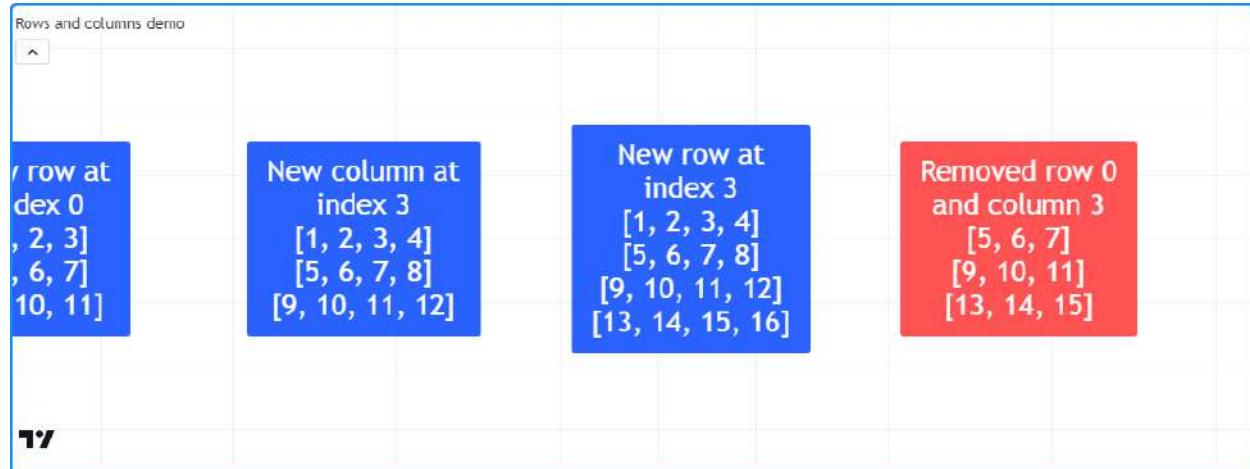
To remove a specific row or column from a matrix, use `matrix.remove_row()` and `matrix.remove_col()`. These functions remove the specified row/column and decrease the index values of all rows/columns after it by 1.

For this example, we've added these lines of code to our “Rows and columns demo” script from the [section above](#):

```
// Removing example

    // Remove the first row and last column from the matrix. `m` will now have 3 rows
    // and 3 columns.
    m.remove_row(0)
    m.remove_col(3)
    debugLabel(m, bar_index + 30, color.red, note = "Removed row 0\nand column 3")
```

This code removes the first row and the last column of the `m` matrix using the `m.remove_row()` and `m.remove_col()` methods and displays the rows in a label at `bar_index + 30`. As we can see, `m` has a `3x3` shape after executing this block, and the index values for all existing rows are reduced by 1:



## Swapping

To swap the rows and columns of a matrix without altering its dimensions, use `matrix.swap_rows()` and `matrix.swap_columns()`. These functions swap the locations of the elements at the `row1/column1` and `row2/column2` indices.

Let's add the following lines to the [previous example](#), which swap the first and last rows of `m` and display the changes in a label at `bar_index + 40`:

```
// Swapping example

    // Swap the first and last row. `m` retains the same dimensions.
    m.swap_rows(0, 2)
    debugLabel(m, bar_index + 40, color.purple, note = "Swapped rows 0\nand 2")
```

In the new label, we see the matrix has the same number of rows as before, and the first and last rows have traded places:



## Replacing

It may be desirable in some cases to completely *replace* a row or column in a matrix. To do so, *insert* the new array at the desired row/column and *remove* the old elements previously at that index.

In the following code, we've defined a `replaceRow()` method that uses the `add_row()` method to insert the new values at the `row` index and uses the `remove_row()` method to remove the old row that moved to the `row + 1` index. This script uses the `replaceRow()` method to fill the rows of a 3x3 matrix with the numbers 1-9. It draws a label on the chart before and after replacing the rows using the custom `debugLabel()` method:



```

1 // @version=5
2 indicator("Replacing rows demo")
3
4 //@function Displays the rows of a matrix in a label with a note.
5 //@param    this The matrix to display.
6 //@param    barIndex The `bar_index` to display the label at.
7 //@param    bgColor The background color of the label.
8 //@param    textColor The color of the label's text.
9 //@param    note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,

```

(continues on next page)

(continued from previous page)

```

12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ←center,
19             textcolor = textColor, size = size.huge
20         )
21
22 // @function Replaces the `row` of `this` matrix with a new array of `values`.
23 // @param    row The row index to replace.
24 // @param    values The array of values to insert.
25 method replaceRow(matrix<float> this, int row, array<float> values) =>
26     this.add_row(row, values) // Inserts a copy of the `values` array at the `row`.
27     this.remove_row(row + 1) // Removes the old elements previously at the `row`.
28
29 // @variable A 3x3 matrix.
30 var matrix<float> m = matrix.new<float>(3, 3, 0.0)
31
32 if bar_index == last_bar_index - 1
33     m.debugLabel(note = "Original")
34     // Replace each row of `m`.
35     m.replaceRow(0, array.from(1.0, 2.0, 3.0))
36     m.replaceRow(1, array.from(4.0, 5.0, 6.0))
37     m.replaceRow(2, array.from(7.0, 8.0, 9.0))
38     m.debugLabel(bar_index + 10, note = "Replaced rows")

```

### 3.15.5 Looping through a matrix

#### 'for'

When a script only needs to iterate over the row/column indices in a matrix, the most common method is to use `for` loops. For example, this line creates a loop with a `row` value that starts at 0 and increases by one until it reaches one less than the number of rows in the `m` matrix (i.e., the last row index):

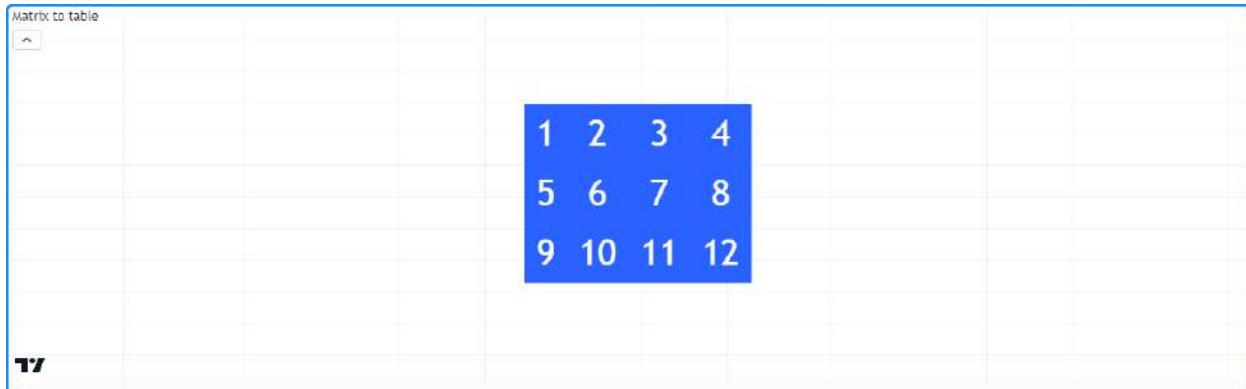
```
for row = 0 to m.rows() - 1
```

To iterate over all index values in the `m` matrix, we can create a *nested* loop that iterates over each `column` index on each `row` value:

```
for row = 0 to m.rows() - 1
    for column = 0 to m.columns() - 1
```

Let's use this nested structure to create a `method` that visualizes matrix elements. In the script below, we've defined a `toTable()` method that displays the elements of a matrix within a `table` object. It iterates over each `row` index and over each `column` index on every `row`. Within the loop, it converts each element to a `string` to display in the corresponding table cell.

On the first bar, the script creates an empty `m` matrix, populates it with rows, and calls `m.toTable()` to display its elements:



```

1 // @version=5
2 indicator("for loop demo", "Matrix to table")
3
4 // @function Displays the elements of `this` matrix in a table.
5 // @param this The matrix to display.
6 // @param position The position of the table on the chart.
7 // @param bgColor The background color of the table.
8 // @param textColor The color of the text in each cell.
9 // @param note A note string to display on the bottom row of the table.
10 // @returns A new `table` object with cells corresponding to each element of `this` ↵
11 //      ↵matrix.
12 method toTable(
13     matrix<float> this, string position = position.middle_center,
14     color bgColor = color.blue, color textColor = color.white,
15     string note = na
16 ) =>
17     // @variable The number of rows in `this` matrix.
18     int rows = this.rows()
19     // @variable The number of columns in `this` matrix.
20     int columns = this.columns()
21     // @variable A table that displays the elements of `this` matrix with an optional ↵
22     //      ↵`note` cell.
23     table result = table.new(position, columns, rows + 1, bgColor)
24
25     // Iterate over each row index of `this` matrix.
26     for row = 0 to rows - 1
27         // Iterate over each column index of `this` matrix on each `row` .
28         for col = 0 to columns - 1
29             // @variable The element from `this` matrix at the `row` and `col` index.
30             float element = this.get(row, col)
31             // Initialize the corresponding `result` cell with the `element` value.
32             result.cell(col, row, str.tostring(element), textColor, textSize_ ↵
33             //      ↵size = size.huge)
34
35             // Initialize a merged cell on the bottom row if a `note` is provided.
36             if not na(note)
37                 result.cell(0, rows, note, textColor, textSize = size.huge)
38                 result.merge_cells(0, rows, columns - 1, rows)
39
40             result // Return the `result` table.
41
42 // @variable A 3x4 matrix of values.
43 var m = matrix.new<float>()

```

(continues on next page)

(continued from previous page)

```

42 if bar_index == 0
43     // Add rows to `m`.
44     m.add_row(0, array.from(1, 2, 3))
45     m.add_row(1, array.from(5, 6, 7))
46     m.add_row(2, array.from(9, 10, 11))
47     // Add a column to `m`.
48     m.add_col(3, array.from(4, 8, 12))
49     // Display the elements of `m` in a table.
50     m.toTable()

```

**'for...in'**

When a script needs to iterate over and retrieve the rows of a matrix, using the `for...in` structure is often preferred over the standard `for` loop. This structure directly references the row `arrays` in a matrix, making it a more convenient option for such use cases. For example, this line creates a loop that returns a `row` array for each row in the `m` matrix:

```
for row in m
```

The following indicator calculates the moving average of OHLC data with an input `length` and displays the values on the chart. The custom `rowWiseAvg()` method loops through the rows of a matrix using a `for...in` structure to produce an array containing the `array.avg()` of each `row`.

On the first chart bar, the script creates a new `m` matrix with four rows and `length` columns, which it queues a new column of OHLC data into via the `m.add_col()` and `m.remove_col()` methods on each subsequent bar. It uses `m.rowWiseAvg()` to calculate the array of row-wise averages, then it plots the element values on the chart:



```

1 //@version=5
2 indicator("for...in loop demo", "Average OHLC", overlay = true)
3
4 //@variable The number of terms in the average.
5 int length = input.int(20, "Length", minval = 1)
6

```

(continues on next page)

(continued from previous page)

```

7 // @function Calculates the average of each matrix row.
8 method rowWiseAvg(matrix<float> this) =>
9     // @variable An array with elements corresponding to each row's average.
10    array<float> result = array.new<float>()
11    // Iterate over each `row` of `this` matrix.
12    for row in this
13        // Push the average of each `row` into the `result`.
14        result.push(row.avg())
15    result // Return the resulting array.
16
17 // @variable A 4x`length` matrix of values.
18 var matrix<float> m = matrix.new<float>(4, length)
19
20 // Add a new column containing OHLC values to the matrix.
21 m.add_col(m.columns(), array.from(open, high, low, close))
22 // Remove the first column.
23 m.remove_col(0)
24
25 // @variable An array containing averages of `open`, `high`, `low`, and `close` over
26 // `length` bars.
27 array<float> averages = m.rowWiseAvg()
28
29 plot(averages.get(0), "Average Open", color.blue, 2)
30 plot(averages.get(1), "Average High", color.green, 2)
31 plot(averages.get(2), "Average Low", color.red, 2)
32 plot(averages.get(3), "Average Close", color.orange, 2)

```

**Note that:**

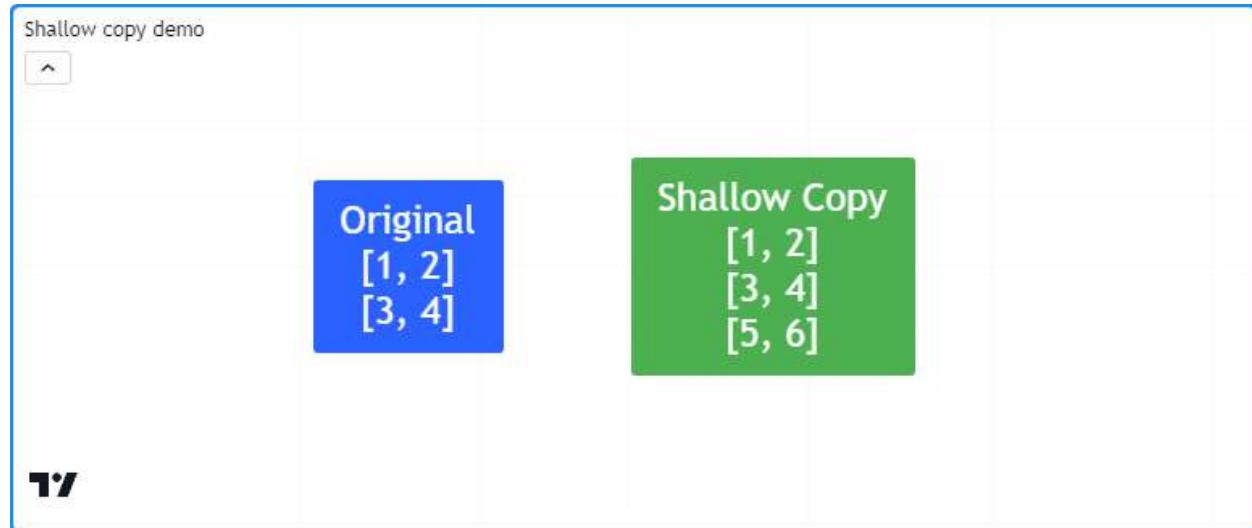
- `for...in` loops can also reference the index value of each row. For example, `for [i, row] in m` creates a tuple containing the `i` row index and the corresponding `row` array from the `m` matrix on each loop iteration.

### 3.15.6 Copying a matrix

#### Shallow copies

Pine scripts can copy matrices via `matrix.copy()`. This function returns a *shallow copy* of a matrix that does not affect the shape of the original matrix or its references.

For example, this script assigns a new matrix to the `myMatrix` variable and adds two columns. It creates a new `myCopy` matrix from `myMatrix` using the `myMatrix.copy()` method, then adds a new row. It displays the rows of both matrices in labels via the user-defined `debugLabel()` function:



```

1 // @version=5
2 indicator("Shallow copy demo")
3
4 //@function Displays the rows of a matrix in a label with a note.
5 //@param    this The matrix to display.
6 //@param    barIndex The `bar_index` to display the label at.
7 //@param    bgColor The background color of the label.
8 //@param    textColor The color of the label's text.
9 //@param    note The text to display above the rows.
10 method debugLabel()
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12         color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textColor = textColor, size = size.huge
20         )
21
22 //@variable A 2x2 `float` matrix.
23 matrix<float> myMatrix = matrix.new<float>()
24 myMatrix.add_col(0, array.from(1.0, 3.0))
25 myMatrix.add_col(1, array.from(2.0, 4.0))
26
27 //@variable A shallow copy of `myMatrix`.
28 matrix<float> myCopy = myMatrix.copy()
29 // Add a row to the last index of `myCopy`.
30 myCopy.add_row(myCopy.rows(), array.from(5.0, 6.0))
31
32 if bar_index == last_bar_index - 1
33     // Display the rows of both matrices in separate labels.
34     myMatrix.debugLabel(note = "Original")
35     myCopy.debugLabel(bar_index + 10, color.green, note = "Shallow Copy")

```

It's important to note that the elements within shallow copies of a matrix point to the same values as the original matrix. When matrices contain special types ([line](#), [linefill](#), [box](#), [polyline](#), [label](#), [table](#), or [chart.point](#)) or [user-defined types](#), the elements of a shallow copy reference the same objects as the original.

This script declares a `myMatrix` variable with a `newLabel` as the initial value. It then copies `myMatrix` to a `myCopy` variable via `myMatrix.copy()` and plots the number of labels. As we see below, there's only one `label` on the chart, as the element in `myCopy` references the same object as the element in `myMatrix`. Consequently, changes to the element values in `myCopy` affect the values in both matrices:



```

1 // @version=5
2 indicator("Shallow copy demo")
3
4 //@variable Initial value of the original matrix elements.
5 var label newLabel = label.new(
6     bar_index, 1, "Original", color = color.blue, textcolor = color.white, size =_
7     size.huge
8 )
9
10 //@variable A 1x1 matrix containing a new `label` instance.
11 var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel)
12 //@variable A shallow copy of `myMatrix`.
13 var matrix<label> myCopy = myMatrix.copy()
14
15 //@variable The first label from the `myCopy` matrix.
16 label testLabel = myCopy.get(0, 0)
17
18 // Change the `text`, `style`, and `x` values of `testLabel`. Also affects the_
19 // `newLabel`.
20 testLabel.set_text("Copy")
21 testLabel.set_style(label.style_label_up)
22 testLabel.set_x(bar_index)
23
24 // Plot the total number of labels.
25 plot(label.all.size(), linewidth = 3)

```

## Deep copies

One can produce a *deep copy* of a matrix (i.e., a matrix whose elements point to copies of the original values) by explicitly copying each object the matrix references.

Here, we've added a `deepCopy()` user-defined method to our previous script. The method creates a new matrix and uses `nested for loops` to assign all elements to copies of the originals. When the script calls this method instead of the built-in `copy()`, we see that there are now two labels on the chart, and any changes to the label from `myCopy` do not affect the one from `myMatrix`:



```

1 //@version=5
2 indicator("Deep copy demo")
3
4 //@function Returns a deep copy of a label matrix.
5 method matrix<label> DeepCopy(matrix<label> this) =>
6     //@variable A deep copy of `this` matrix.
7     matrix<label> that = this.copy()
8     for row = 0 to that.rows() - 1
9         for column = 0 to that.columns() - 1
10            // Assign the element at each `row` and `column` of `that` matrix to a
11            // copy of the retrieved label.
12            that.set(row, column, that.get(row, column).copy())
13
14 //@variable Initial value of the original matrix.
15 var label newLabel = label.new(
16     bar_index, 2, "Original", color = color.blue, textcolor = color.white, size =
17     size.huge
18 )
19 //@variable A 1x1 matrix containing a new `label` instance.
20 var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel)
21 //@variable A deep copy of `myMatrix`.
22 var matrix<label> myCopy = myMatrix.deepCopy()
23
24 //@variable The first label from the `myCopy` matrix.
25 label testLabel = myCopy.get(0, 0)
26
27 // Change the `text`, `style`, and `x` values of `testLabel`. Does not affect the
28 // `newLabel`.

```

(continues on next page)

(continued from previous page)

```

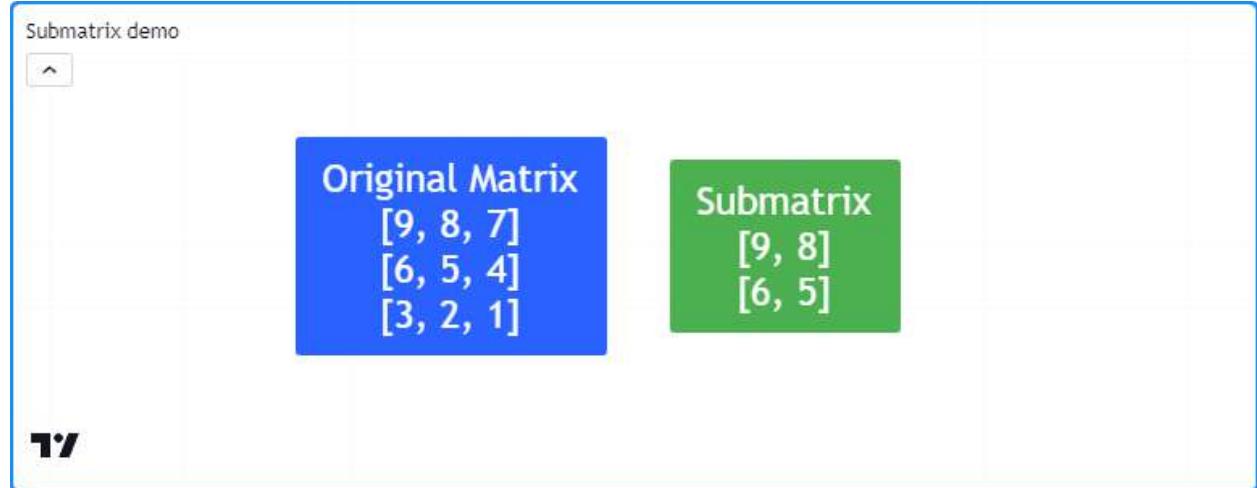
28 testLabel.set_text("Copy")
29 testLabel.set_style(label.style_label_up)
30 testLabel.set_x(bar_index)
31
32 // Change the `x` value of `newLabel`.
33 newLabel.set_x(bar_index)
34
35 // Plot the total number of labels.
36 plot(label.all.size(), linewidth = 3)

```

## Submatrices

In Pine, a *submatrix* is a *shallow copy* of an existing matrix that only includes the rows and columns specified by the `from_row/column` and `to_row/column` parameters. In essence, it is a sliced copy of a matrix.

For example, the script below creates an `mSub` matrix from the `m` matrix via the `m.submatrix()` method, then calls our user-defined `debugLabel()` function to display the rows of both matrices in labels:



```

1 // @version=5
2 indicator("Submatrix demo")
3
4 //@function Displays the rows of a matrix in a label with a note.
5 //@param this The matrix to display.
6 //@param barIndex The `bar_index` to display the label at.
7 //@param bgColor The background color of the label.
8 //@param textColor The color of the label's text.
9 //@param note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textColor = textColor, size = size.huge
20         )

```

(continues on next page)

(continued from previous page)

```

20 // @variable A 3x3 matrix of values.
21 var m = matrix.new<float>()
22
23 if bar_index == last_bar_index - 1
24     // Add columns to `m`.
25     m.add_col(0, array.from(9, 6, 3))
26     m.add_col(1, array.from(8, 5, 2))
27     m.add_col(2, array.from(7, 4, 1))
28     // Display the rows of `m`.
29     m.debugLabel(note = "Original Matrix")
30
31
32     // @variable A 2x2 submatrix of `m` containing the first two rows and columns.
33     matrix<float> mSub = m.submatrix(from_row = 0, to_row = 2, from_column = 0, to_
34     ↪column = 2)
35     // Display the rows of `mSub`
36     debugLabel(mSub, bar_index + 10, bgColor = color.green, note = "Submatrix")

```

### 3.15.7 Scope and history

Matrix variables leave historical trails on each bar, allowing scripts to use the history-referencing operator `[]` to interact with past matrix instances previously assigned to a variable. Additionally, scripts can modify matrices assigned to global variables from within the scopes of *functions*, *methods*, and *conditional structures*.

This script calculates the average ratios of body and wick distances relative to the bar range over `length` bars. It displays the data along with values from `length` bars ago in a table. The user-defined `addData()` function adds columns of current and historical ratios to the `globalMatrix`, and the `calcAvg()` function references previous matrices assigned to `globalMatrix` using the `[]` operator to calculate a matrix of averages:



```

1 // @version=5
2 indicator("Scope and history demo", "Bar ratio comparison")
3

```

(continues on next page)

(continued from previous page)

```

4 int length = input.int(10, "Length", 1)
5
6 //@variable A global matrix.
7 matrix<float> globalMatrix = matrix.new<float>()
8
9 //@function Calculates the ratio of body range to candle range.
10 bodyRatio() =>
11     math.abs(close - open) / (high - low)
12
13 //@function Calculates the ratio of upper wick range to candle range.
14 upperWickRatio() =>
15     (high - math.max(open, close)) / (high - low)
16
17 //@function Calculates the ratio of lower wick range to candle range.
18 lowerWickRatio() =>
19     (math.min(open, close) - low) / (high - low)
20
21 //@function Adds data to the `globalMatrix`.
22 addData() =>
23     // Add a new column of data at `column` 0.
24     globalMatrix.add_col(0, array.from(bodyRatio(), upperWickRatio(),  

25     ↵lowerWickRatio()))
26     // @variable The column of `globalMatrix` from index 0 `length` bars ago.
27     array<float> pastValues = globalMatrix.col(0)[length]
28     // Add `pastValues` to the `globalMatrix`, or an array of `na` if `pastValues` is  

29     ↵`na`.
30     if na(pastValues)
31         globalMatrix.add_col(1, array.new<float>(3))
32     else
33         globalMatrix.add_col(1, pastValues)
34
35 //@function Returns the `length`-bar average of matrices assigned to `globalMatrix`  

36     ↵on historical bars.
37 calcAvg() =>
38     // @variable The sum historical `globalMatrix` matrices.
39     matrix<float> sums = matrix.new<float>(globalMatrix.rows(), globalMatrix.  

40     ↵columns(), 0.0)
41     for i = 0 to length - 1
42         // @variable The `globalMatrix` matrix `i` bars before the current bar.
43         matrix<float> previous = globalMatrix[i]
44         // Break the loop if `previous` is `na`.
45         if na(previous)
46             sums.fill(na)
47             break
48         // Assign the sum of `sums` and `previous` to `sums`.
49         sums := matrix.sum(sums, previous)
50         // Divide the `sums` matrix by the `length`.
51         result = sums.mult(1.0 / length)
52
53 // Add data to the `globalMatrix`.
54 addData()
55
56 //@variable The historical average of the `globalMatrix` matrices.
57 globalAvg = calcAvg()
58
59 //@variable A `table` displaying information from the `globalMatrix`.
60 var table infoTable = table.new(

```

(continues on next page)

(continued from previous page)

```

57     position.middle_center, globalMatrix.columns() + 1, globalMatrix.rows() + 1,_
58     bgcolor = color.navy
59 )
60
60 // Define value cells.
61 for [i, row] in globalAvg
62     for [j, value] in row
63         color textColor = value > 0.333 ? color.orange : color.gray
64         infoTable.cell(j + 1, i + 1, str.tostring(value), text_color = textColor,_
65         text_size = size.huge)
66
66 // Define header cells.
67 infoTable.cell(0, 1, "Body ratio", text_color = color.white, text_size = size.huge)
68 infoTable.cell(0, 2, "Upper wick ratio", text_color = color.white, text_size = size.-
69     huge)
69 infoTable.cell(0, 3, "Lower wick ratio", text_color = color.white, text_size = size.-
70     huge)
70 infoTable.cell(1, 0, "Current average", text_color = color.white, text_size = size.-
71     huge)
71 infoTable.cell(2, 0, str.format("{0} bars ago", length), text_color = color.white,_
    text_size = size.huge)

```

**Note that:**

- The `addData()` and `calcAvg()` functions have no parameters, as they directly interact with the `globalMatrix` and `length` variables declared in the outer scope.
- `calcAvg()` calculates the average by adding previous matrices using `matrix.sum()` and multiplying all elements by `1 / length` using `matrix.mult()`. We discuss these and other specialized functions in our *Matrix calculations* section below.

### 3.15.8 Inspecting a matrix

The ability to inspect the shape of a matrix and patterns within its elements is crucial, as it helps reveal important information about a matrix and its compatibility with various calculations and transformations. Pine Script™ includes several built-ins for matrix inspection, including `matrix.is_square()`, `matrix.is_identity()`, `matrix.is_diagonal()`, `matrix.is_antidiagonal()`, `matrix.is_symmetric()`, `matrix.is_antisymmetric()`, `matrix.is_triangular()`, `matrix.is_stochastic()`, `matrix.is_binary()`, and `matrix.is_zero()`.

To demonstrate these features, this example contains a custom `inspect()` method that uses conditional blocks with `matrix.is_*` functions to return information about a matrix. It displays a string representation of an `m` matrix and the description returned from `m.inspect()` in labels on the chart:

Matrix inspection demo

The matrix  $m$  is displayed as:

```
[1, 0, 0, 0]
[0, 1, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]
```

**This matrix:**

- Has an equal number of rows and columns.
- Contains only 1s and 0s.
- Contains only 0s above and/or below its main diagonal.
- Only has nonzero values in its main diagonal.
- Equals its transpose.
- Equals the negative of its transpose.
- Is the identity matrix.

17

```

1 //@version=5
2 indicator("Matrix inspection demo")
3
4 //@function Inspects a matrix using `matrix.is_*()` functions and returns a `string` describing some of its features.
5 method inspect(matrix<int> this)=>
6     //@variable A string describing `this` matrix.
7     string result = "This matrix:\n"
8     if this.is_square()
9         result += "- Has an equal number of rows and columns.\n"
10    if this.is_binary()
11        result += "- Contains only 1s and 0s.\n"
12    if this.is_zero()
13        result += "- Is filled with 0s.\n"
14    if this.is_triangular()
15        result += "- Contains only 0s above and/or below its main diagonal.\n"
16    if this.is_diagonal()
17        result += "- Only has nonzero values in its main diagonal.\n"
18    if this.is_antidiagonal()
19        result += "- Only has nonzero values in its main antidiagonal.\n"
20    if this.is_symmetric()
21        result += "- Equals its transpose.\n"
22    if this.is_antisymmetric()
23        result += "- Equals the negative of its transpose.\n"
24    if this.is_identity()
25        result += "- Is the identity matrix.\n"
26    result
27
28 //@variable A 4x4 identity matrix.
29 matrix<int> m = matrix.new<int>()
30
31 // Add rows to the matrix.
32 m.add_row(0, array.from(1, 0, 0, 0))
33 m.add_row(1, array.from(0, 1, 0, 0))
34 m.add_row(2, array.from(0, 0, 1, 0))
35 m.add_row(3, array.from(0, 0, 0, 1))
36
37 if bar_index == last_bar_index - 1
38     // Display the `m` matrix in a blue label.
39     label.new(
40         bar_index, 0, str.tostring(m), color = color.blue, style = label.style_label_right,

```

(continues on next page)

(continued from previous page)

```

41     textcolor = color.white, size = size.huge
42   )
43 // Display the result of `m.inspect()` in a purple label.
44 label.new(
45   bar_index, 0, m.inspect(), color = color.purple, style = label.style_label_
46 ←left,
47   textcolor = color.white, size = size.huge
    )

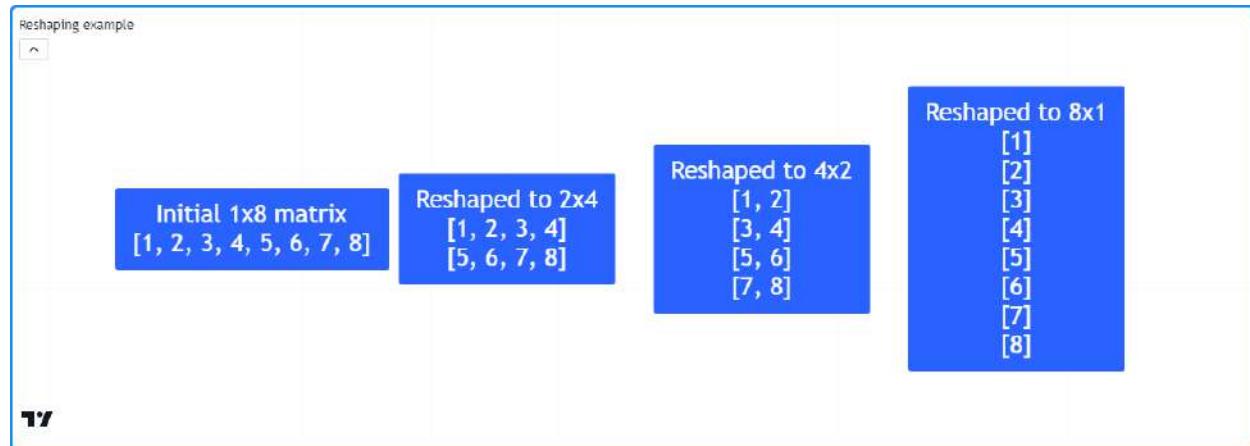
```

### 3.15.9 Manipulating a matrix

#### Reshaping

The shape of a matrix can determine its compatibility with various matrix operations. In some cases, it is necessary to change the dimensions of a matrix without affecting the number of elements or the values they reference, otherwise known as *reshaping*. To reshape a matrix in Pine, use the `matrix.reshape()` function.

This example demonstrates the results of multiple reshaping operations on a matrix. The initial `m` matrix has a  $1 \times 8$  shape (one row and eight columns). Through successive calls to the `m.reshape()` method, the script changes the shape of `m` to  $2 \times 4$ ,  $4 \times 2$ , and  $8 \times 1$ . It displays each reshaped matrix in a label on the chart using the custom `debugLabel()` method:



```

1 //@version=5
2 indicator("Reshaping example")
3
4 //@function Displays the rows of a matrix in a label with a note.
5 //@param    this The matrix to display.
6 //@param    barIndex The `bar_index` to display the label at.
7 //@param    bgColor The background color of the label.
8 //@param    textColor The color of the label's text.
9 //@param    note The text to display above the rows.
10 method debugLabel(
11   matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12   color textColor = color.white, string note = ""
13 ) =>
14   labelText = note + "\n" + str.tostring(this)
15   if barstate.ishistory
16     label.new(
17       barIndex, 0, labelText, color = bgColor, style = label.style_label_

```

(continues on next page)

(continued from previous page)

```

18     ↪center,
19         textcolor = textColor, size = size.huge
20     )
21
22 // @variable A matrix containing the values 1-8.
23 matrix<int> m = matrix.new<int>()
24
25 if bar_index == last_bar_index - 1
26     // Add the initial vector of values.
27     m.add_row(0, array.from(1, 2, 3, 4, 5, 6, 7, 8))
28     m.debugLabel(note = "Initial 1x8 matrix")
29
30     // Reshape. `m` now has 2 rows and 4 columns.
31     m.reshape(2, 4)
32     m.debugLabel(bar_index + 10, note = "Reshaped to 2x4")
33
34     // Reshape. `m` now has 4 rows and 2 columns.
35     m.reshape(4, 2)
36     m.debugLabel(bar_index + 20, note = "Reshaped to 4x2")
37
38     // Reshape. `m` now has 8 rows and 1 column.
39     m.reshape(8, 1)
40     m.debugLabel(bar_index + 30, note = "Reshaped to 8x1")

```

**Note that:**

- The order of elements in `m` does not change with each `m.reshape()` call.
- When reshaping a matrix, the product of the `rows` and `columns` arguments must equal the `matrix.elements_count()` value, as `matrix.reshape()` cannot change the number of elements in a matrix.

**Reversing**

One can reverse the order of all elements in a matrix using `matrix.reverse()`. This function moves the references of an  $m$ -by- $n$  matrix `id` at the  $i$ -th row and  $j$ -th column to the  $m - 1 - i$  row and  $n - 1 - j$  column.

For example, this script creates a 3x3 matrix containing the values 1-9 in ascending order, then uses the `reverse()` method to reverse its contents. It displays the original and modified versions of the matrix in labels on the chart via `m.debugLabel()`:



```

1 // @version=5
2 indicator("Reversing demo")
3
4 //@function Displays the rows of a matrix in a label with a note.
5 //@param this The matrix to display.
6 //@param barIndex The `bar_index` to display the label at.
7 //@param bgColor The background color of the label.
8 //@param textColor The color of the label's text.
9 //@param note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textcolor = textColor, size = size.huge
20         )
21
22 //@variable A 3x3 matrix.
23 matrix<float> m = matrix.new<float>()
24
25 // Add rows to `m`.
26 m.add_row(0, array.from(1, 2, 3))
27 m.add_row(1, array.from(4, 5, 6))
28 m.add_row(2, array.from(7, 8, 9))
29
30 if bar_index == last_bar_index - 1
31     // Display the contents of `m`.
32     m.debugLabel(note = "Original")
33     // Reverse `m`, then display its contents.
34     m.reverse()
35     m.debugLabel(bar_index + 10, color.red, note = "Reversed")

```

## Transposing

Transposing a matrix is a fundamental operation that flips all rows and columns in a matrix about its *main diagonal* (the diagonal vector of all values in which the row index equals the column index). This process produces a new matrix with reversed row and column dimensions, known as the *transpose*. Scripts can calculate the transpose of a matrix using `matrix.transpose()`.

For any  $m \times n$  matrix, the matrix returned from `matrix.transpose()` will have  $n$  rows and  $m$  columns. All elements in a matrix at the  $i$ -th row and  $j$ -th column correspond to the elements in its transpose at the  $j$ -th row and  $i$ -th column.

This example declares a  $2 \times 4$  `m` matrix, calculates its transpose using the `m.transpose()` method, and displays both matrices on the chart using our custom `debugLabel()` method. As we can see below, the transposed matrix has a  $4 \times 2$  shape, and the rows of the transpose match the columns of the original:

## Transpose example



**Original**  
[1, 2, 3, 4]  
[5, 6, 7, 8]

**Transpose**  
[1, 5]  
[2, 6]  
[3, 7]  
[4, 8]



```

1 //@version=5
2 indicator("Transpose example")
3
4 //@function Displays the rows of a matrix in a label with a note.
5 //@param    this The matrix to display.
6 //@param    barIndex The `bar_index` to display the label at.
7 //@param    bgColor The background color of the label.
8 //@param    textColor The color of the label's text.
9 //@param    note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ←center,
19             textColor = textColor, size = size.huge
20         )
21
22 //@variable A 2x4 matrix.
23 matrix<int> m = matrix.new<int>()
24
25 // Add columns to `m`.
26 m.add_col(0, array.from(1, 5))
27 m.add_col(1, array.from(2, 6))
28 m.add_col(2, array.from(3, 7))
29 m.add_col(3, array.from(4, 8))
30
31 //@variable The transpose of `m`. Has a 4x2 shape.
32 matrix<int> mt = m.transpose()
33
34 if bar_index == last_bar_index - 1
35     m.debugLabel(note = "Original")
            mt.debugLabel(bar_index + 10, note = "Transpose")

```

## Sorting

Scripts can sort the contents of a matrix via `matrix.sort()`. Unlike `array.sort()`, which sorts *elements*, this function organizes all *rows* in a matrix in a specified `order` (`order.ascending` by default) based on the values in a specified `column`.

This script declares a 3x3 `m` matrix, sorts the rows of the `m1` copy in ascending order based on the first column, then sorts the rows of the `m2` copy in descending order based on the second column. It displays the original matrix and sorted copies in labels using our `debugLabel()` method:

Sorting rows example

The screenshot shows a Pine Script code example with three sorted matrix labels. The first label, 'Original', has a blue background and contains the matrix [[3, 2, 4], [1, 9, 6], [7, 8, 9]]. The second label, 'Sorted using col 0 (Ascending)', has a green background and contains the matrix [[1, 9, 6], [3, 2, 4], [7, 8, 9]]. The third label, 'Sorted using col 1 (Descending)', has a red background and contains the matrix [[1, 9, 6], [7, 8, 9], [3, 2, 4]].

```

1 // @version=5
2 indicator("Sorting rows example")
3
4 // @function Displays the rows of a matrix in a label with a note.
5 // @param    this The matrix to display.
6 // @param    barIndex The `bar_index` to display the label at.
7 // @param    bgColor The background color of the label.
8 // @param    textColor The color of the label's text.
9 // @param    note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ←center,
19             textcolor = textColor, size = size.huge
20         )
21
22 // @variable A 3x3 matrix.
23 matrix<int> m = matrix.new<int>()
24
25 if bar_index == last_bar_index - 1
26     // Add rows to `m`.
27     m.add_row(0, array.from(3, 2, 4))
28     m.add_row(1, array.from(1, 9, 6))
29     m.add_row(2, array.from(7, 8, 9))
30     m.debugLabel(note = "Original")
31
32     // Copy `m` and sort rows in ascending order based on the first column (default).
33     matrix<int> m1 = m.copy()

```

(continues on next page)

(continued from previous page)

```

33     m1.sort()
34     m1.debugLabel(bar_index + 10, color.green, note = "Sorted using col 0\n(Ascending)
35     ↪")
36
37     // Copy `m` and sort rows in descending order based on the second column.
38     matrix<int> m2 = m.copy()
39     m2.sort(1, order.descending)
40     m2.debugLabel(bar_index + 20, color.red, note = "Sorted using col 1\n(Descending)
41     ↪")

```

It's important to note that `matrix.sort()` does not sort the columns of a matrix. However, one *can* use this function to sort matrix columns with the help of `matrix.transpose()`.

As an example, this script contains a `sortColumns()` method that uses the `sort()` method to sort the `transpose` of a matrix using the column corresponding to the `row` of the original matrix. The script uses this method to sort the `m` matrix based on the contents of its first row:

Sorting columns example

The screenshot shows a Pine Script code example titled "Sorting columns example". It displays two boxes side-by-side. The left box, with a blue background, is labeled "Original" and contains the matrix [[3, 2, 4], [1, 9, 6], [7, 8, 9]]. The right box, with a green background, is labeled "Sorted using row 0 (Ascending)" and contains the matrix [[2, 3, 4], [9, 1, 6], [8, 7, 9]].

```

1 // @version=5
2 indicator("Sorting columns example")
3
4 // @function Displays the rows of a matrix in a label with a note.
5 // @param    this The matrix to display.
6 // @param    barIndex The `bar_index` to display the label at.
7 // @param    bgColor The background color of the label.
8 // @param    textColor The color of the label's text.
9 // @param    note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textcolor = textColor, size = size.huge
20         )
21
// @function Sorts the columns of `this` matrix based on the values in the specified
22 ↪`row` .

```

(continues on next page)

(continued from previous page)

```

22 method sortColumns(matrix<int> this, int row = 0, bool ascending = true) =>
23     // @variable The transpose of `this` matrix.
24     matrix<int> thisT = this.transpose()
25     // @variable Is `order.ascending` when `ascending` is `true`, `order.descending` ↵
26     // otherwise.
27     order = ascending ? order.ascending : order.descending
28     // Sort the rows of `thisT` using the `row` column.
29     thisT.sort(row, order)
30     // @variable A copy of `this` matrix with sorted columns.
31     result = thisT.transpose()
32
33 // @variable A 3x3 matrix.
34 matrix<int> m = matrix.new<int>()
35
36 if bar_index == last_bar_index - 1
37     // Add rows to `m`.
38     m.add_row(0, array.from(3, 2, 4))
39     m.add_row(1, array.from(1, 9, 6))
40     m.add_row(2, array.from(7, 8, 9))
41     m.debugLabel(note = "Original")
42
43     // Sort the columns of `m` based on the first row and display the result.
44     m.sortColumns(0).debugLabel(bar_index + 10, note = "Sorted using row 0\"
45     ↵n(Ascending)")

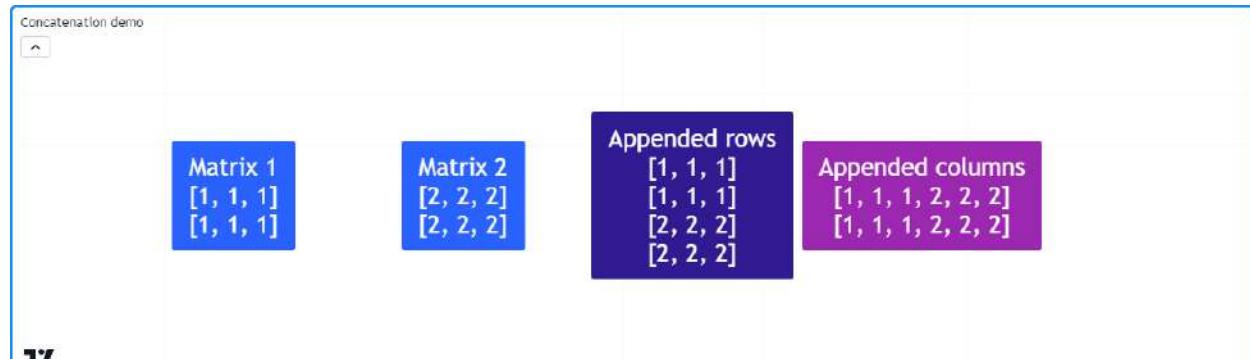
```

## Concatenating

Scripts can *concatenate* two matrices using `matrix.concat()`. This function appends the rows of an `id2` matrix to the end of an `id1` matrix with the same number of columns.

To create a matrix with elements representing the *columns* of a matrix appended to another, *transpose* both matrices, use `matrix.concat()` on the transposed matrices, then *transpose* the result.

For example, this script appends the rows of the `m2` matrix to the `m1` matrix and appends their columns using *transposed copies* of the matrices. It displays the `m1` and `m2` matrices and the results after concatenating their rows and columns in labels using the custom `debugLabel()` method:



```

1 // @version=5
2 indicator("Concatenation demo")
3
4 // @function Displays the rows of a matrix in a label with a note.
5 // @param    this The matrix to display.

```

(continues on next page)

(continued from previous page)

```

6 // @param barIndex The `bar_index` to display the label at.
7 // @param bgColor The background color of the label.
8 // @param textColor The color of the label's text.
9 // @param note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textcolor = textColor, size = size.huge
20         )
21
22 // @variable A 2x3 matrix filled with 1s.
23 matrix<int> m1 = matrix.new<int>(2, 3, 1)
24 // @variable A 2x3 matrix filled with 2s.
25 matrix<int> m2 = matrix.new<int>(2, 3, 2)
26
27 // @variable The transpose of `m1`.
28 t1 = m1.transpose()
29 // @variable The transpose of `m2`.
30 t2 = m2.transpose()
31
32 if bar_index == last_bar_index - 1
33     // Display the original matrices.
34     m1.debugLabel(note = "Matrix 1")
35     m2.debugLabel(bar_index + 10, note = "Matrix 2")
36     // Append the rows of `m2` to the end of `m1` and display `m1`.
37     m1.concat(m2)
38     m1.debugLabel(bar_index + 20, color.blue, note = "Appended rows")
39     // Append the rows of `t2` to the end of `t1`, then display the transpose of `t1`.
40     t1.concat(t2)
41     t1.transpose().debugLabel(bar_index + 30, color.purple, note = "Appended columns")

```

### 3.15.10 Matrix calculations

#### Element-wise calculations

Pine scripts can calculate the *average*, *minimum*, *maximum*, and *mode* of all elements within a matrix via `matrix.avg()`, `matrix.min()`, `matrix.max()`, and `matrix.mode()`. These functions operate the same as their `array.*` equivalents, allowing users to run element-wise calculations on a matrix, its *submatrices*, and its *rows and columns* using the same syntax. For example, the built-in `*.avg()` functions called on a 3x3 matrix with values 1-9 and an `array` with the same nine elements will both return a value of 5.

The script below uses `*.avg()`, `*.max()`, and `*.min()` methods to calculate developing averages and extremes of OHLC data in a period. It adds a new column of `open`, `high`, `low`, and `close` values to the end of the `ohlcData` matrix whenever `queueColumn` is `true`. When `false`, the script uses the `get()` and `set()` matrix methods to adjust the elements in the last column for developing HLC values in the current period. It uses the `ohlcData` matrix, a `submatrix()`, and `row()` and `col()` arrays to calculate the developing OHLC4 and HL2 averages over `length` periods, the maximum high and minimum low over `length` periods, and the current period's developing OHLC4 price:



```

1 //@version=5
2 indicator("Element-wise calculations example", "Developing values", overlay = true)
3
4 //@variable The number of data points in the averages.
5 int length = input.int(3, "Length", 1)
6 //@variable The timeframe of each reset period.
7 string timeframe = input.timeframe("D", "Reset Timeframe")
8
9 //@variable A 4x`length` matrix of OHLC values.
10 var matrix<float> ohlcData = matrix.new<float>(4, length)
11
12 //@variable Is `true` at the start of a new bar at the `timeframe`.
13 bool queueColumn = timeframe.change(timeframe)
14
15 if queueColumn
16     // Add new values to the end column of `ohlcData`.
17     ohlcData.add_col(length, array.from(open, high, low, close))
18     // Remove the oldest column from `ohlcData`.
19     ohlcData.remove_col(0)
20 else
21     // Adjust the last element of column 1 for new highs.
22     if high > ohlcData.get(1, length - 1)
23         ohlcData.set(1, length - 1, high)
24     // Adjust the last element of column 2 for new lows.
25     if low < ohlcData.get(2, length - 1)
26         ohlcData.set(2, length - 1, low)
27     // Adjust the last element of column 3 for the new closing price.
28     ohlcData.set(3, length - 1, close)
29
30 //@variable The `matrix.avg()` of all elements in `ohlcData`.
31 avgOHLC4 = ohlcData.avg()
32 //@variable The `matrix.avg()` of all elements in rows 1 and 2, i.e., the average of
33 // all `high` and `low` values.
34 avgHL2   = ohlcData.submatrix(from_row = 1, to_row = 3).avg()
35 //@variable The `matrix.max()` of all values in `ohlcData`. Equivalent to `ohlcData.

```

(continues on next page)

(continued from previous page)

```

35  ↵row(1).max()`.
maxHigh = ohlcData.max()
//@variable The `array.min()` of all `low` values in `ohlcData`. Equivalent to
36  ↵`ohlcData.min()`.
minLow = ohlcData.row(2).min()
37 //@variable The `array.avg()` of the last column in `ohlcData`, i.e., the current
38 // OHLC4.
39 ohlc4Value = ohlcData.col(length - 1).avg()
40
41 plot(avgOHLC4, "Average OHLC4", color.purple, 2)
42 plot(avgHL2, "Average HL2", color.navy, 2)
43 plot(maxHigh, "Max High", color.green)
44 plot(minLow, "Min Low", color.red)
45 plot(ohlc4Value, "Current OHLC4", color.blue)

```

**Note that:**

- In this example, we used `array.*()` and `matrix.*()` methods interchangeably to demonstrate their similarities in syntax and behavior.
- Users can calculate the matrix equivalent of `array.sum()` by multiplying the `matrix.avg()` by the `matrix.elements_count()`.

**Special calculations**

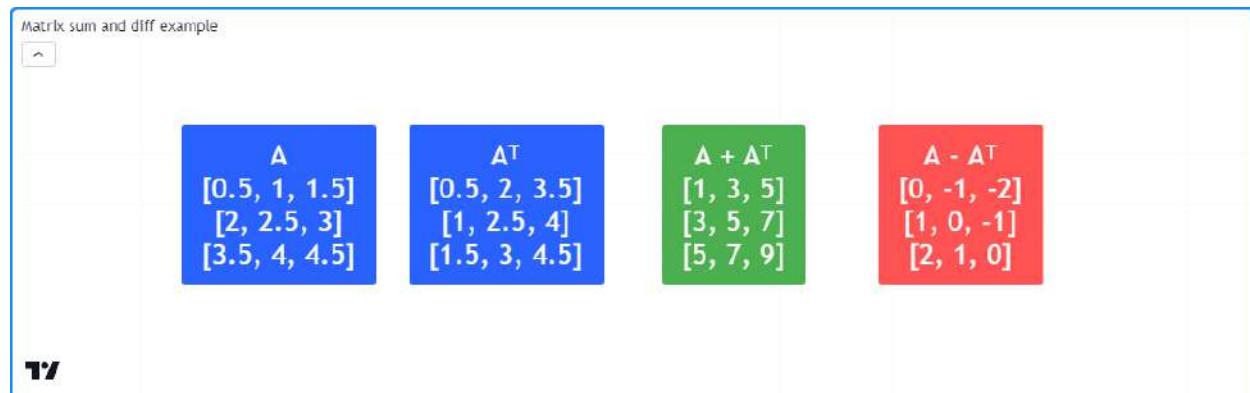
Pine Script™ features several built-in functions for performing essential matrix arithmetic and linear algebra operations, including `matrix.sum()`, `matrix.diff()`, `matrix.mult()`, `matrix.pow()`, `matrix.det()`, `matrix.inv()`, `matrix.pinv()`, `matrix.rank()`, `matrix.trace()`, `matrix.eigenvalues()`, `matrix.eigenvectors()`, and `matrix.kron()`. These functions are advanced features that facilitate a variety of matrix calculations and transformations.

Below, we explain a few fundamental functions with some basic examples.

**`matrix.sum()` and `matrix.diff()`**

Scripts can perform addition and subtraction of two matrices with the same shape or a matrix and a scalar value using the `matrix.sum()` and `matrix.diff()` functions. These functions use the values from the `id2` matrix or scalar to add to or subtract from the elements in `id1`.

This script demonstrates a simple example of matrix addition and subtraction in Pine. It creates a 3x3 matrix, calculates its *transpose*, then calculates the `matrix.sum()` and `matrix.diff()` of the two matrices. This example displays the original matrix, its *transpose*, and the resulting sum and difference matrices in labels on the chart:



```

1 // @version=5
2 indicator("Matrix sum and diff example")
3
4 // @function Displays the rows of a matrix in a label with a note.
5 // @param    this The matrix to display.
6 // @param    barIndex The `bar_index` to display the label at.
7 // @param    bgColor The background color of the label.
8 // @param    textColor The color of the label's text.
9 // @param    note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textcolor = textColor, size = size.huge
20         )
21
22 // @variable A 3x3 matrix.
23 m = matrix.new<float>()
24
25 // Add rows to `m`.
26 m.add_row(0, array.from(0.5, 1.0, 1.5))
27 m.add_row(1, array.from(2.0, 2.5, 3.0))
28 m.add_row(2, array.from(3.5, 4.0, 4.5))
29
30 if bar_index == last_bar_index - 1
31     // Display `m`.
32     m.debugLabel(note = "A")
33     // Get and display the transpose of `m`.
34     matrix<float> t = m.transpose()
35     t.debugLabel(bar_index + 10, note = "AT")
36     // Calculate the sum of the two matrices. The resulting matrix is symmetric.
37     matrix.sum(m, t).debugLabel(bar_index + 20, color.green, note = "A + AT")
38     // Calculate the difference between the two matrices. The resulting matrix is
39     ↪antisymmetric.
40     matrix.diff(m, t).debugLabel(bar_index + 30, color.red, note = "A - AT")

```

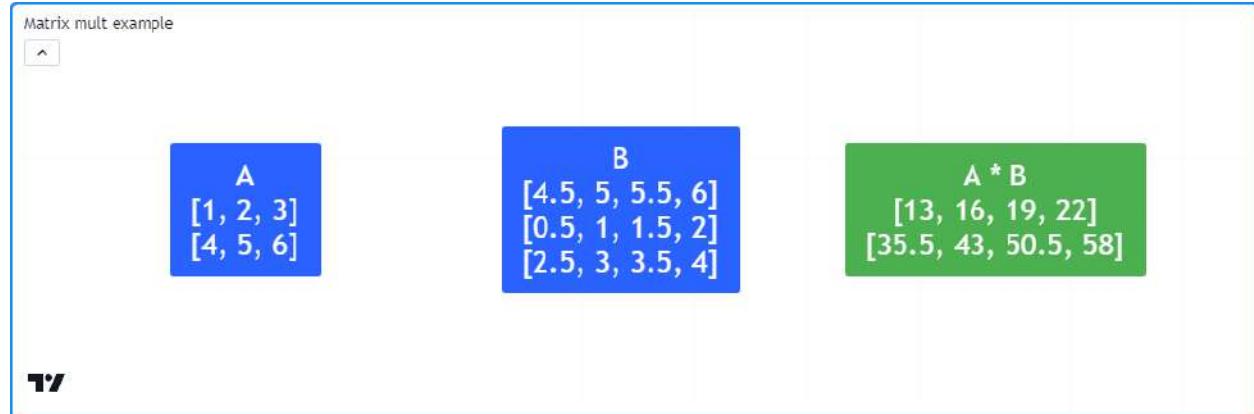
**Note that:**

- In this example, we've labeled the original matrix as “A” and the transpose as “ $A^T$ ”.
- Adding “A” and “ $A^T$ ” produces a **symmetric** matrix, and subtracting them produces an **antisymmetric** matrix.

## `matrix.mult()`

Scripts can multiply two matrices via the `matrix.mult()` function. This function also facilitates the multiplication of a matrix by an `array` or a scalar value.

In the case of multiplying two matrices, unlike addition and subtraction, matrix multiplication does not require two matrices to share the same shape. However, the number of columns in the first matrix must equal the number of rows in the second one. The resulting matrix returned by `matrix.mult()` will contain the same number of rows as `id1` and the same number of columns as `id2`. For instance, a 2x3 matrix multiplied by a 3x4 matrix will produce a matrix with two rows and four columns, as shown below. Each value within the resulting matrix is the `dot product` of the corresponding row in `id1` and column in `id2`:



```

1 // @version=5
2 indicator("Matrix mult example")
3
4 //@function Displays the rows of a matrix in a label with a note.
5 //@param    this The matrix to display.
6 //@param    barIndex The `bar_index` to display the label at.
7 //@param    bgColor The background color of the label.
8 //@param    textColor The color of the label's text.
9 //@param    note The text to display above the rows.
10 method debugLabel(
11     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
12     color textColor = color.white, string note = ""
13 ) =>
14     labelText = note + "\n" + str.tostring(this)
15     if barstate.ishistory
16         label.new(
17             barIndex, 0, labelText, color = bgColor, style = label.style_label_
18             ↪center,
19             textColor = textColor, size = size.huge
20         )
21
22 //@variable A 2x3 matrix.
23 a = matrix.new<float>()
24 //@variable A 3x4 matrix.
25 b = matrix.new<float>()
26
27 // Add rows to `a`.
28 a.add_row(0, array.from(1, 2, 3))
29 a.add_row(1, array.from(4, 5, 6))

```

(continues on next page)

(continued from previous page)

```

30 // Add rows to `b`.
31 b.add_row(0, array.from(0.5, 1.0, 1.5, 2.0))
32 b.add_row(1, array.from(2.5, 3.0, 3.5, 4.0))
33 b.add_row(0, array.from(4.5, 5.0, 5.5, 6.0))
34
35 if bar_index == last_bar_index - 1
36     // @variable The result of `a` * `b`.
37     matrix<float> ab = a.mult(b)
38     // Display `a`, `b`, and `ab` matrices.
39     debugLabel(a, note = "A")
40     debugLabel(b, bar_index + 10, note = "B")
41     debugLabel(ab, bar_index + 20, color.green, note = "A * B")

```

**Note that:**

- In contrast to the multiplication of scalars, matrix multiplication is *non-commutative*, i.e., `matrix.mult(a, b)` does not necessarily produce the same result as `matrix.mult(b, a)`. In the context of our example, the latter will raise a runtime error because the number of columns in `b` doesn't equal the number of rows in `a`.

When multiplying a matrix and an `array`, this function treats the operation the same as multiplying `id1` by a single-column matrix, but it returns an `array` with the same number of elements as the number of rows in `id1`. When `matrix.mult()` passes a scalar as its `id2` value, the function returns a new matrix whose elements are the elements in `id1` multiplied by the `id2` value.

**`matrix.det()`**

A *determinant* is a scalar value associated with a `square` matrix that describes some of its characteristics, namely its invertibility. If a matrix has an `inverse`, its determinant is nonzero. Otherwise, the matrix is *singular* (non-invertible). Scripts can calculate the determinant of a matrix via `matrix.det()`.

Programmers can use determinants to detect similarities between matrices, identify *full-rank* and *rank-deficient* matrices, and solve systems of linear equations, among other applications.

For example, this script utilizes determinants to solve a system of linear equations with a matching number of unknown values using `Cramer's rule`. The user-defined `solve()` function returns an `array` containing solutions for each unknown value in the system, where the n-th element of the array is the determinant of the coefficient matrix with the n-th column replaced by the column of constants divided by the determinant of the original coefficients.

In this script, we've defined the matrix `m` that holds coefficients and constants for these three equations:

```

3 * x0 + 4 * x1 - 1 * x2 = 8
5 * x0 - 2 * x1 + 1 * x2 = 4
2 * x0 - 2 * x1 + 1 * x2 = 1

```

The solution to this system is ( $x_0 = 1$ ,  $x_1 = 2$ ,  $x_2 = 3$ ). The script calculates these values from `m` via `m.solve()` and plots them on the chart:



```

1 // @version=5
2 indicator("Determinants example", "Cramer's Rule")
3
4 // @function Solves a system of linear equations with a matching number of unknowns
5 // using Cramer's rule.
6 // @param    this An augmented matrix containing the coefficients for each unknown and
7 //           the results of
8 //           the equations. For example, a row containing the values 2, -1, and 3
9 //           represents the equation
10 //           `2 * x0 + (-1) * x1 = 3`, where `x0` and `x1` are the unknown values in
11 //           the system.
12 // @returns An array containing solutions for each variable in the system.
13 solve(matrix<float> this) =>
14     // @variable The coefficient matrix for the system of equations.
15     matrix<float> coefficients = this.submatrix(from_column = 0, to_column = this.
16         columns() - 1)
17     // @variable The array of resulting constants for each equation.
18     array<float> constants = this.col(this.columns() - 1)
19     // @variable An array containing solutions for each unknown in the system.
20     array<float> result = array.new<float>()
21
22     // @variable The determinant value of the coefficient matrix.
23     float baseDet = coefficients.det()
24     matrix<float> modified = na
25     for col = 0 to coefficients.columns() - 1
26         modified := coefficients.copy()
27         modified.add_col(col, constants)
28         modified.remove_col(col + 1)
29
30         // Calculate the solution for the column's unknown by dividing the
31         // determinant of `modified` by the `baseDet`.
32         result.push(modified.det() / baseDet)
33
34     result
35
36 // @variable A 3x4 matrix containing coefficients and results for a system of three
37 // equations.
38 m = matrix.new<float>()
39
40 // Add rows for the following equations:
41 // Equation 1: 3 * x0 + 4 * x1 - 1 * x2 = 8
42 // Equation 2: 5 * x0 - 2 * x1 + 1 * x2 = 4
43 // Equation 3: 2 * x0 - 2 * x1 + 1 * x2 = 1
44 m.add_row(0, array.from(3.0, 4.0, -1.0, 8.0))
45 m.add_row(1, array.from(5.0, -2.0, 1.0, 4.0))
46 m.add_row(2, array.from(2.0, -2.0, 1.0, 1.0))

```

(continues on next page)

(continued from previous page)

```

40 // @variable An array of solutions to the unknowns in the system of equations
41 // represented by `m`.
42 solutions = solve(m)
43
44 plot(solutions.get(0), "x0", color.red, 3) // Plots 1.
45 plot(solutions.get(1), "x1", color.green, 3) // Plots 2.
46 plot(solutions.get(2), "x2", color.blue, 3) // Plots 3.

```

**Note that:**

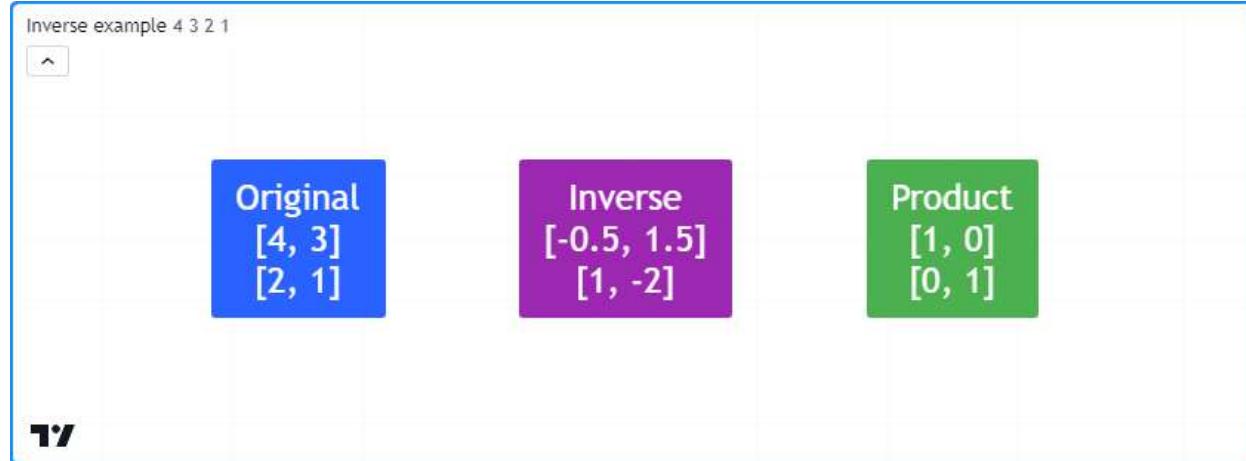
- Solving systems of equations is particularly useful for *regression analysis*, e.g., linear and polynomial regression.
- Cramer's rule works fine for small systems of equations. However, it's computationally inefficient on larger systems. Other methods, such as [Gaussian elimination](#), are often preferred for such use cases.

**`matrix.inv()` and `matrix.pinv()`**

For any non-singular [square](#) matrix, there is an inverse matrix that yields the [identity](#) matrix when *multiplied* by the original. Inverses have utility in various matrix transformations and solving systems of equations. Scripts can calculate the inverse of a matrix **when one exists** via the [matrix.inv\(\)](#) function.

For singular (non-invertible) matrices, one can calculate a generalized inverse ([pseudoinverse](#)), regardless of whether the matrix is square or has a nonzero determinant, via the [matrix.pinv\(\)](#) function. Keep in mind that unlike a true inverse, the product of a pseudoinverse and the original matrix does not necessarily equal the identity matrix unless the original matrix is *invertible*.

The following example forms a 2x2 `m` matrix from user inputs, then uses the [m.inv\(\)](#) and [m.pinv\(\)](#) methods to calculate the inverse or pseudoinverse of `m`. The script displays the original matrix, its inverse or pseudoinverse, and their product in labels on the chart:



```

1 // @version=5
2 indicator("Inverse example")
3
4 // Element inputs for the 2x2 matrix.
5 float r0c0 = input.float(4.0, "Row 0, Col 0")
6 float r0c1 = input.float(3.0, "Row 0, Col 1")
7 float r1c0 = input.float(2.0, "Row 1, Col 0")

```

(continues on next page)

(continued from previous page)

```

8 float r1c1 = input.float(1.0, "Row 1, Col 1")
9
10 // @function Displays the rows of a matrix in a label with a note.
11 // @param this The matrix to display.
12 // @param barIndex The `bar_index` to display the label at.
13 // @param bgColor The background color of the label.
14 // @param textColor The color of the label's text.
15 // @param note The text to display above the rows.
16 method debugLabel(
17     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
18     color textColor = color.white, string note = ""
19 ) =>
20     labelText = note + "\n" + str.tostring(this)
21     if barstate.ishistory
22         label.new(
23             barIndex, 0, labelText, color = bgColor, style = label.style_label_
24             ↪center,
25             textColor = textColor, size = size.huge
26         )
27
28 // @variable A 2x2 matrix of input values.
29 m = matrix.new<float>()
30
31 // Add input values to `m`.
32 m.add_row(0, array.from(r0c0, r0c1))
33 m.add_row(1, array.from(r1c0, r1c1))
34
35 // @variable Is `true` if `m` is square with a nonzero determinant, indicating_
36 // ↪invertibility.
37 bool isInvertible = m.is_square() and m.det()
38
39 // @variable The inverse or pseudoinverse of `m`.
40 mInverse = isInvertible ? m.inv() : m.pinv()
41
42 // @variable The product of `m` and `mInverse`. Returns the identity matrix when_
43 // ↪`isInvertible` is `true`.
44 matrix<float> product = m.mult(mInverse)
45
46 if bar_index == last_bar_index - 1
47     // Display `m`, `mInverse`, and their `product`.
48     m.debugLabel(note = "Original")
49     mInverse.debugLabel(bar_index + 10, color.purple, note = isInvertible ? "Inverse"_
50       ↪: "Pseudoinverse")
51     product.debugLabel(bar_index + 20, color.green, note = "Product")

```

**Note that:**

- This script will only call `m.inv()` when `isInvertible` is `true`, i.e., when `m` is `square` and has a nonzero `determinant`. Otherwise, it uses `m.pinv()` to calculate the generalized inverse.

### ``matrix.rank()``

The *rank* of a matrix represents the number of linearly independent vectors (rows or columns) it contains. In essence, matrix rank measures the number of vectors one cannot express as a linear combination of others, or in other words, the number of vectors that contain **unique** information. Scripts can calculate the rank of a matrix via `matrix.rank()`.

This script identifies the number of linearly independent vectors in two 3x3 matrices (`m1` and `m2`) and plots the values in a separate pane. As we see on the chart, the `m1.rank()` value is 3 because each vector is unique. The `m2.rank()` value, on the other hand, is 1 because it has just one unique vector:



```

1 // @version=5
2 indicator("Matrix rank example")
3
4 //@variable A 3x3 full-rank matrix.
5 m1 = matrix.new<float>()
6 //@variable A 3x3 rank-deficient matrix.
7 m2 = matrix.new<float>()
8
9 // Add linearly independent vectors to `m1`.
10 m1.add_row(0, array.from(3, 2, 3))
11 m1.add_row(1, array.from(4, 6, 6))
12 m1.add_row(2, array.from(7, 4, 9))
13
14 // Add linearly dependent vectors to `m2`.
15 m2.add_row(0, array.from(1, 2, 3))
16 m2.add_row(1, array.from(2, 4, 6))
17 m2.add_row(2, array.from(3, 6, 9))
18
19 // Plot `matrix.rank()` values.
20 plot(m1.rank(), color = color.green, linewidth = 3)
21 plot(m2.rank(), color = color.red, linewidth = 3)

```

#### Note that:

- The highest rank value a matrix can have is the minimum of its number of rows and columns. A matrix with the maximum possible rank is known as a *full-rank* matrix, and any matrix without full rank is known as a *rank-deficient* matrix.
- The *determinants* of full-rank square matrices are nonzero, and such matrices have *inverses*. Conversely, the *determinant* of a rank-deficient matrix is always 0.
- For any matrix that contains nothing but the same value in each of its elements (e.g., a matrix filled with 0), the rank is always 0 since none of the vectors hold unique information. For any other matrix with distinct

values, the minimum possible rank is 1.

### 3.15.11 Error handling

In addition to usual **compiler** errors, which occur during a script's compilation due to improper syntax, scripts using matrices can raise specific **runtime** errors during their execution. When a script raises a runtime error, it displays a red exclamation point next to the script title. Users can view the error message by clicking this icon.

In this section, we discuss runtime errors that users may encounter while utilizing matrices in their scripts.

#### The row/column index (xx) is out of bounds, row/column size is (yy).

This runtime error occurs when trying to access indices outside the matrix dimensions with functions including `matrix.get()`, `matrix.set()`, `matrix.fill()`, and `matrix.submatrix()`, as well as some of the functions relating to the *rows and columns* of a matrix.

For example, this code contains two lines that will produce this runtime error. The `m.set()` method references a `row` index that doesn't exist (2). The `m.submatrix()` method references all column indices up to `to_column - 1`. A `to_column` value of 4 results in a runtime error because the last column index referenced (3) does not exist in `m`:

```

1 // @version=5
2 indicator("Out of bounds demo")
3
4 // @variable A 2x3 matrix with a max row index of 1 and max column index of 2.
5 matrix<float> m = matrix.new<float>(2, 3, 0.0)
6
7 m.set(row = 2, column = 0, value = 1.0)      // The `row` index is out of bounds on_
8     //this line. The max value is 1.
9 m.submatrix(from_column = 1, to_column = 4) // The `to_column` index is invalid on_
10    //this line. The max value is 3.
11
12 if bar_index == last_bar_index - 1
13     label.new(bar_index, 0, str.tostring(m), color = color.navy, textcolor = color.
14         white, size = size.huge)

```

Users can avoid this error in their scripts by ensuring their function calls do not reference indices greater than or equal to the number of rows/columns.

#### The array size does not match the number of rows/columns in the matrix.

When using `matrix.add_row()` and `matrix.add_col()` functions to *insert* rows and columns into a non-empty matrix, the size of the inserted array must align with the matrix dimensions. The size of an inserted row must match the number of columns, and the size of an inserted column must match the number of rows. Otherwise, the script will raise this runtime error. For example:

```

1 // @version=5
2 indicator("Invalid array size demo")
3
4 // Declare an empty matrix.
5 m = matrix.new<float>()
6
7 m.add_col(0, array.from(1, 2))      // Add a column. Changes the shape of `m` to 2x1.
8 m.add_col(1, array.from(1, 2, 3)) // Raises a runtime error because `m` has 2 rows,_
9     //not 3.

```

(continues on next page)

(continued from previous page)

```

9
10 plot(m.col(0).get(1))

```

**Note that:**

- When `m` is empty, one can insert a row or column array of *any* size, as shown in the first `m.add_col()` line.

**Cannot call matrix methods when the ID of matrix is 'na'.**

When a matrix variable is assigned to `na`, it means that the variable doesn't reference an existing object. Consequently, one cannot use built-in `matrix.*()` functions and methods with it. For example:

```

1 // @version=5
2 indicator("na matrix methods demo")
3
4 // @variable A `matrix` variable assigned to `na`.
5 matrix<float> m = na
6
7 mCopy = m.copy() // Raises a runtime error. You can't copy a matrix that doesn't exist.
8
9 if bar_index == last_bar_index - 1
10    label.new(bar_index, 0, str.tostring(mCopy), color = color.navy, textcolor = color.white, size = size.huge)

```

To resolve this error, assign `m` to a valid matrix instance before using `matrix.*()` functions.

**Matrix is too large. Maximum size of the matrix is 100,000 elements.**

The total number of elements in a matrix (`matrix.elements_count()`) cannot exceed **100,000**, regardless of its shape. For example, this script will raise an error because it *inserts* 1000 rows with 101 elements into the `m` matrix:

```

1 // @version=5
2 indicator("Matrix too large demo")
3
4 var matrix<float> m = matrix.new<float>()
5
6 if bar_index == 0
7    for i = 1 to 1000
8       // This raises an error because the script adds 101 elements on each iteration.
9       // 1000 rows * 101 elements per row = 101000 total elements. This is too large.
10      m.add_row(m.rows(), array.new<float>(101, i))
11
12 plot(m.get(0, 0))

```

### The row/column index must be 0 <= from\_row/column < to\_row/column.

When using `matrix.*()` functions with `from_row/column` and `to_row/column` indices, the `from_*` values must be less than the corresponding `to_*` values, with the minimum possible value being 0. Otherwise, the script will raise a runtime error.

For example, this script shows an attempt to declare a `submatrix` from a 4x4 `m` matrix with a `from_row` value of 2 and a `to_row` value of 2, which will result in an error:

```

1 // @version=5
2 indicator("Invalid from_row, to_row demo")
3
4 //@variable A 4x4 matrix filled with a random value.
5 matrix<float> m = matrix.new<float>(4, 4, math.random())
6
7 matrix<float> mSub = m.submatrix(from_row = 2, to_row = 2) // Raises an error. `from_
8   ↪row` can't equal `to_row`.
9 plot(mSub.get(0, 0))

```

### Matrices 'id1' and 'id2' must have an equal number of rows and columns to be added.

When using `matrix.sum()` and `matrix.diff()` functions, the `id1` and `id2` matrices must have the same number of rows and the same number of columns. Attempting to add or subtract two matrices with mismatched dimensions will raise an error, as demonstrated by this code:

```

1 // @version=5
2 indicator("Invalid sum dimensions demo")
3
4 //@variable A 2x3 matrix.
5 matrix<float> m1 = matrix.new<float>(2, 3, 1)
6 //@variable A 3x4 matrix.
7 matrix<float> m2 = matrix.new<float>(3, 4, 2)
8
9 mSum = matrix.sum(m1, m2) // Raises an error. `m1` and `m2` don't have matching_
10   ↪dimensions.
11 plot(mSum.get(0, 0))

```

### The number of columns in the 'id1' matrix must equal the number of rows in the matrix (or the number of elements in the array) 'id2'.

When using `matrix.mult()` to multiply an `id1` matrix by an `id2` matrix or array, the `matrix.rows()` or `array.size()` of `id2` must equal the `matrix.columns()` in `id1`. If they don't align, the script will raise this error.

For example, this script tries to multiply two 2x3 matrices. While *adding* these matrices is possible, *multiplying* them is not:

```

1 // @version=5
2 indicator("Invalid mult dimensions demo")
3
4 //@variable A 2x3 matrix.
5 matrix<float> m1 = matrix.new<float>(2, 3, 1)
6 //@variable A 2x3 matrix.
7 matrix<float> m2 = matrix.new<float>(2, 3, 2)

```

(continues on next page)

(continued from previous page)

```

8 mSum = matrix.mult(m1, m2) // Raises an error. The number of columns in `m1` and rows
9   ↪in `m2` aren't equal.
10
11 plot(mSum.get(0, 0))

```

**Operation not available for non-square matrices.**

Some matrix operations, including `matrix.inv()`, `matrix.det()`, `matrix.eigenvalues()`, and `matrix.eigenvectors()` only work with **square** matrices, i.e., matrices with the same number of rows and columns. When attempting to execute such functions on non-square matrices, the script will raise an error stating the operation isn't available or that it cannot calculate the result for the matrix id. For example:

```

1 // @version=5
2 indicator("Non-square demo")
3
4 // @variable A 3x5 matrix.
5 matrix<float> m = matrix.new<float>(3, 5, 1)
6
7 plot(m.det()) // Raises a runtime error. You can't calculate the determinant of a 3x5
8   ↪matrix.

```



Advanced



## 3.16 Maps

- *Introduction*
- *Declaring a map*
- *Reading and writing*
- *Looping through a map*
- *Copying a map*
- *Scope and history*
- *Maps of other collections*

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

### 3.16.1 Introduction

Pine Script™ Maps are collections that store elements in *key-value pairs*. They allow scripts to collect multiple value references associated with unique identifiers (keys).

Unlike *arrays* and *matrices*, maps are considered *unordered* collections. Scripts quickly access a map's values by referencing the keys from the key-value pairs put into them rather than traversing an internal index.

A map's keys can be of any *fundamental type*, and its values can be of any built-in or *user-defined* type. Maps cannot directly use other *collections* (maps, *arrays*, or *matrices*) as values, but they can hold *UDT* instances containing these data structures within their fields. See [this section](#) for more information.

As with other collections, maps can contain up to 100,000 elements in total. Since each key-value pair in a map consists of two elements (a unique *key* and its associated *value*), the maximum number of key-value pairs a map can hold is 50,000.

### 3.16.2 Declaring a map

Pine Script™ uses the following syntax to declare maps:

```
[var/varip] [map<keyType, valueType>] <identifier> = <expression>
```

Where `<keyType, valueType>` is the map's *type template* that declares the types of keys and values it will contain, and the `<expression>` returns either a map instance or na.

When declaring a map variable assigned to na, users must include the `map` keyword followed by a *type template* to tell the compiler that the variable can accept maps with `keyType` keys and `valueType` values.

For example, this line of code declares a new `myMap` variable that can accept map instances holding pairs of `string` keys and `float` values:

```
map<string, float> myMap = na
```

When the `<expression>` is not na, the compiler does not require explicit type declaration, as it will infer the type information from the assigned map object.

This line declares a `myMap` variable assigned to an empty map with `string` keys and `float` values. Any maps assigned to this variable later must have the same key and value types:

```
myMap = map.new<string, float>()
```

#### Using `var` and `varip` keywords

Users can include the `var` or `varip` keywords to instruct their scripts to declare map variables only on the first chart bar. Variables that use these keywords point to the same map instances on each script iteration until explicitly reassigned.

For example, this script declares a `colorMap` variable assigned to a map that holds pairs of `string` keys and `color` values on the first chart bar. The script displays an `oscillator` on the chart and uses the values it `put` into the `colorMap` on the *first* bar to color the plots on *all* bars:



```

1 // @version=5
2 indicator("var map demo")
3
4 // @variable A map associating color values with string keys.
5 var colorMap = map.new<string, color>()
6
7 // Put `<string, color>` pairs into `colorMap` on the first bar.
8 if bar_index == 0
9     colorMap.put("Bull", color.green)
10    colorMap.put("Bear", color.red)
11    colorMap.put("Neutral", color.gray)
12
13 // @variable The 14-bar RSI of `close`.
14 float oscillator = ta.rsi(close, 14)
15
16 // @variable The color of the `oscillator`.
17 color oscColor = switch
18     oscillator > 50 => colorMap.get("Bull")
19     oscillator < 50 => colorMap.get("Bear")
20     => colorMap.get("Neutral")
21
22 // Plot the `oscillator` using the `oscColor` from our `colorMap`.
23 plot(oscillator, "Histogram", oscColor, 2, plot.style_histogram, histbase = 50)
24 plot(oscillator, "Line", oscColor, 3)

```

**Note:** Map variables declared using `varip` behave as ones using `var` on historical data, but they update their key-value pairs for realtime bars (i.e., the bars since the script's last compilation) on each new price tick. Maps assigned to `varip` variables can only hold values of `int`, `float`, `bool`, `color`, or `string` types or *user-defined types* that exclusively contain within their fields these types or collections (*arrays*, *matrices*, or maps) of these types.

---

### 3.16.3 Reading and writing

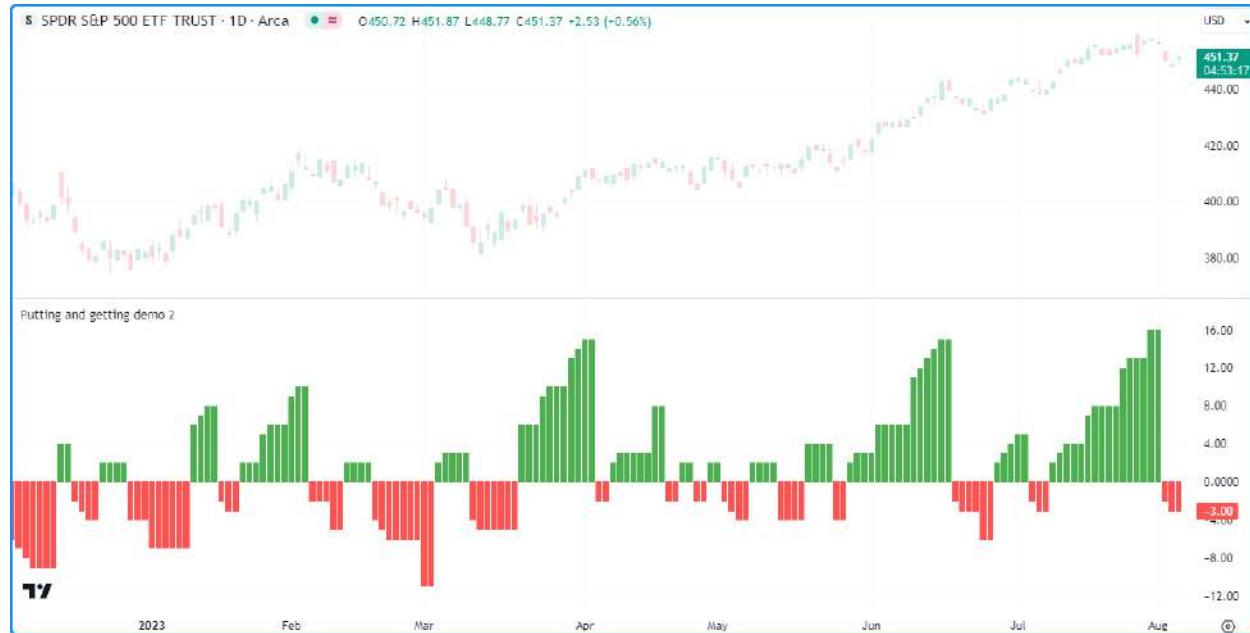
#### Putting and getting key-value pairs

The `map.put()` function is one that map users will utilize quite often, as it's the primary method to put a new key-value pair into a map. It associates the `key` argument with the `value` argument in the call and adds the pair to the map `id`.

If the `key` argument in the `map.put()` call already exists in the map's `keys`, the new pair passed into the function will **replace** the existing one.

To retrieve the value from a map `id` associated with a given `key`, use `map.get()`. This function returns the value if the `id` map `contains` the `key`. Otherwise, it returns `na`.

The following example calculates the difference between the `bar_index` values from when `close` was last `rising` and `falling` over a given `length` with the help of `map.put()` and `map.get()` methods. The script puts a ("Rising", `bar_index`) pair into the data map when the price is rising and puts a ("Falling", `bar_index`) pair into the map when the price is falling. It then puts a pair containing the "Difference" between the "Rising" and "Falling" values into the map and plots its value on the chart:



```

1 // @version=5
2 indicator("Putting and getting demo")
3
4 //@variable The length of the `ta.rising()` and `ta.falling()` calculation.
5 int length = input.int(2, "Length")
6
7 //@variable A map associating `string` keys with `int` values.
8 var data = map.new<string, int>()
9
10 // Put a new ("Rising", `bar_index`) pair into the `data` map when `close` is rising.
11 if ta.rising(close, length)
12     data.put("Rising", bar_index)
13 // Put a new ("Falling", `bar_index`) pair into the `data` map when `close` is falling.
14 if ta.falling(close, length)
15     data.put("Falling", bar_index)

```

(continues on next page)

(continued from previous page)

```

16
17 // Put the "Difference" between current "Rising" and "Falling" values into the `data` ↵
18 ↵map.
19 data.put("Difference", data.get("Rising") - data.get("Falling"))
20
21 // @variable The difference between the last "Rising" and "Falling" `bar_index`.
22 int index = data.get("Difference")
23
24 // @variable Returns `color.green` when `index` is positive, `color.red` when negative,
25 ↵ and `color.gray` otherwise.
26 color indexColor = index > 0 ? color.green : index < 0 ? color.red : color.gray
27
28 plot(index, color = indexColor, style = plot.style_columns)

```

### Note that:

- This script replaces the values associated with the “Rising”, “Falling”, and “Difference” keys on successive `data.put()` calls, as each of these keys is unique and can only appear once in the `data` map.
- Replacing the pairs in a map does not change the internal *insertion order* of its keys. We discuss this further in the [next section](#).

Similar to working with other collections, when putting a value of a *special type* (`line`, `linefill`, `box`, `polyline`, `label`, `table`, or `chart.point`) or a *user-defined type* into a map, it's important to note the inserted pair's `value` points to that same object without copying it. Modifying the value referenced by a key-value pair will also affect the original object.

For example, this script contains a custom `ChartData` type with `o`, `h`, `l`, and `c` fields. On the first chart bar, the script declares a `myMap` variable and adds the pair `("A", myData)`, where `myData` is a `ChartData` instance with initial field values of `na`. It adds the pair `("B", myData)` to `myMap` and updates the object from this pair on every bar via the user-defined `update()` method.

Each change to the object with the “B” key affects the one referenced by the “A” key, as shown by the candle plot of the “A” object's fields:



```

1 // @version=5
2 indicator("Putting and getting objects demo")
3
4 // @type A custom type to hold OHLC data.
5 type ChartData
6     float o
7     float h

```

(continues on next page)

(continued from previous page)

```

8   float l
9   float c
10
11 // @function Updates the fields of a `ChartData` object.
12 method update(ChartData this) =>
13     this.o := open
14     this.h := high
15     this.l := low
16     this.c := close
17
18 // @variable A new `ChartData` instance declared on the first bar.
19 var myData = ChartData.new()
20 // @variable A map associating `string` keys with `ChartData` instances.
21 var myMap = map.new<string, ChartData>()
22
23 // Put a new pair with the "A" key into `myMap` only on the first bar.
24 if bar_index == 0
25     myMap.put("A", myData)
26
27 // Put a pair with the "B" key into `myMap` on every bar.
28 myMap.put("B", myData)
29
30 // @variable The `ChartData` value associated with the "A" key in `myMap`.
31 ChartData oldest = myMap.get("A")
32 // @variable The `ChartData` value associated with the "B" key in `myMap`.
33 ChartData newest = myMap.get("B")
34
35 // Update `newest`. Also affects `oldest` and `myData` since they all reference the
36 // same `ChartData` object.
37 newest.update()
38
39 // Plot the fields of `oldest` as candles.
plotcandle(oldest.o, oldest.h, oldest.l, oldest.c)

```

**Note that:**

- This script would behave differently if it passed a copy of `myData` into each `myMap.put()` call. For more information, see [this](#) section of our User Manual's page on [objects](#).

**Inspecting keys and values****`map.keys()` and `map.values()`**

To retrieve all keys and values put into a map, use `map.keys()` and `map.values()`. These functions copy all key/value references within a map `id` to a new `array` object. Modifying the array returned from either of these functions does not affect the `id` map.

Although maps are *unordered* collections, Pine Script™ internally maintains the *insertion order* of a map's key-value pairs. As a result, the `map.keys()` and `map.values()` functions always return `arrays` with their elements ordered based on the `id` map's insertion order.

The script below demonstrates this by displaying the key and value arrays from an `m` map in a `label` once every 50 bars. As we see on the chart, the order of elements in each array returned by `m.keys()` and `m.values()` aligns with the insertion order of the key-value pairs in `m`:

Keys and values demo

Pairs: 3  
Keys: [First, Second, Third]  
Values: [98.0, 99.0, 100.0]

Pairs: 3  
Keys: [First, Second, Third]  
Values: [49.0, 50.0, 51.0]

Pairs: 3  
Keys: [First, Second, Third]  
Values: [34.0, 35.0, 36.0]

77

```

1 // @version=5
2 indicator("Keys and values demo")
3
4 if bar_index % 50 == 0
5     // @variable A map containing pairs of `string` keys and `float` values.
6     m = map.new<string, float>()
7
8     // Put pairs into `m`. The map will maintain this insertion order.
9     m.put("First", math.round(math.random(0, 100)))
10    m.put("Second", m.get("First") + 1)
11    m.put("Third", m.get("Second") + 1)
12
13    // @variable An array containing the keys of `m` in their insertion order.
14    array<string> keys = m.keys()
15    // @variable An array containing the values of `m` in their insertion order.
16    array<float> values = m.values()
17
18    // @variable A label displaying the `size` of `m` and the `keys` and `values` arrays.
19    label debugLabel = label.new(
20        bar_index, 0,
21        str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys, values),
22        color = color.navy, style = label.style_label_center,
23        textcolor = color.white, size = size.huge
24    )

```

### Note that:

- The value with the “First” key is a `random` whole number between 0 and 100. The “Second” value is one greater than the “First”, and the “Third” value is one greater than the “Second”.

It’s important to note a map’s internal insertion order **does not** change when replacing its key-value pairs. The locations of the new elements in the `keys()` and `values()` arrays will be the same as the old elements in such cases. The only exception is if the script completely `removes` the key beforehand.

Below, we’ve added a line of code to `put` a new value with the “Second” key into the `m` map, overwriting the previous value associated with that key. Although the script puts this new pair into the map *after* the one with the “Third” key, the pair’s key and value are still second in the `keys` and `values` arrays since the key was already present in `m` *before* the change:

Keys and values demo

Pairs: 3  
 Keys: [First, Second, Third]  
 Values: [12.0, -2.0, 14.0]

Pairs: 3  
 Keys: [First, Second, Third]  
 Values: [32.0, -2.0, 34.0]

Pairs: 3  
 Keys: [First, Second, Third]  
 Values: [55.0, -2.0, 57.0]

▼

```

1 // @version=5
2 indicator("Keys and values demo")
3
4 if bar_index % 50 == 0
5     // @variable A map containing pairs of `string` keys and `float` values.
6     m = map.new<string, float>()
7
8     // Put pairs into `m`. The map will maintain this insertion order.
9     m.put("First", math.round(math.random(0, 100)))
10    m.put("Second", m.get("First") + 1)
11    m.put("Third", m.get("Second") + 1)
12
13    // Overwrite the "Second" pair in `m`. This will NOT affect the insertion order.
14    // The key and value will still appear second in the `keys` and `values` arrays.
15    m.put("Second", -2)
16
17    // @variable An array containing the keys of `m` in their insertion order.
18    array<string> keys = m.keys()
19    // @variable An array containing the values of `m` in their insertion order.
20    array<float> values = m.values()
21
22    // @variable A label displaying the `size` of `m` and the `keys` and `values` ↵
23    // arrays.
24    label debugLabel = label.new(
25        bar_index, 0,
26        str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys, values),
27        color = color.navy, style = label.style_label_center,
28        textcolor = color.white, size = size.huge
  )
```

**Note:** The elements in a `map.values()` array point to the same values as the map id. Consequently, when the map's values are of *reference types*, including `line`, `linefill`, `box`, `polyline`, `label`, `table`, `chart.point`, or `UDTs`, modifying the instances referenced by the `map.values()` array will also affect those referenced by the map id since the contents of both collections point to identical objects.

### `map.contains()`

To check if a specific key exists within a map id, use `map.contains()`. This function is a convenient alternative to calling `array.includes()` on the array returned from `map.keys()`.

For example, this script checks if various keys exist within an m map, then displays the results in a label:

Inspecting keys demo

Tested keys: [A, B, C, D, E, F]  
Keys found: [A, C, E]

17

```

1 // @version=5
2 indicator("Inspecting keys demo")
3
4 // @variable A map containing `string` keys and `string` values.
5 m = map.new<string, string>()
6
7 // Put key-value pairs into the map.
8 m.put("A", "B")
9 m.put("C", "D")
10 m.put("E", "F")
11
12 // @variable An array of keys to check for in `m`.
13 array<string> testKeys = array.from("A", "B", "C", "D", "E", "F")
14
15 // @variable An array containing all elements from `testKeys` found in the keys of `m`.
16 array<string> mappedKeys = array.new<string>()
17
18 for key in testKeys
19     // Add the `key` to `mappedKeys` if `m` contains it.
20     if m.contains(key)
21         mappedKeys.push(key)
22
23 // @variable A string representing the `testKeys` array and the elements found within
24 // the keys of `m`.
25 string testText = str.format("Tested keys: {0}\nKeys found: {1}", testKeys,
26                             mappedKeys)
27
28 if bar_index == last_bar_index - 1
29     // @variable Displays the `testText` in a label at the `bar_index` before the last.
30     label debugLabel = label.new(
31         bar_index, 0, testText, style = label.style_label_center,
32         textcolor = color.white, size = size.huge
33     )

```

## Removing key-value pairs

To remove a specific key-value pair from a map `id`, use `map.remove()`. This function removes the key and its associated value from the map while preserving the insertion order of other key-value pairs. It returns the removed value if the map *contained* the key. Otherwise, it returns `na`.

To remove all key-value pairs from a map `id` at once, use `map.clear()`.

The following script creates a new `m` map, *puts* key-value pairs into the map, uses `m.remove()` within a loop to remove each valid key listed in the `removeKeys` array, then calls `m.clear()` to remove all remaining key-value pairs. It uses a custom `debugLabel()` method to display the size, keys, and values of `m` after each change:

Removing key-value pairs demo

Added pairs  
Size: 5  
Keys: [A, B, C, D, E]  
Values: [0, 1, 2, 3, 4]

Removed pairs  
Size: 3  
Keys: [A, C, E]  
Values: [0, 2, 4]

Cleared the map  
Size: 0  
Keys: []  
Values: []

```

1 //@version=5
2 indicator("Removing key-value pairs demo")
3
4 //@function Returns a label to display the keys and values from a map.
5 method debugLabel(
6     map<string, int> this, int barIndex = bar_index,
7     color bgColor = color.blue, string note = ""
8 ) =>
9     //@variable A string representing the size, keys, and values in `this` map.
10    string repr = str.format(
11        "{0}\nSize: {1}\nKeys: {2}\nValues: {3}",
12        note, this.size(), str.tostring(this.keys()), str.tostring(this.values())
13    )
14    label.new(
15        barIndex, 0, repr, color = bgColor, style = label.style_label_center,
16        textcolor = color.white, size = size.huge
17    )
18
19 if bar_index == last_bar_index - 1
20     //@variable A map containing `string` keys and `int` values.
21     m = map.new<string, int>()
22
23     // Put key-value pairs into `m`.
24     for [i, key] in array.from("A", "B", "C", "D", "E")
25         m.put(key, i)
26     m.debugLabel(bar_index, color.green, "Added pairs")
27
28     //@variable An array of keys to remove from `m`.
29     array<string> removeKeys = array.from("B", "B", "D", "F", "a")
30
31     // Remove each `key` in `removeKeys` from `m`.
32     for key in removeKeys
33         m.remove(key)

```

(continues on next page)

(continued from previous page)

```

34     m.debugLine(bar_index + 10, color.red, "Removed pairs")
35
36     // Remove all remaining keys from `m`.
37     m.clear()
38     m.debugLine(bar_index + 20, color.purple, "Cleared the map")

```

**Note that:**

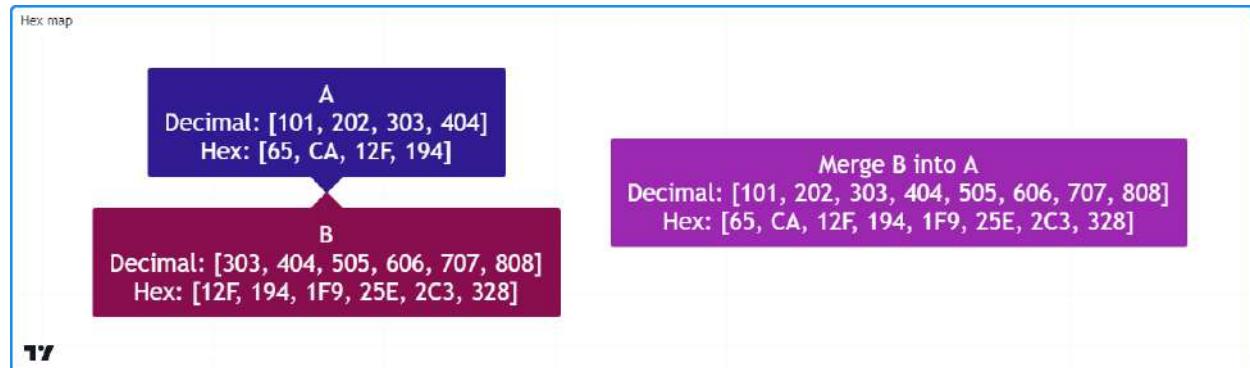
- Not all strings in the `removeKeys` array were present in the keys of `m`. Attempting to remove non-existent keys (“F”, “a”, and the second “B” in this example) has no effect on a map’s contents.

**Combining maps**

Scripts can combine two maps via `map.put_all()`. This function puts *all* key-value pairs from the `id2` map, in their insertion order, into the `id1` map. As with `map.put()`, if any keys in `id2` are also present in `id1`, this function **replaces** the key-value pairs that contain those keys without affecting their initial insertion order.

This example contains a user-defined `hexMap()` function that maps decimal `int` keys to `string` representations of their `hexadecimal` forms. The script uses this function to create two maps, `mapA` and `mapB`, then uses `mapA.put_all(mapB)` to put all key-value pairs from `mapB` into `mapA`.

The script uses a custom `debugLabel()` function to display labels showing the `keys` and `values` of `mapA` and `mapB`, then another label displaying the contents of `mapA` after putting all key-value pairs from `mapB` into it:



```

1  //@version=5
2  indicator("Combining maps demo", "Hex map")
3
4  //@variable An array of string hex digits.
5  var array<string> hexDigits = str.split("0123456789ABCDEF", "")
6
7  //@function Returns a hexadecimal string for the specified `value`.
8  hex(int value) =>
9      //@variable A string representing the hex form of the `value`.
10     string result = ""
11     //@variable A temporary value for digit calculation.
12     int tempValue = value
13     while tempValue > 0
14         //@variable The next integer digit.
15         int digit = tempValue % 16
16         // Add the hex form of the `digit` to the `result`.
17         result := hexDigits.get(digit) + result
18         // Divide the `tempValue` by the base.

```

(continues on next page)

(continued from previous page)

```

19     tempValue := int(tempValue / 16)
20     result
21
22 // @function Returns a map holding the `numbers` as keys and their `hex` strings as_
23 // values.
23 hexMap(array<int> numbers) =>
24     // @variable A map associating `int` keys with `string` values.
25     result = map.new<int, string>()
26     for number in numbers
27         // Put a pair containing the `number` and its `hex()` representation into the_
28 // `result`.
28         result.put(number, hex(number))
29     result
30
31 // @function Returns a label to display the keys and values of a hex map.
32 debugLabel(
33     map<int, string> this, int barIndex = bar_index, color bgColor = color.blue,
34     string style = label.style_label_center, string note = ""
35 ) =>
36     string repr = str.format(
37         "{0}\nDecimal: {1}\nHex: {2}",
38         note, str.tostring(this.keys()), str.tostring(this.values())
39     )
40     label.new(
41         barIndex, 0, repr, color = bgColor, style = style,
42         textcolor = color.white, size = size.huge
43     )
44
45 if bar_index == last_bar_index - 1
46     // @variable A map with decimal `int` keys and hexadecimal `string` values.
47     map<int, string> mapA = hexMap(array.from(101, 202, 303, 404))
48     debugLabel(mapA, bar_index, color.navy, label.style_label_down, "A")
49
50     // @variable A map containing key-value pairs to add to `mapA`.
51     map<int, string> mapB = hexMap(array.from(303, 404, 505, 606, 707, 808))
52     debugLabel(mapB, bar_index, color.maroon, label.style_label_up, "B")
53
54     // Put all pairs from `mapB` into `mapA`.
55     mapA.put_all(mapB)
56     debugLabel(mapA, bar_index + 10, color.purple, note = "Merge B into A")

```

### 3.16.4 Looping through a map

There are several ways scripts can iteratively access the keys and values in a map. For example, one could loop through a map's `keys()` array and `get()` the value for each key, like so:

```
for key in thisMap.keys()
    value = thisMap.get(key)
```

However, we recommend using a `for...in` loop directly on a map, as it iterates over the map's key-value pairs in their insertion order, returning a tuple containing the next pair's key and value on each iteration.

For example, this line of code loops through each key and `value` in `thisMap`, starting from the first key-value pair put into it:

```
for [key, value] in thisMap
```

Let's use this structure to write a script that displays a map's key-value pairs in a [table](#). In the example below, we've defined a custom `toTable()` method that creates a [table](#), then uses a `for...in` loop to iterate over the map's key-value pairs and populate the table's cells. The script uses this method to visualize a map containing length-bar averages of price and volume data:



```

1 // @version=5
2 indicator("Looping through a map demo", "Table of averages")
3
4 //@variable The length of the moving average.
5 int length = input.int(20, "Length")
6 //@variable The size of the table text.
7 string txtSize = input.string(
8     size.huge, "Text size",
9     options = [size.auto, size.tiny, size.small, size.normal, size.large, size.huge]
10 )
11
12 //@function Displays the pairs of `this` map within a table.
13 //@param this A map with `string` keys and `float` values.
14 //@param position The position of the table on the chart.
15 //@param header The string to display on the top row of the table.
16 //@param textSize The size of the text in the table.
17 //@returns A new `table` object with cells displaying each pair in `this`.
18 method toTable()
19     map<string, float> this, string position = position.middle_center, string header_
20     = na,
21     string textSize = size.huge
22 ) =>
23     // Color variables
24     borderColor = #000000
25     headerColor = color.rgb(1, 88, 80)
26     pairColor = color.maroon
27     textColor = color.white

```

(continues on next page)

(continued from previous page)

```

28     // @variable A table that displays the key-value pairs of `this` map.
29     table result = table.new(
30         position, this.size() + 1, 3, border_width = 2, border_color = borderColor
31     )
32     // Initialize top and side header cells.
33     result.cell(1, 0, header, bgcolor = headerColor, text_color = textColor, text_
34     ↪size = textSize)
35     result.merge_cells(1, 0, this.size(), 0)
36     result.cell(0, 1, "Key", bgcolor = headerColor, text_color = textColor, text_size_
37     ↪= textSize)
38     result.cell(0, 2, "Value", bgcolor = headerColor, text_color = textColor, text_
39     ↪size = textSize)

40
41     // @variable The column index of the table. Updates on each loop iteration.
42     int col = 1

43
44     // Loop over each `key` and `value` from `this` map in the insertion order.
45     for [key, value] in this
46         // Initialize a `key` cell in the `result` table on row 1.
47         result.cell(
48             col, 1, str.tostring(key), bgcolor = color.maroon,
49             text_color = color.white, text_size = textSize
50         )
51         // Initialize a `value` cell in the `result` table on row 2.
52         result.cell(
53             col, 2, str.tostring(value), bgcolor = color.maroon,
54             text_color = color.white, text_size = textSize
55         )
56         // Move to the next column index.
57         col += 1
58     result // Return the `result` table.

59
60     // @variable A map with `string` keys and `float` values to hold `length`-bar averages.
61     averages = map.new<string, float>()

62
63     // Put key-value pairs into the `averages` map.
64     averages.put("Open", ta.sma(open, length))
65     averages.put("High", ta.sma(high, length))
66     averages.put("Low", ta.sma(low, length))
67     averages.put("Close", ta.sma(close, length))
68     averages.put("Volume", ta.sma(volume, length))

69
70     // @variable The text to display at the top of the table.
71     string headerText = str.format("{0} {1}-bar averages", "" + syminfo.tickerid + "",_
72     ↪length)
73     // Display the `averages` map in a `table` with the `headerText`.
74     averages.ToTable(header = headerText, textSize = txtSize)

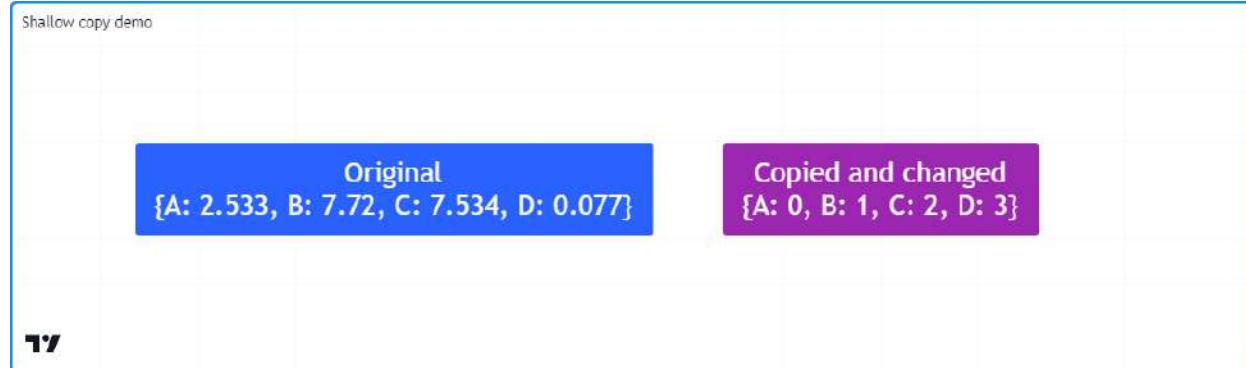
```

### 3.16.5 Copying a map

#### Shallow copies

Scripts can make a *shallow copy* of a map `id` using the `map.copy()` function. Modifications to a shallow copy do not affect the original `id` map or its internal insertion order.

For example, this script constructs an `m` map with the keys “A”, “B”, “C”, and “D” assigned to four `random` values between 0 and 10. It then creates an `mCopy` map as a shallow copy of `m` and updates the values associated with its keys. The script displays the key-value pairs in `m` and `mCopy` on the chart using our custom `debugLabel()` method:



```

1 // @version=5
2 indicator("Shallow copy demo")
3
4 // @function Displays the key-value pairs of `this` map in a label.
5 method debugLabel()
6     map<string, float> this, int barIndex = bar_index, color bgColor = color.blue,
7     color textColor = color.white, string note = ""
8 ) =>
9     // @variable The text to display in the label.
10    labelText = note + "\n{"
11    for [key, value] in this
12        labelText += str.format("{0}: {1}, ", key, value)
13    labelText := str.replace(labelText, ", ", "}", this.size() - 1)
14
15    if barstate.ishistory
16        label result = label.new(
17            barIndex, 0, labelText, color = bgColor, style = label.style_label_
18            ←center,
19            textColor = textColor, size = size.huge
20        )
21
22    if bar_index == last_bar_index - 1
23        // @variable A map of `string` keys and random `float` values.
24        m = map.new<string, float>()
25
26        // Assign random values to an array of keys in `m`.
27        for key in array.from("A", "B", "C", "D")
28            m.put(key, math.random(0, 10))
29
30        // @variable A shallow copy of `m`.
31        mCopy = m.copy()
32
33        // Assign the insertion order value `i` to each `key` in `mCopy`.

```

(continues on next page)

(continued from previous page)

```

33     for [i, key] in mCopy.keys()
34         mCopy.put(key, i)
35
36     // Display the labels.
37     m.debugLabel(bar_index, note = "Original")
38     mCopy.debugLabel(bar_index + 10, color.purple, note = "Copied and changed")

```

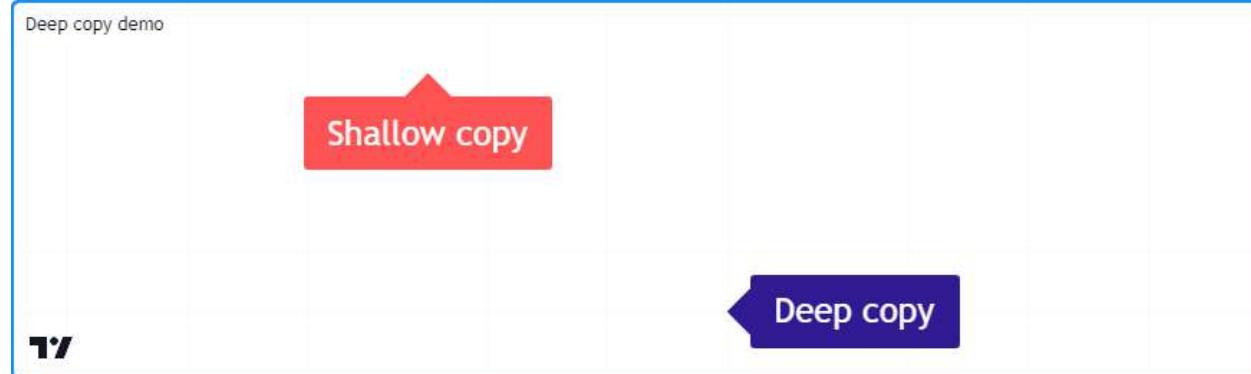
## Deep copies

While a *shallow copy* will suffice when copying maps that have values of a *fundamental type*, it's important to remember that shallow copies of a map holding values of a *reference type* (`line`, `linefill`, `box`, `polyline`, `label`, `table`, `chart.point` or a `UDT`) point to the same objects as the original. Modifying the objects referenced by a shallow copy will affect the instances referenced by the original map and vice versa.

To ensure changes to objects referenced by a copied map do not affect instances referenced in other locations, one can make a *deep copy* by creating a new map with key-value pairs containing copies of each value in the original map.

This example creates an `original` map of `string` keys and `label` values and `puts` a key-value pair into it. The script copies the map to a `shallow` variable via the built-in `copy()` method, then to a `deep` variable using a custom `deepCopy()` method.

As we see from the chart, changes to the label retrieved from the `shallow` copy also affect the instance referenced by the `original` map, but changes to the one from the `deep` copy do not:



```

1  //@version=5
2  indicator("Deep copy demo")
3
4  //@function Returns a deep copy of `this` map.
5  method map<string, label> DeepCopy(map<string, label> this) =>
6      // @variable A deep copy of `this` map.
7      result = map.new<string, label>()
8      // Add key-value pairs with copies of each `value` to the `result`.
9      for [key, value] in this
10         result.put(key, value.copy())
11     result //Return the `result`.
12
13  //@variable A map containing `string` keys and `label` values.
14  var original = map.new<string, label>()
15
16  if bar_index == last_bar_index - 1
17      // Put a new key-value pair into the `original` map.
18      map.put(

```

(continues on next page)

(continued from previous page)

```

19         original, "Test",
20         label.new(bar_index, 0, "Original", textcolor = color.white, size = size.
21     ↪huge)
22     )
23
24     // @variable A shallow copy of the `original` map.
25     map<string, label> shallow = original.copy()
26     // @variable A deep copy of the `original` map.
27     map<string, label> deep = original.deepCopy()
28
29     // @variable The "Test" label from the `shallow` copy.
30     label shallowLabel = shallow.get("Test")
31     // @variable The "Test" label from the `deep` copy.
32     label deepLabel = deep.get("Test")
33
34     // Modify the "Test" label's `y` attribute in the `original` map.
35     // This also affects the `shallowLabel`.
36     original.get("Test").set_y(label.all.size())
37
38     // Modify the `shallowLabel`. Also modifies the "Test" label in the `original` ↪
39     ↪map.
40     shallowLabel.set_text("Shallow copy")
41     shallowLabel.set_color(color.red)
42     shallowLabel.set_style(label.style_label_up)
43
44     // Modify the `deepLabel`. Does not modify any other label instance.
45     deepLabel.set_text("Deep copy")
46     deepLabel.set_color(color.navy)
47     deepLabel.set_style(label.style_label_left)
48     deepLabel.set_x(bar_index + 5)

```

**Note that:**

- The `deepCopy()` method loops through the `original` map, copying each value and *putting* key-value pairs containing the copies into a `new` map instance.

### 3.16.6 Scope and history

As with other collections in Pine, map variables leave historical trails on each bar, allowing a script to access past map instances assigned to a variable using the history-referencing operator `[]`. Scripts can also assign maps to global variables and interact with them from the scopes of *functions*, *methods*, and *conditional structures*.

As an example, this script uses a global map and its history to calculate an aggregate set of `EMAs`. It declares a global `globalData` map of `int` keys and `float` values, where each key in the map corresponds to the length of each EMA calculation. The user-defined `update()` function calculates each key-length EMA by mixing the values from the previous map assigned to `globalData` with the current `source` value.

The script plots the `maximum` and `minimum` values in the global map's `values()` array and the value from `globalData.get(50)` (i.e., the 50-bar EMA):



```

1 // @version=5
2 indicator("Scope and history demo", overlay = true)
3
4 // @variable The source value for EMA calculation.
5 float source = input.source(close, "Source")
6
7 // @variable A map containing global key-value pairs.
8 globalData = map.new<int, float>()
9
10 // @function Calculates a set of EMAs and updates the key-value pairs in `globalData`.
11 update() =>
12     // @variable The previous map instance assigned to `globalData`.
13     map<int, float> previous = globalData[1]
14
15     // Put key-value pairs with keys 10-200 into `globalData` if `previous` is `na`.
16     if na(previous)
17         for i = 10 to 200
18             globalData.put(i, source)
19     else
20         // Iterate each `key` and `value` in the `previous` map.
21         for [key, value] in previous
22             // @variable The smoothing parameter for the `key`-length EMA.
23             float alpha = 2.0 / (key + 1.0)
24             // @variable The `key`-length EMA value.
25             float ema = (1.0 - alpha) * value + alpha * source
26             // Put the `key`-length `ema` into the `globalData` map.
27             globalData.put(key, ema)
28
29 // Update the `globalData` map.
30 update()
31
32 // @variable The array of values from `globalData` in their insertion order.
33 array<float> values = globalData.values()
34
35 // Plot the max EMA, min EMA, and 50-bar EMA values.
36 plot(values.max(), "Max EMA", color.green, 2)
37 plot(values.min(), "Min EMA", color.red, 2)
38 plot(globalData.get(50), "50-bar EMA", color.orange, 3)

```

### 3.16.7 Maps of other collections

Maps cannot directly use other maps, *arrays*, or *matrices* as values, but they can hold values of a *user-defined type* that contains collections within its fields.

For example, suppose we want to create a “2D” map that uses *string* keys to access *nested maps* that hold pairs of *string* keys and *float* values. Since maps cannot use other collections as values, we will first create a *wrapper type* with a field to hold a *map<string, float>* instance, like so:

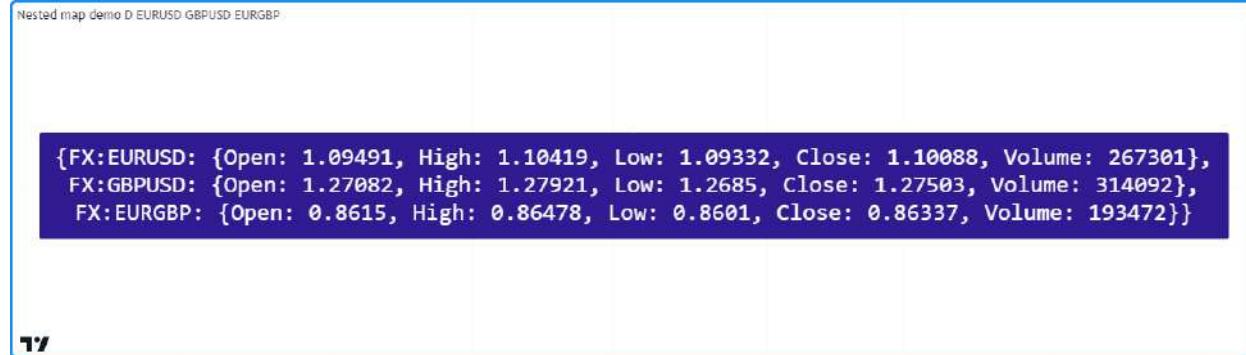
```
//@type A wrapper type for maps with `string` keys and `float` values.
type Wrapper
    map<string, float> data
```

With our *Wrapper* type defined, we can create maps of *string* keys and *Wrapper* values, where the *data* field of each value in the map points to a *map<string, float>* instance:

```
mapOfMaps = map.new<string, Wrapper>()
```

The script below uses this concept to construct a map containing maps that hold OHLCV data requested from multiple tickers. The user-defined *requestData()* function requests price and volume data from a ticker, creates a *<string, float>* map, *puts* the data into it, then returns a *Wrapper* instance containing the new map.

The script *puts* the results from each call to *requestData()* into the *mapOfMaps*, then creates a *string* representation of the nested maps with a user-defined *toString()* method, which it displays on the chart in a *label*:



```
1 //@version=5
2 indicator("Nested map demo")
3
4 //@variable The timeframe of the requested data.
5 string tf = input.timeframe("D", "Timeframe")
6 // Symbol inputs.
7 string symbol1 = input.symbol("EURUSD", "Symbol 1")
8 string symbol2 = input.symbol("GBPUSD", "Symbol 2")
9 string symbol3 = input.symbol("EURGBP", "Symbol 3")
10
11 //@type A wrapper type for maps with `string` keys and `float` values.
12 type Wrapper
13     map<string, float> data
14
15 //@function Returns a wrapped map containing OHLCV data from the `tickerID` at the
16 // `timeframe`.
17 requestData(string tickerID, string timeframe) =>
18     // Request a tuple of OHLCV values from the specified ticker and timeframe.
19     [o, h, l, c, v] = request.security(
        tickerID, timeframe,
```

(continues on next page)

(continued from previous page)

```

20     [open, high, low, close, volume]
21   )
22   //@variable A map containing requested OHLCV data.
23   result = map.new<string, float>()
24   // Put key-value pairs into the `result`.
25   result.put("Open", o)
26   result.put("High", h)
27   result.put("Low", l)
28   result.put("Close", c)
29   result.put("Volume", v)
30   //Return the wrapped `result`.
31   Wrapper.new(result)
32
33 // @function Returns a string representing `this` map of `string` keys and `Wrapper` ↴
34 // values.
35 method toString(map<string, Wrapper> this) =>
36   // @variable A string representation of `this` map.
37   string result = "{"
38
39   // Iterate over each `key1` and associated `wrapper` in `this`.
40   for [key1, wrapper] in this
41     // Add `key1` to the `result`.
42     result += key1
43
44     // @variable A string representation of the `wrapper.data` map.
45     string innerStr = ": {"
46     // Iterate over each `key2` and associated `value` in the wrapped map.
47     for [key2, value] in wrapper.data
48       // Add the key-value pair's representation to `innerStr`.
49       innerStr += str.format("{0}: {1}, ", key2, str.tostring(value))
50
51     // Replace the end of `innerStr` with ")" and add to `result`.
52     result += str.replace(innerStr, ", ", "},\n", wrapper.data.size() - 1)
53
54   // Replace the blank line at the end of `result` with ")".
55   result := str.replace(result, ",\n", "}", this.size() - 1)
56   result
57
58 // @variable A map of wrapped maps containing OHLCV data from multiple tickers.
59 var mapOfMaps = map.new<string, Wrapper>()
60
61 // @variable A label showing the contents of the `mapOfMaps`.
62 var debugLabel = label.new(
63   bar_index, 0, color = color.navy, textcolor = color.white, size = size.huge,
64   style = label.style_label_center, text_font_family = font.family_monospace
65 )
66
67 // Put wrapped maps into `mapOfMaps`.
68 mapOfMaps.put(symbol1, requestData(symbol1, tf))
69 mapOfMaps.put(symbol2, requestData(symbol2, tf))
70 mapOfMaps.put(symbol3, requestData(symbol3, tf))
71
72 // Update the label.
73 debugLabel.set_text(mapOfMaps.toString())
74 debugLabel.set_x(bar_index)

```



## CONCEPTS



### 4.1 Alerts

- *Introduction*
- *Script alerts*
- ``alertcondition()` events`
- *Avoiding repainting with alerts*

#### 4.1.1 Introduction

TradingView alerts run 24x7 on our servers and do not require users to be logged in to execute. Alerts are created from the charts user interface (*UI*). You will find all the information necessary to understand how alerts work and how to create them from the charts UI in the Help Center's [About TradingView alerts](#) page.

Some of the alert types available on TradingView (*generic alerts*, *drawing alerts* and *script alerts* on order fill events) are created from symbols or scripts loaded on the chart and do not require specific coding. Any user can create these types of alerts from the charts UI.

Other types of alerts (*script alerts* triggering on *alert()* *function calls*, and *alertcondition()* *alerts*) require specific Pine Script™ code to be present in a script to create an *alert event* before script users can create alerts from them using the charts UI. Additionally, while script users can create *script alerts* triggering on *order fill events* from the charts UI on any strategy loaded on their chart, Programmers can specify explicit order fill alert messages in their script for each type of order filled by the broker emulator.

This page covers the different ways Pine Script™ programmers can code their scripts to create alert events from which script users will in turn be able to create alerts from the charts UI. We will cover:

- How to use the `alert()` function to *alert() function calls* in indicators or strategies, which can then be included in *script alerts* created from the charts UI.
- How to add custom alert messages to be included in *script alerts* triggering on the *order fill events* of strategies.

- How to use the `alertcondition()` function to generate, in indicators only, *alertcondition()* events which can then be used to create *alertcondition()* alerts from the charts UI.

Keep in mind that:

- No alert-related Pine Script™ code can create a running alert in the charts UI; it merely creates alert events which can then be used by script users to create running alerts from the charts UI.
- Alerts only trigger in the realtime bar. The operational scope of Pine Script™ code dealing with any type of alert is therefore restricted to realtime bars only.
- When an alert is created in the charts UI, TradingView saves a mirror image of the script and its inputs, along with the chart's main symbol and timeframe to run the alert on its servers. Subsequent changes to your script's inputs or the chart will thus not affect running alerts previously created from them. If you want any changes to your context to be reflected in a running alert's behavior, you will need to delete the alert and create a new one in the new context.

### Background

The different methods Pine programmers can use today to create alert events in their script are the result of successive enhancements deployed throughout Pine Script™'s evolution. The `alertcondition()` function, which works in indicators only, was the first feature allowing Pine Script™ programmers to create alert events. Then came order fill alerts for strategies, which trigger when the broker emulator creates *order fill events*. *Order fill events* require no special code for script users to create alerts on them, but by way of the `alert_message` parameter for order-generating strategy.\*() functions, programmers can customize the message of alerts triggering on *order fill events* by defining a distinct alert message for any number of order fulfillment events.

The `alert()` function is the most recent addition to Pine Script™. It more or less supersedes `alertcondition()`, and when used in strategies, provides a useful complement to alerts on *order fill events*.

### Which type of alert is best?

For Pine Script™ programmers, the `alert()` function will generally be easier and more flexible to work with. Contrary to `alertcondition()`, it allows for dynamic alert messages, works in both indicators and strategies and the programmer decides on the frequency of `alert()` events.

While `alert()` calls can be generated on any logic programmable in Pine, including when orders are **sent** to the broker emulator in strategies, they cannot be coded to trigger when orders are **executed** (or **filled**) because after orders are sent to the broker emulator, the emulator controls their execution and does not report fill events back to the script directly.

When a script user wants to generate an alert on a strategy's order fill events, he must include those events when creating a *script alert* on the strategy in the “Create Alert” dialog box. No special code is required in scripts for users to be able to do this. The message sent with order fill events can, however, be customized by programmers through use of the `alert_message` parameter in order-generating strategy.\*() function calls. A combination of `alert()` calls and the use of custom `alert_message` arguments in order-generating strategy.\*() calls should allow programmers to generate alert events on most conditions occurring in their script's execution.

The `alertcondition()` function remains in Pine Script™ for backward compatibility, but it can also be used advantageously to generate distinct alerts available for selection as individual items in the “Create Alert” dialog box's “Condition” field.

## 4.1.2 Script alerts

When a script user creates a *script alert* using the “Create Alert” dialog box, the events able to trigger the alert will vary depending on whether the alert is created from an indicator or a strategy.

A *script alert* created from an **indicator** will trigger when:

- The indicator contains `alert()` calls.
- The code’s logic allows a specific `alert()` call to execute.
- The frequency specified in the `alert()` call allows the alert to trigger.

A *script alert* created from a **strategy** can trigger on *alert() function calls*, on *order fill events*, or both. The script user creating an alert on a strategy decides which type of events he wishes to include in his *script alert*. While users can create a *script alert* on *order fill events* without the need for a strategy to include special code, it must contain `alert()` calls for users to include *alert() function calls* in their *script alert*.

### `alert()` function events

The `alert()` function has the following signature:

```
alert (message, freq)
```

#### **message**

A “series string” representing the message text sent when the alert triggers. Because this argument allows “series” values, it can be generated at runtime and differ bar to bar, making it dynamic.

#### **freq**

An “input string” specifying the triggering frequency of the alert. Valid arguments are:

- `alert.freq_once_per_bar`: Only the first call per realtime bar triggers the alert (default value).
- `alert.freq_once_per_bar_close`: An alert is only triggered when the realtime bar closes and an `alert()` call is executed during that script iteration.
- `alert.freq_all`: All calls during the realtime bar trigger the alert.

The `alert()` function can be used in both indicators and strategies. For an `alert()` call to trigger a *script alert* configured on *alert() function calls*, the script’s logic must allow the `alert()` call to execute, **and** the frequency determined by the `freq` parameter must allow the alert to trigger.

Note that by default, strategies are recalculated at the bar’s close, so if the `alert()` function with the frequency `alert.freq_all` or `alert.freq_once_per_bar` is used in a strategy, then it will be called no more often than once at the bar’s close. In order to enable the `alert()` function to be called during the bar construction process, you need to enable the `calc_on_every_tick` option.

### Using all `alert()` calls

Let’s look at an example where we detect crosses of the RSI centerline:

```
1 // @version=5
2 indicator("All `alert()` calls")
3 r = ta.rsi(close, 20)
4
5 // Detect crosses.
6 xUp = ta.crossover(r, 50)
7 xDn = ta.crossunder(r, 50)
```

(continues on next page)

(continued from previous page)

```

8 // Trigger an alert on crosses.
9 if xUp
10   alert("Go long (RSI is " + str.tostring(r, "#.00)"))
11 else if xDn
12   alert("Go short (RSI is " + str.tostring(r, "#.00)"))
13
14 plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
15 plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
16 hline(50)
17 plot(r)

```

If a *script alert* is created from this script:

- When RSI crosses the centerline up, the *script alert* will trigger with the “Go long...” message. When RSI crosses the centerline down, the *script alert* will trigger with the “Go short...” message.
- Because no argument is specified for the `freq` parameter in the `alert()` call, the default value of `alert.freq_once_per_bar` will be used, so the alert will only trigger the first time each of the `alert()` calls is executed during the realtime bar.
- The message sent with the alert is composed of two parts: a constant string and then the result of the `str.tostring()` call which will include the value of RSI at the moment where the `alert()` call is executed by the script. An alert message for a cross up would look like: “Go long (RSI is 53.41)”.
- Because a *script alert* always triggers on any occurrence of a call to `alert()`, as long as the frequency used in the call allows for it, this particular script does not allow a script user to restrict his *script alert* to longs only, for example.

Note that:

- Contrary to an `alertcondition()` call which is always placed at column 0 (in the script’s global scope), the `alert()` call is placed in the local scope of an `if` branch so it only executes when our triggering condition is met. If an `alert()` call was placed in the script’s global scope at column 0, it would execute on all bars, which would likely not be the desired behavior.
- An `alertcondition()` could not accept the same string we use for our alert’s message because of its use of the `str.tostring()` call. `alertcondition()` messages must be constant strings.

Lastly, because `alert()` messages can be constructed dynamically at runtime, we could have used the following code to generate our alert events:

```

// Trigger an alert on crosses.
if xUp or xDn
  firstPart = (xUp ? "Go long" : "Go short") + " (RSI is "
  alert(firstPart + str.tostring(r, "#.00")))

```

## Using selective `alert()` calls

When users create a *script alert* on `alert() function calls`, the alert will trigger on any call the script makes to the `alert()` function, provided its frequency constraints are met. If you want to allow your script’s users to select which `alert()` function call in your script will trigger a *script alert*, you will need to provide them with the means to indicate their preference in your script’s inputs, and code the appropriate logic in your script. This way, script users will be able to create multiple *script alerts* from a single script, each behaving differently as per the choices made in the script’s inputs prior to creating the alert in the charts UI.

Suppose, for our next example, that we want to provide the option of triggering alerts on only longs, only shorts, or both. You could code your script like this:

```

1 //@version=5
2 indicator("Selective `alert()` calls")
3 detectLongsInput = input.bool(true, "Detect Longs")
4 detectShortsInput = input.bool(true, "Detect Shorts")
5 repaintInput = input.bool(false, "Allow Repainting")
6
7 r = ta.rsi(close, 20)
8 // Detect crosses.
9 xUp = ta.crossover(r, 50)
10 xDn = ta.crossunder(r, 50)
11 // Only generate entries when the trade's direction is allowed in inputs.
12 enterLong = detectLongsInput and xUp and (repaintInput or barstate.isconfirmed)
13 enterShort = detectShortsInput and xDn and (repaintInput or barstate.isconfirmed)
14 // Trigger the alerts only when the compound condition is met.
15 if enterLong
16     alert("Go long (RSI is " + str.tostring(r, "#.00") + ")")
17 else if enterShort
18     alert("Go short (RSI is " + str.tostring(r, "#.00") + ")")
19
20 plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
21 plotchar(enterShort, "Go Short", "▼", location.top, color.red, size = size.tiny)
22 hline(50)
23 plot(r)

```

Note how:

- We create a compound condition that is met only when the user's selection allows for an entry in that direction. A long entry on a crossover of the centerline only triggers the alert when long entries have been enabled in the script's Inputs.
- We offer the user to indicate his repainting preference. When he does not allow the calculations to repaint, we wait until the bar's confirmation to trigger the compound condition. This way, the alert and the marker only appear at the end of the realtime bar.
- If a user of this script wanted to create two distinct script alerts from this script, i.e., one triggering only on longs, and one only on shorts, then he would need to:
  - Select only “Detect Longs” in the inputs and create a first *script alert* on the script.
  - Select only “Detect Shorts” in the Inputs and create another *script alert* on the script.

## In strategies

`alert()` function calls can be used in strategies also, with the provision that strategies, by default, only execute on the `close` of realtime bars. Unless `calc_on_every_tick = true` is used in the `strategy()` declaration statement, all `alert()` calls will use the `alert.freq_once_per_bar_close` frequency, regardless of the argument used for `freq`.

While *script alerts* on strategies will use *order fill events* to trigger alerts when the broker emulator fills orders, `alert()` can be used advantageously to generate other alert events in strategies.

This strategy creates *alert() function calls* when RSI moves against the trade for three consecutive bars:

```

1 //@version=5
2 strategy("Strategy with selective `alert()` calls")
3 r = ta.rsi(close, 20)
4
5 // Detect crosses.
6 xUp = ta.crossover(r, 50)

```

(continues on next page)

(continued from previous page)

```

7 xDn = ta.crossunder(r, 50)
8 // Place orders on crosses.
9 if xUp
10     strategy.entry("Long", strategy.long)
11 else if xDn
12     strategy.entry("Short", strategy.short)
13
14 // Trigger an alert when RSI diverges from our trade's direction.
15 divInLongTrade = strategy.position_size > 0 and ta.falling(r, 3)
16 divInShortTrade = strategy.position_size < 0 and ta.rising(r, 3)
17 if divInLongTrade
18     alert("WARNING: Falling RSI", alert.freq_once_per_bar_close)
19 if divInShortTrade
20     alert("WARNING: Rising RSI", alert.freq_once_per_bar_close)
21
22 plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
23 plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
24 plotchar(divInLongTrade, "WARNING: Falling RSI", "•", location.top, color.red, _ 
25     size = size.tiny)
26 plotchar(divInShortTrade, "WARNING: Rising RSI", "•", location.bottom, color.lime, _ 
27     size = size.tiny)
hline(50)
plot(r)

```

If a user created a *script alert* from this strategy and included both *order fill events* and *alert() function calls* in his alert, the alert would trigger whenever an order is executed, or when one of the `alert()` calls was executed by the script on the realtime bar's closing iteration, i.e., when `barstate.isrealtime` and `barstate.isconfirmed` are both true. The *alert() function events* in the script would only trigger the alert when the realtime bar closes because `alert.freq_once_per_bar_close` is the argument used for the `freq` parameter in the `alert()` calls.

## Order fill events

When a *script alert* is created from an indicator, it can only trigger on *alert() function calls*. However, when a *script alert* is created from a strategy, the user can specify that *order fill events* also trigger the *script alert*. An *order fill event* is any event generated by the broker emulator which causes a simulated order to be executed. It is the equivalent of a trade order being filled by a broker/exchange. Orders are not necessarily executed when they are placed. In a strategy, the execution of orders can only be detected indirectly and after the fact, by analyzing changes in built-in variables such as `strategy.opentrades` or `strategy.position_size`. *Script alerts* configured on *order fill events* are thus useful in that they allow the triggering of alerts at the precise moment of an order's execution, before a script's logic can detect it.

Pine Script™ programmers can customize the alert message sent when specific orders are executed. While this is not a pre-requisite for *order fill events* to trigger, custom alert messages can be useful because they allow custom syntax to be included with alerts in order to route actual orders to a third-party execution engine, for example. Specifying custom alert messages for specific *order fill events* is done by means of the `alert_message` parameter in functions which can generate orders: `strategy.close()`, `strategy.entry()`, `strategy.exit()` and `strategy.order()`.

The argument used for the `alert_message` parameter is a “series string”, so it can be constructed dynamically using any variable available to the script, as long as it is converted to string format.

Let's look at a strategy where we use the `alert_message` parameter in both our `strategy.entry()` calls:

```

1 //@version=5
2 strategy("Strategy using `alert_message`")
3 r = ta.rsi(close, 20)
4

```

(continues on next page)

(continued from previous page)

```

5 // Detect crosses.
6 xUp = ta.crossover( r, 50)
7 xDn = ta.crossunder(r, 50)
8 // Place order on crosses using a custom alert message for each.
9 if xUp
10     strategy.entry("Long", strategy.long, stop = high, alert_message = "Stop-buy_"
11     ↪executed (stop was " + str.tostring(high) + ")")
12 else if xDn
13     strategy.entry("Short", strategy.short, stop = low, alert_message = "Stop-sell_"
14     ↪executed (stop was " + str.tostring(low) + ")")
15
16 plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
17 plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
18 hline(50)
19 plot(r)

```

Note that:

- We use the `stop` parameter in our `strategy.entry()` calls, which creates stop-buy and stop-sell orders. This entails that buy orders will only execute once price is higher than the `high` on the bar where the order is placed, and sell orders will only execute once price is lower than the `low` on the bar where the order is placed.
- The up/down arrows which we plot with `plotchar()` are plotted when orders are **placed**. Any number of bars may elapse before the order is actually executed, and in some cases the order will never be executed because price does not meet the required condition.
- Because we use the same `id` argument for all buy orders, any new buy order placed before a previous order's condition is met will replace that order. The same applies to sell orders.
- Variables included in the `alert_message` argument are evaluated when the order is executed, so when the alert triggers.

When the `alert_message` parameter is used in a strategy's order-generating `strategy.*()` function calls, script users must include the `{}{{strategy.order.alert_message}}` placeholder in the “Create Alert” dialog box’s “Message” field when creating *script alerts on order fill events*. This is required so the `alert_message` argument used in the order-generating `strategy.*()` function calls is used in the message of alerts triggering on each *order fill event*. When only using the `{}{{strategy.order.alert_message}}` placeholder in the “Message” field and the `alert_message` parameter is present in only some of the order-generating `strategy.*()` function calls in your strategy, an empty string will replace the placeholder in the message of alerts triggered by any order-generating `strategy.*()` function call not using the `alert_message` parameter.

While other placeholders can be used in the “Create Alert” dialog box’s “Message” field by users creating alerts on *order fill events*, they cannot be used in the argument of `alert_message`.

### 4.1.3 `alertcondition()` events

The `alertcondition()` function allows programmers to create individual *alertcondition events* in their indicators. One indicator may contain more than one `alertcondition()` call. Each call to `alertcondition()` in a script will create a corresponding alert selectable in the “Condition” dropdown menu of the “Create Alert” dialog box.

While the presence of `alertcondition()` calls in a `strategy` script will not cause a compilation error, alerts cannot be created from them.

The `alertcondition()` function has the following signature:

```
alertcondition(condition, title, message)
```

#### condition

A “series bool” value (`true` or `false`) which determines when the alert will trigger. It is a required argument. When the value is `true` the alert will trigger. When the value is `false` the alert will not trigger. Contrary to `alert()` function calls, `alertcondition()` calls must start at column zero of a line, so cannot be placed in conditional blocks.

#### title

A “const string” optional argument that sets the name of the alert condition as it will appear in the “Create Alert” dialog box’s “Condition” field in the charts UI. If no argument is supplied, “Alert” will be used.

#### message

A “const string” optional argument that specifies the text message to display when the alert triggers. The text will appear in the “Message” field of the “Create Alert” dialog box, from where script users can then modify it when creating an alert. **As this argument must be a “const string”, it must be known at compilation time and thus cannot vary bar to bar.** It can, however, contain placeholders which will be replaced at runtime by dynamic values that may change bar to bar. See this page’s *Placeholders* section for a list.

The `alertcondition()` function does not include a `freq` parameter. The frequency of `alertcondition()` alerts is determined by users in the “Create Alert” dialog box.

## Using one condition

Here is an example of code creating `alertcondition()` events:

```

1 //@version=5
2 indicator(``alertcondition()` on single condition")
3 r = ta.rsi(close, 20)
4
5 xUp = ta.crossover(r, 50)
6 xDn = ta.crossunder(r, 50)
7
8 plot(r, "RSI")
9 hline(50)
10 plotchar(xUp, "Long", "▲", location.bottom, color.lime, size = size.tiny)
11 plotchar(xDn, "Short", "▼", location.top, color.red, size = size.tiny)
12
13 alertcondition(xUp, "Long Alert", "Go long")
14 alertcondition(xDn, "Short Alert", "Go short ")

```

Because we have two `alertcondition()` calls in our script, two different alerts will be available in the “Create Alert” dialog box’s “Condition” field: “Long Alert” and “Short Alert”.

If we wanted to include the value of RSI when the cross occurs, we could not simply add its value to the message string using `str.tostring(r)`, as we could in an `alert()` call or in an `alert_message` argument in a strategy. We can, however, include it using a placeholder. This shows two alternatives:

```

alertcondition(xUp, "Long Alert", "Go long. RSI is {{plot_0}}")
alertcondition(xDn, "Short Alert", 'Go short. RSI is {{plot("RSI")}}')

```

Note that:

- The first line uses the `{{plot_0}}` placeholder, where the plot number corresponds to the order of the plot in the script.
- The second line uses the `{{plot("[plot_title]")}}` type of placeholder, which must include the `title` of the `plot()` call used in our script to plot RSI. Double quotes are used to wrap the plot’s title inside the `{{plot("RSI")}}` placeholder. This requires that we use single quotes to wrap the message string.

- Using one of these methods, we can include any numeric value that is plotted by our indicator, but as strings cannot be plotted, no string variable can be used.

## Using compound conditions

If we want to offer script users the possibility of creating a single alert from an indicator using multiple `alertcondition()` calls, we will need to provide options in the script's inputs through which users will indicate the conditions they want to trigger their alert before creating it.

This script demonstrates one way to do it:

```

1 //@version=5
2 indicator(`alertcondition()` on multiple conditions")
3 detectLongsInput = input.bool(true, "Detect Longs")
4 detectShortsInput = input.bool(true, "Detect Shorts")
5
6 r = ta.rsi(close, 20)
7 // Detect crosses.
8 xUp = ta.crossover(r, 50)
9 xDn = ta.crossunder(r, 50)
10 // Only generate entries when the trade's direction is allowed in inputs.
11 enterLong = detectLongsInput and xUp
12 enterShort = detectShortsInput and xDn
13
14 plot(r)
15 plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
16 plotchar(enterShort, "Go Short", "▼", location.top, color.red, size = size.tiny)
17 hline(50)
18 // Trigger the alert when one of the conditions is met.
19 alertcondition(enterLong or enterShort, "Compound alert", "Entry")

```

Note how the `alertcondition()` call is allowed to trigger on one of two conditions. Each condition can only trigger the alert if the user enables it in the script's inputs before creating the alert.

## Placeholders

These placeholders can be used in the `message` argument of `alertcondition()` calls. They will be replaced with dynamic values when the alert triggers. They are the only way to include dynamic values (values that can vary bar to bar) in `alertcondition()` messages.

Note that users creating `alertcondition()` alerts from the “Create Alert” dialog box in the charts UI are also able to use these placeholders in the dialog box’s “Message” field.

### `{{exchange}}`

Exchange of the symbol used in the alert (NASDAQ, NYSE, MOEX, etc.). Note that for delayed symbols, the exchange will end with “\_DL” or “\_DLY.” For example, “NYMEX\_DL.”

### `{{interval}}`

Returns the timeframe of the chart the alert is created on. Note that Range charts are calculated based on 1m data, so the placeholder will always return “1” on any alert created on a Range chart.

### `{{open}}, {{high}}, {{low}}, {{close}}, {{volume}}`

Corresponding values of the bar on which the alert has been triggered.

### `{{plot_0}}, {{plot_1}}, [...], {{plot_19}}`

Value of the corresponding plot number. Plots are numbered from zero to 19 in order of appearance in the script,

so only one of the first 20 plots can be used. For example, the built-in “Volume” indicator has two output series: Volume and Volume MA, so you could use the following:

```
alertcondition(volume > ta.sma(volume, 20), "Volume alert", "Volume {{plot_0}} >_<br/>average {{plot_1}}")
```

### `{{plot("[plot_title])}}`

This placeholder can be used when one needs to refer to a plot using the `title` argument used in a `plot()` call.

Note that double quotation marks ("") **must** be used inside the placeholder to wrap the `title` argument. This requires that a single quotation mark ('') be used to wrap the message string:

```
1 //@version=5
2 indicator("")
3 r = ta.rsi(close, 14)
4 xUp = ta.crossover(r, 50)
5 plot(r, "RSI", display = display.none)
6 alertcondition(xUp, "xUp alert", message = 'RSI is bullish at: {{plot("RSI")}}')
```

### `{{ticker}}`

Ticker of the symbol used in the alert (AAPL, BTCUSD, etc.).

### `{{time}}`

Returns the time at the beginning of the bar. Time is UTC, formatted as yyyy-MM-ddTHH:mm:ssZ, so for example: 2019-08-27T09:56:00Z.

### `{{timenow}}`

Current time when the alert triggers, formatted in the same way as `{{time}}`. The precision is to the nearest second, regardless of the chart's timeframe.

### 4.1.4 Avoiding repainting with alerts

The most common instances of repainting traders want to avoid with alerts are ones where they must prevent an alert from triggering at some point during the realtime bar when it would **not** have triggered at its close. This can happen when these conditions are met:

- The calculations used in the condition triggering the alert can vary during the realtime bar. This will be the case with any calculation using `high`, `low` or `close`, for example, which includes almost all built-in indicators. It will also be the case with the result of any `request.security()` call using a higher timeframe than the chart's, when the higher timeframe's current bar has not closed yet.
- The alert can trigger before the close of the realtime bar, so with any frequency other than “Once Per Bar Close”.

The simplest way to avoid this type of repainting is to configure the triggering frequency of alerts so they only trigger on the close of the realtime bar. There is no panacea; avoiding this type of repainting **always** entails waiting for confirmed information, which means the trader must sacrifice immediacy to achieve reliability.

Note that other types of repainting such as those documented in our [Repainting](#) section may not be preventable by simply triggering alerts on the close of realtime bars.





## 4.2 Backgrounds

The `bgcolor()` function changes the color of the script's background. If the script is running in `overlay = true` mode, then it will color the chart's background.

The function's signature is:

```
bgcolor(color, offset, editable, show_last, title) → void
```

Its `color` parameter allows a “series color” to be used for its argument, so it can be dynamically calculated in an expression.

If the correct transparency is not part of the color to be used, it can be generated using the `color.new()` function.

Here is a script that colors the background of trading sessions (try it on 30min EURUSD, for example):

```

1 // @version=5
2 indicator("Session backgrounds", overlay = true)
3
4 // Default color constants using transparency of 25.
5 BLUE_COLOR    = #0050FF40
6 PURPLE_COLOR  = #0000FF40
7 PINK_COLOR    = #5000FF40
8 NO_COLOR      = color(na)
9
10 // Allow user to change the colors.
11 preMarketColor = input.color(BLUE_COLOR, "Pre-market")
12 regSessionColor = input.color(PURPLE_COLOR, "Pre-market")
13 postMarketColor = input.color(PINK_COLOR, "Pre-market")
14
15 // Function returns `true` when the bar's time is
16 timeInRange(tf, session) =>
17     time(tf, session) != 0
18
19 // Function prints a message at the bottom-right of the chart.
20 f_print(_text) =>
21     var table _t = table.new(position.bottom_right, 1, 1)
22     table.cell(_t, 0, 0, _text, bgcolor = color.yellow)
23
24 var chartIs30MinOrLess = timeframe.isseconds or (timeframe.isintraday and timeframe.
25     ↪multiplier <=30)
26 sessionColor = if chartIs30MinOrLess
27     switch
28         timeInRange(timeframe.period, "0400-0930") => preMarketColor
29         timeInRange(timeframe.period, "0930-1600") => regSessionColor
30         timeInRange(timeframe.period, "1600-2000") => postMarketColor
31     else
32         => NO_COLOR
33     f_print("No background is displayed.\nChart timeframe must be <= 30min.")

```

(continues on next page)

(continued from previous page)

```
33 NO_COLOR
34
35 bgcolor(sessionColor)
```



Note that:

- The script only works on chart timeframes of 30min or less. It prints an error message when the chart's timeframe is higher than 30min.
- When the `if` structure's `else` branch is used because the chart's timeframe is incorrect, the local block returns the `NO_COLOR` color so that no background is displayed in that case.
- We first initialize constants using our base colors, which include the `40` transparency in hex notation at the end. `40` in the hexadecimal notation on the reversed `00-FF` scale for transparency corresponds to `75` in Pine Script™'s `0-100` decimal scale for transparency.
- We provide color inputs allowing script users to change the default colors we propose.

In our next example, we generate a gradient for the background of a CCI line:

```
1 // @version=5
2 indicator("CCI Background")
3
4 bullColor = input.color(color.lime, "#", inline = "1")
5 bearColor = input.color(color.fuchsia, "#", inline = "1")
6
7 // Calculate CCI.
8 myCCI = ta.cci(hlc3, 20)
9 // Get relative position of CCI in last 100 bars, on a 0-100% scale.
10 myCCIPosition = ta.percentrank(myCCI, 100)
11 // Generate a bull gradient when position is 50-100%, bear gradient when position is
12 // 0-50%.
13 backgroundColor = if myCCIPosition >= 50
14     color.from_gradient(myCCIPosition, 50, 100, color.new(bullColor, 75), bullColor)
15 else
16     color.from_gradient(myCCIPosition, 0, 50, bearColor, color.new(bearColor, 75))
17
18 // Wider white line background.
19 plot(myCCI, "CCI", color.white, 3)
20 // Thin black line.
21 plot(myCCI, "CCI", color.black, 1)
22 // Zero level.
```

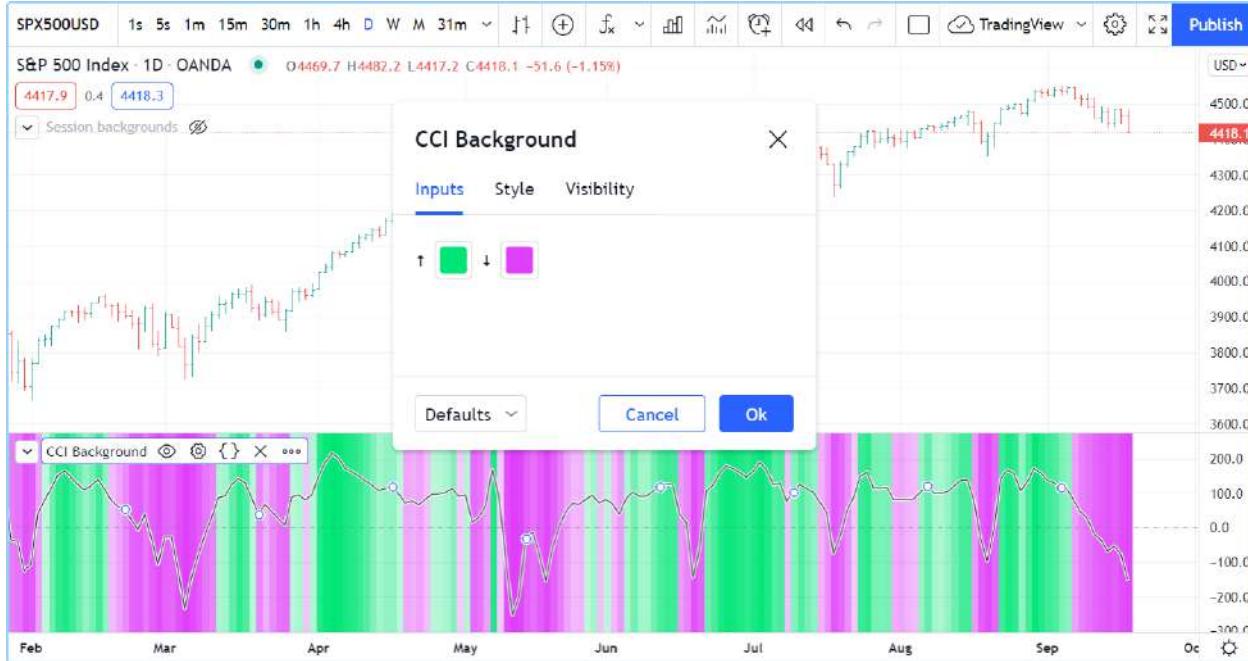
(continues on next page)

(continued from previous page)

```

22 hline(0)
23 // Gradient background.
24 bgcolor(backgroundColor)

```



Note that:

- We use the `ta.cci()` built-in function to calculate the indicator value.
- We use the `ta.percentrank()` built-in function to calculate `myCCIPosition`, i.e., the percentage of past `myCCI` values in the last 100 bars that are below the current value of `myCCI`.
- To calculate the gradient, we use two different calls of the `color.from_gradient()` built-in: one for the bull gradient when `myCCIPosition` is in the 50-100% range, which means that more past values are below its current value, and another for the bear gradient when `myCCIPosition` is in the 0-49.99% range, which means that more past values are above it.
- We provide inputs so the user can change the bull/bear colors, and we place both color input widgets on the same line using `inline = "1"` in both `input.color()` calls.
- We plot the CCI signal using two `plot()` calls to achieve the best contrast over the busy background: the first plot is a 3-pixel wide white background, the second `plot()` call plots the thin, 1-pixel wide black line.

See the [Colors](#) page for more examples of backgrounds.



## 4.3 Bar coloring

The `barcolor()` function lets you color chart bars. It is the only Pine Script™ function that allows a script running in a pane to affect the chart.

The function's signature is:

```
barcolor(color, offset, editable, show_last, title) → void
```

The coloring can be conditional because the `color` parameter accepts “series color” arguments.

The following script renders *inside* and *outside* bars in different colors:



```
1 // @version=5
2 indicator("barcolor example", overlay = true)
3 isUp = close > open
4 isDown = close <= open
5 isOutsideUp = high > high[1] and low < low[1] and isUp
6 isOutsideDown = high > high[1] and low < low[1] and isDown
7 isInside = high < high[1] and low > low[1]
8 barcolor(isInside ? color.yellow : isOutsideUp ? color.aqua : isOutsideDown ? color.
   ↪purple : na)
```

Note that:

- The `na` value leaves bars as is.
- In the `barcolor()` call, we use embedded ?: ternary operator expressions to select the color.



## 4.4 Bar plotting

- *Introduction*
- *Plotting candles with `plotcandle()`*
- *Plotting bars with `plotbar()`*

### 4.4.1 Introduction

The `plotcandle()` built-in function is used to plot candles. `plotbar()` is used to plot conventional bars.

Both functions require four arguments that will be used for the OHLC prices (`open`, `high`, `low`, `close`) of the bars they will be plotting. If one of those is `na`, no bar is plotted.

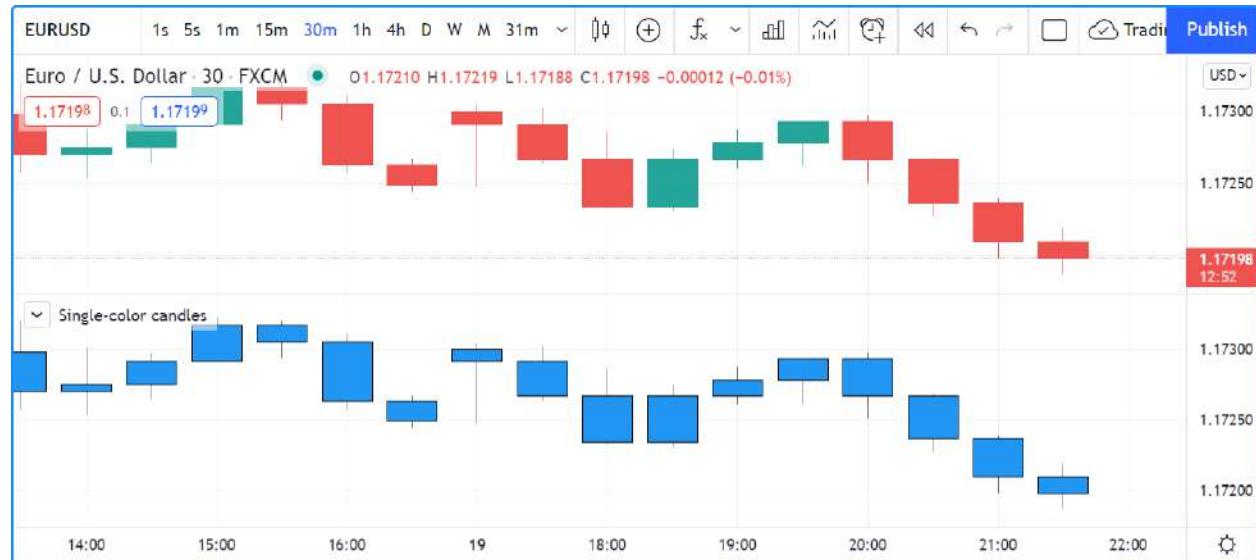
### 4.4.2 Plotting candles with `plotcandle()`

The signature of `plotcandle()` is:

```
plotcandle(open, high, low, close, title, color, wickcolor, editable, show_last,_
→bordercolor, display) → void
```

This plots simple candles, all in blue, using the habitual OHLC values, in a separate pane:

```
1 //@version=5
2 indicator("Single-color candles")
3 plotcandle(open, high, low, close)
```



To color them green or red, we can use the following code:

```
1 //@version=5
2 indicator("Example 2")
3 paletteColor = close >= open ? color.lime : color.red
4 plotbar(open, high, low, close, color = paletteColor)
```



Note that the `color` parameter accepts “series color” arguments, so constant values such as `color.red`, `color.lime`, “#FF9090”, as well as expressions that calculate colors at runtime, as is done with the `paletteColor` variable here, will all work.

You can build bars or candles using values other than the actual OHLC values. For example you could calculate and plot smoothed candles using the following code, which also colors wicks depending on the position of `close` relative to the smoothed close (`c`) of our indicator:

```

1 //@version=5
2 indicator("Smoothed candles", overlay = true)
3 lenInput = input.int(9)
4 smooth(source, length) =>
5     ta.sma(source, length)
6 o = smooth(open, lenInput)
7 h = smooth(high, lenInput)
8 l = smooth(low, lenInput)
9 c = smooth(close, lenInput)
10 ourWickColor = close > c ? color.green : color.red
11 plotcandle(o, h, l, c, wickcolor = ourWickColor)

```

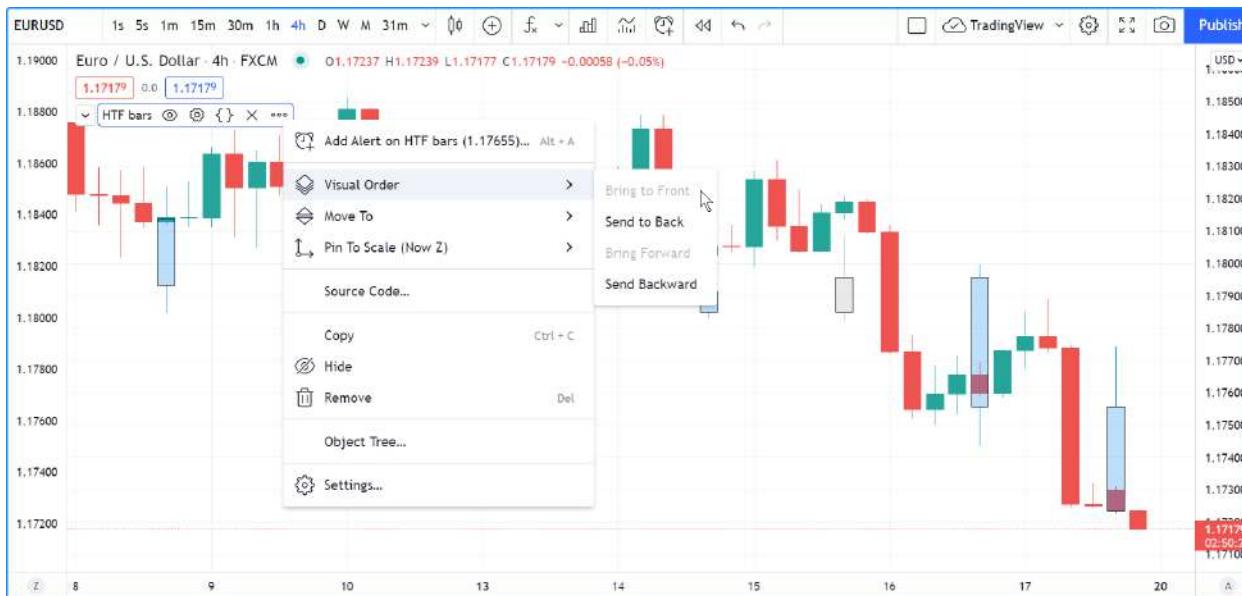


You may find it useful to plot OHLC values taken from a higher timeframe. You can, for example, plot daily bars on an intraday chart:

```

1 // NOTE: Use this script on an intraday chart.
2 //@version=5
3 indicator("Daily bars")
4
5 // Use gaps to only return data when the 1D timeframe completes, `na` otherwise.
6 [o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close], gaps_
7   ↪= barmerge.gaps_on)
8
9 var color UP_COLOR = color.silver
10 var color DN_COLOR = color.blue
11 color wickColor = c >= o ? UP_COLOR : DN_COLOR
12 color bodyColor = c >= o ? color.new(UP_COLOR, 70) : color.new(DN_COLOR, 70)
13 // Only plot candles on intraday timeframes,
14 // and when non `na` values are returned by `request.security()` because a HTF has_
15   ↪completed.
16 plotcandle(timeframe.isintraday ? o : na, h, l, c, color = bodyColor, wickcolor =
17   ↪wickColor)

```



Note that:

- We show the script's plot after having used “Visual Order/Bring to Front” from the script's “More” menu. This causes our script's candles to appear on top of the chart's candles.
- The script will only display candles when two conditions are met:
  - The chart is using an intraday timeframe (see the check on `timeframe.isintraday` in the `plotcandle()` call). We do this because it's not useful to show a daily value on timeframes higher or equal to 1D.
  - The `request.security()` function returns non `na` values (see `gaps = barmerge.gaps_on` in the function call).
- We use a tuple (`[open, high, low, close]`) with `request.security()` to fetch four values in one call.
- We use `var` to declare our `UP_COLOR` and `DN_COLOR` color constants on bar zero only. We use constants because those colors are used in more than one place in our code. This way, if we need to change them, we need only do so in one place.

- We create a lighter transparency for the body of our candles in the `bodyColor` variable initialization, so they don't obstruct the chart's candles.

#### 4.4.3 Plotting bars with `plotbar()`

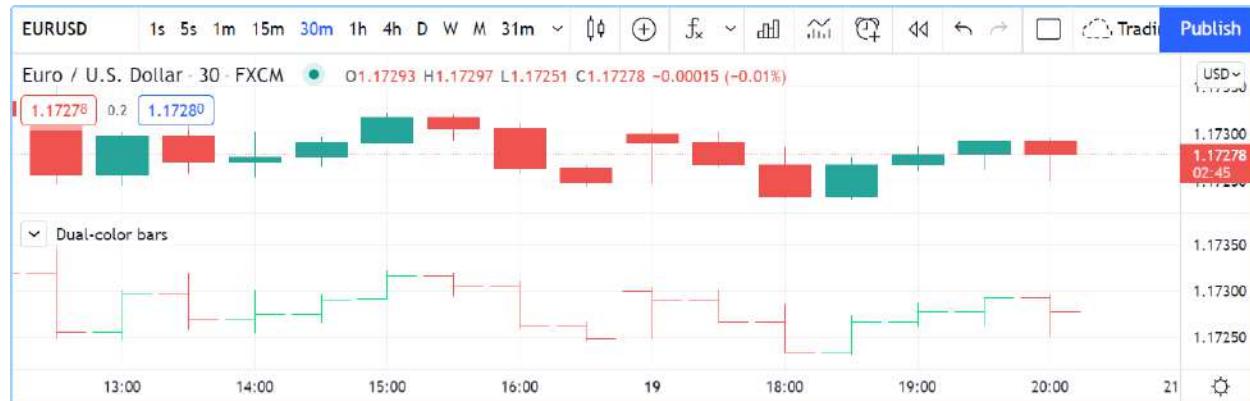
The signature of `plotbar()` is:

```
plotbar(open, high, low, close, title, color, editable, show_last, display) → void
```

Note that `plotbar()` has no parameter for `bordercolor` or `wickcolor`, as there are no borders or wicks on conventional bars.

This plots conventional bars using the same coloring logic as in the second example of the previous section:

```
1 //@version=5
2 indicator("Dual-color bars")
3 paletteColor = close >= open ? color.lime : color.red
4 plotbar(open, high, low, close, color = paletteColor)
```



 **TradingView**



#### 4.5 Bar states

- *Introduction*
- *Bar state built-in variables*
- *Example*

## 4.5.1 Introduction

A set of built-in variables in the `barstate` namespace allow your script to detect different properties of the bar on which the script is currently executing.

These states can be used to restrict the execution or the logic of your code to specific bars.

Some built-ins return information on the trading session the current bar belongs to. They are explained in the [Session states](#) section.

## 4.5.2 Bar state built-in variables

Note that while indicators and libraries run on all price or volume updates in real time, strategies not using `calc_on_every_tick` will not; they will only execute when the realtime bar closes. This will affect the detection of bar states in that type of script. On open markets, for example, this code will not display a background until the realtime closes because that is when the strategy runs:

```
1 // @version=5
2 strategy("S")
3 bgcolor(barstate.islast ? color.silver : na)
```

### 'barstate.isfirst'

`barstate.isfirst` is only `true` on the dataset's first bar, i.e., when `bar_index` is zero.

It can be useful to initialize variables on the first bar only, e.g.:

```
1 // Declare array and set its values on the first bar only.
2 FILL_COLOR = color.green
3 var fillColors = array.new_color(0)
4 if barstate.isfirst
5     // Initialize the array elements with progressively lighter shades of the fill_
6     ↪color.
7     array.push(fillColors, color.new(FILL_COLOR, 70))
8     array.push(fillColors, color.new(FILL_COLOR, 75))
9     array.push(fillColors, color.new(FILL_COLOR, 80))
10    array.push(fillColors, color.new(FILL_COLOR, 85))
11    array.push(fillColors, color.new(FILL_COLOR, 90))
```

### 'barstate.islast'

`barstate.islast` is `true` if the current bar is the last one on the chart, whether that bar is a realtime bar or not.

It can be used to restrict the execution of code to the chart's last bar, which is often useful when drawing lines, labels or tables. Here, we use it to determine when to update a label which we want to appear only on the last bar. We create the label only once and then update its properties using `label.set_*` functions because it is more efficient:

```
1 // @version=5
2 indicator("", "", true)
3 // Create label on the first bar only.
4 var label hiLabel = label.new(na, na, "")
5 // Update the label's position and text on the last bar,
6 // including on all realtime bar updates.
7 if barstate.islast
```

(continues on next page)

(continued from previous page)

```

8     label.set_xy(hiLabel, bar_index, high)
9     label.set_text(hiLabel, str.tostring(high, format.mintick))

```

### **'barstate.ishistory'**

`barstate.ishistory` is `true` on all historical bars. It can never be `true` on a bar when `barstate.isrealtime` is also `true`, and it does not become `true` on a realtime bar's closing update, when `barstate.isconfirmed` becomes `true`. On closed markets, it can be `true` on the same bar where `barstate.islast` is also `true`.

### **'barstate.isrealtime'**

`barstate.isrealtime` is `true` if the current data update is a real-time bar update, `false` otherwise (thus it is historical). Note that `barstate.islast` is also `true` on all realtime bars.

### **'barstate.isnew'**

`barstate.isnew` is `true` on all historical bars and on the realtime bar's first (opening) update.

All historical bars are considered *new* bars because the Pine Script™ runtime executes your script on each bar sequentially, from the chart's first bar in time, to the last. Each historical bar is thus *discovered* by your script as it executes, bar to bar.

`barstate.isnew` can be useful to reset `varip` variables when a new realtime bar comes in. The following code will reset `updateNo` to 1 on all historical bars and at the beginning of each realtime bar. It calculates the number of realtime updates during each realtime bar:

```

1 // @version=5
2 indicator("")
3 updateNo() =>
4     varip int updateNo = na
5     if barstate.isnew
6         updateNo := 1
7     else
8         updateNo += 1
9 plot(updateNo())

```

### **'barstate.isconfirmed'**

`barstate.isconfirmed` is `true` on all historical bars and on the last (closing) update of a realtime bar.

It can be useful to avoid repainting by requiring the realtime bar to be closed before a condition can become `true`. We use it here to hold plotting of our RSI until the realtime bar closes and becomes an elapsed realtime bar. It will plot on historical bars because `barstate.isconfirmed` is always `true` on them:

```

1 // @version=5
2 indicator("")
3 myRSI = ta.rsi(close, 20)
4 plot(barstate.isconfirmed ? myRSI : na)

```

`barstate.isconfirmed` will not work when used in a `request.security()` call.

### **`barstate.islastconfirmedhistory`**

`barstate.islastconfirmedhistory` is `true` if the script is executing on the dataset's last bar when the market is closed, or on the bar immediately preceding the realtime bar if the market is open.

It can be used to detect the first realtime bar with `barstate.islastconfirmedhistory[1]`, or to postpone server-intensive calculations until the last historical bar, which would otherwise be undetectable on open markets.

#### **4.5.3 Example**

Here is an example of a script using `barstate.*` variables:

```

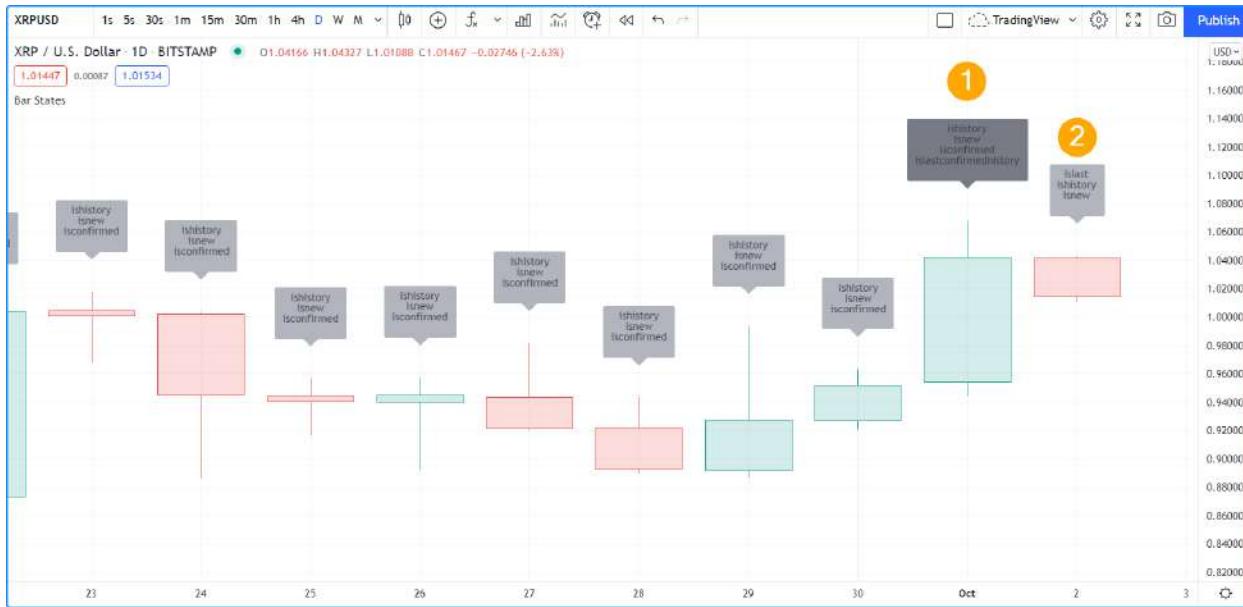
1 //@version=5
2 indicator("Bar States", overlay = true, max_labels_count = 500)
3
4 stateText() =>
5     string txt = ""
6     txt += barstate.isfirst ? "isfirst\n" : ""
7     txt += barstate.islast ? "islast\n" : ""
8     txt += barstate.ishistory ? "ishistory\n" : ""
9     txt += barstate.isrealtime ? "isrealtime\n" : ""
10    txt += barstate.isnew ? "isnew\n" : ""
11    txt += barstate.isconfirmed ? "isconfirmed\n" : ""
12    txt += barstate.islastconfirmedhistory ? "islastconfirmedhistory\n" : ""
13
14 labelColor = switch
15     barstate.isfirst => color.fuchsia
16     barstate.islastconfirmedhistory => color.gray
17     barstate.ishistory => color.silver
18     barstate.isconfirmed => color.orange
19     barstate.isnew => color.red
20     => color.yellow
21
22 label.new(bar_index, na, stateText(), yloc = yloc.abovebar, color = labelColor)

```

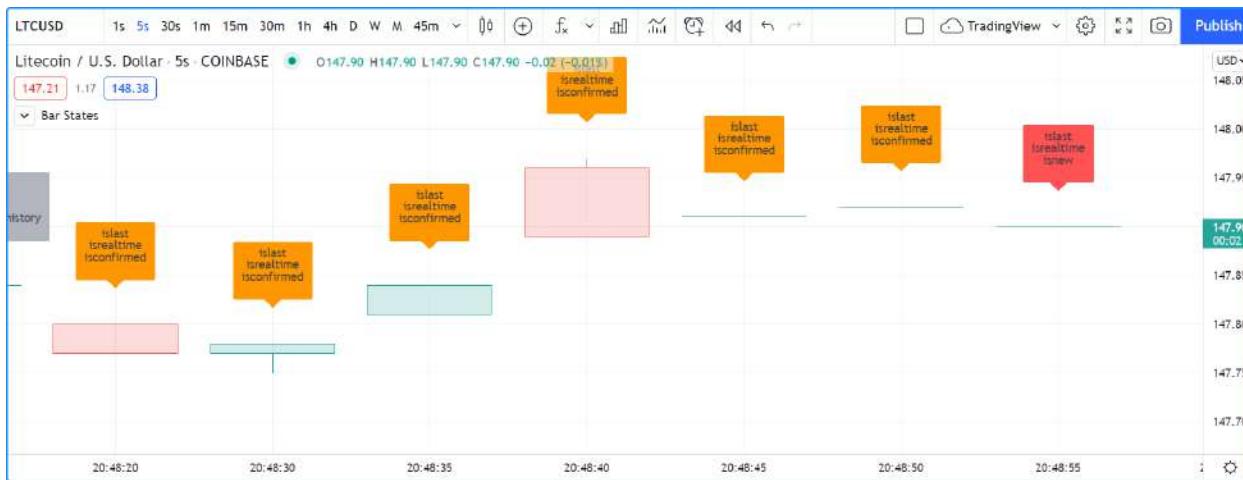
Note that:

- Each state's name will appear in the label's text when it is `true`.
- There are five possible colors for the label's background:
  - fuchsia on the first bar
  - silver on historical bars
  - gray on the last confirmed historical bar
  - orange when a realtime bar is confirmed (when it closes and becomes an elapsed realtime bar)
  - red on the realtime bar's first execution
  - yellow for other executions of the realtime bar

We begin by adding the indicator to the chart of an open market, but before any realtime update is received. Note how the last confirmed history bar is identified in #1, and how the last bar is identified as the last one, but is still considered a historical bar because no realtime updates have been received.



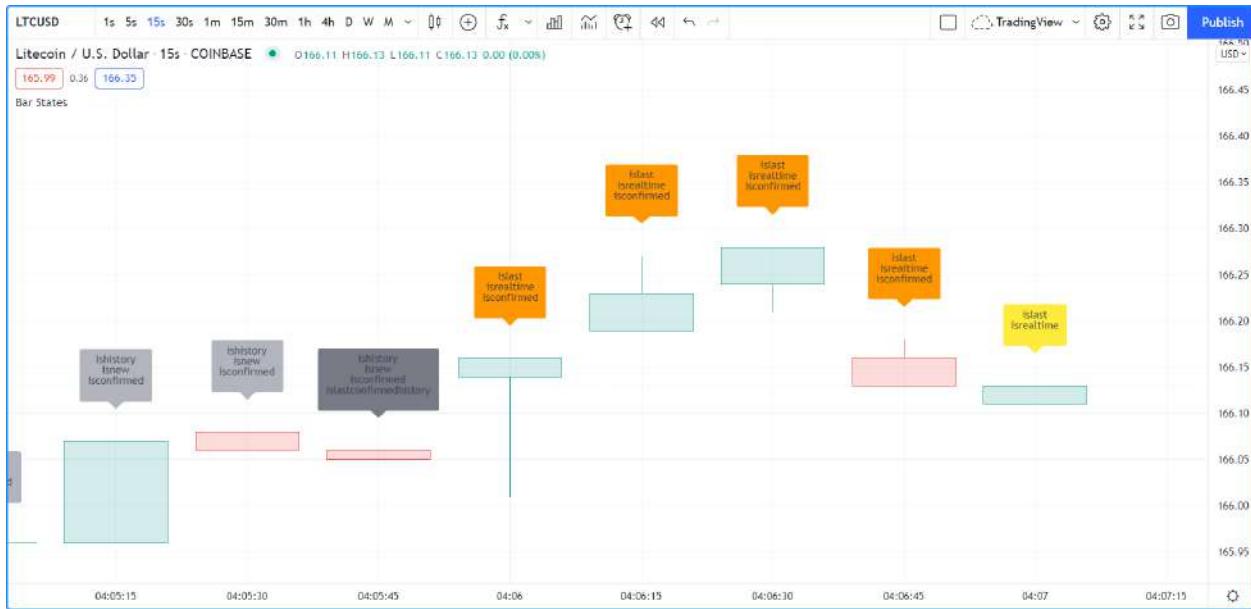
Let's look at what happens when realtime updates start coming in:



Note that:

- The realtime bar is red because it is its first execution, because `barstate.isnew` is true and `barstate.ishistory` is no longer true, so our `switch` structure determining our color uses the `barstate.isnew => color.red` branch. This will usually not last long because on the next update `barstate.isnew` will no longer be true so the label's color will turn yellow.
- The label of elapsed realtime bars is orange because those bars were not historical bars when they closed. Accordingly, the `barstate.ishistory => color.silver` branch in the `switch` structure was not executed, but the next one, `barstate.isconfirmed => color.orange` was.

This last example shows how the realtime bar's label will turn yellow after the first execution on the bar. This is the way the label will usually appear on realtime bars:



 **TradingView**



## 4.6 Chart information

- *Introduction*
- *Prices and volume*
- *Symbol information*
- *Chart timeframe*
- *Session information*

### 4.6.1 Introduction

The way scripts can obtain information about the chart and symbol they are currently running on is through a subset of Pine Script™'s *built-in variables*. The ones we cover here allow scripts to access information relating to:

- The chart's prices and volume
- The chart's symbol
- The chart's timeframe
- The session (or time period) the symbol trades on

## 4.6.2 Prices and volume

The built-in variables for OHLCV values are:

- `open`: the bar's opening price.
- `high`: the bar's highest price, or the highest price reached during the realtime bar's elapsed time.
- `low`: the bar's lowest price, or the lowest price reached during the realtime bar's elapsed time.
- `close`: the bar's closing price, or the **current price** in the realtime bar.
- `volume`: the volume traded during the bar, or the volume traded during the realtime bar's elapsed time. The unit of volume information varies with the instrument. It is in shares for stocks, in lots for forex, in contracts for futures, in the base currency for crypto, etc.

Other values are available through:

- `hl2`: the average of the bar's `high` and `low` values.
- `hlc3`: the average of the bar's `high`, `low` and `close` values.
- `ohlc4`: the average of the bar's `open`, `high`, `low` and `close` values.

On historical bars, the values of the above variables do not vary during the bar because only OHLCV information is available on them. When running on historical bars, scripts execute on the bar's `close`, when all the bar's information is known and cannot change during the script's execution on the bar.

Realtime bars are another story altogether. When indicators (or strategies using `calc_on_every_tick = true`) run in realtime, the values of the above variables (except `open`) will vary between successive iterations of the script on the realtime bar, because they represent their **current** value at one point in time during the progress of the realtime bar. This may lead to one form of *repainting*. See the page on Pine Script™'s *execution model* for more details.

The `[] history-referencing operator` can be used to refer to past values of the built-in variables, e.g., `close[1]` refers to the value of `close` on the previous bar, relative to the particular bar the script is executing on.

## 4.6.3 Symbol information

Built-in variables in the `syminfo` namespace provide scripts with information on the symbol of the chart the script is running on. This information changes every time a script user changes the chart's symbol. The script then re-executes on all the chart's bars using the new values of the built-in variables:

- `syminfo.basecurrency`: the base currency, e.g., "BTC" in "BTCUSD", or "EUR" in "EURUSD".
- `syminfo.currency`: the quote currency, e.g., "USD" in "BTCUSD", or "CAD" in "USDCAD".
- `syminfo.description`: The long description of the symbol.
- `syminfo.mintick`: The symbol's tick value, or the minimum increment price can move in. Not to be confused with *pips* or *points*. On "ES1!" ("S&P 500 E-Mini") the tick size is 0.25 because that is the minimal increment the price moves in.
- `syminfo.pointvalue`: The point value is the multiple of the underlying asset determining a contract's value. On "ES1!" ("S&P 500 E-Mini") the point value is 50, so a contract is worth 50 times the price of the instrument.
- `syminfo.prefix`: The prefix is the exchange or broker's identifier: "NASDAQ" or "BATS" for "AAPL", "CME\_MINI\_DL" for "ES1!".
- `syminfo.root`: It is the ticker's prefix for structured tickers like those of futures. It is "ES" for "ES1!", "ZW" for "ZW1!".

- `syminfo.session`: It reflects the session setting on the chart for that symbol. If the “Chart settings/Symbol/Session” field is set to “Extended”, it will only return “extended” if the symbol and the user’s feed allow for extended sessions. It is rarely displayed and used mostly as an argument to the `session` parameter in `ticker.new()`.
- `syminfo.ticker`: It is the symbol’s name, without the exchange part (`syminfo.prefix`): “BTCUSD”, “AAPL”, “ES1！”, “USDCAD”.
- `syminfo.tickerid`: This string is rarely displayed. It is mostly used as an argument for `request.security()`’s `symbol` parameter. It includes session, prefix and ticker information.
- `syminfo.timezone`: The timezone the symbol is traded in. The string is an IANA time zone database name (e.g., “America/New\_York”).
- `syminfo.type`: The type of market the symbol belongs to. The values are “stock”, “futures”, “index”, “forex”, “crypto”, “fund”, “dr”, “cfd”, “bond”, “warrant”, “structured” and “right”.

This script will display the values of those built-in variables on the chart:

```

1 // @version=5
2 indicator(``syminfo.*` built-ins", "", true)
3 printTable(txtLeft, txtRight) =>
4     var table t = table.new(position.middle_right, 2, 1)
5     table.cell(t, 0, 0, txtLeft, bgcolor = color.yellow, text_halign = text.align_
6         ↪right)
7     table.cell(t, 1, 0, txtRight, bgcolor = color.yellow, text_halign = text.align_
8         ↪left)
9
10 nl = "\n"
11 left =
12     "syminfo.basecurrency: " + nl +
13     "syminfo.currency: " + nl +
14     "syminfo.description: " + nl +
15     "syminfo.mintick: " + nl +
16     "syminfo.pointvalue: " + nl +
17     "syminfo.prefix: " + nl +
18     "syminfo.root: " + nl +
19     "syminfo.session: " + nl +
20     "syminfo.ticker: " + nl +
21     "syminfo.tickerid: " + nl +
22     "syminfo.timezone: " + nl +
23     "syminfo.type: "
24
25 right =
26     syminfo.basecurrency + nl +
27     syminfo.currency + nl +
28     syminfo.description + nl +
29     str.tostring(syminfo.mintick) + nl +
30     str.tostring(syminfo.pointvalue) + nl +
31     syminfo.prefix + nl +
32     syminfo.root + nl +
33     syminfo.session + nl +
34     syminfo.ticker + nl +
35     syminfo.tickerid + nl +
36     syminfo.timezone + nl +
37     syminfo.type
38
39 printTable(left, right)

```

#### 4.6.4 Chart timeframe

A script can obtain information on the type of timeframe used on the chart using these built-ins, which all return a “simple bool” result:

- `timeframe.isseconds`
- `timeframe.isminutes`
- `timeframe.isintraday`
- `timeframe.isdaily`
- `timeframe.isweekly`
- `timeframe.ismonthly`
- `timeframe.isdwm`

Two additional built-ins return more specific timeframe information:

- `timeframe.multiplier` returns a “simple int” containing the multiplier of the timeframe unit. A chart timeframe of one hour will return 60 because intraday timeframes are expressed in minutes. A 30sec timeframe will return 30 (seconds), a daily chart will return 1 (day), a quarterly chart will return 3 (months), and a yearly chart will return 12 (months). The value of this variable cannot be used as an argument to `timeframe` parameters in built-in functions, as they expect a string in timeframe specifications format.
- `timeframe.period` returns a string in Pine Script™’s timeframe specification format.

See the page on [Timeframes](#) for more information.

#### 4.6.5 Session information

Session information is available in different forms:

- The `syminfo.session` built-in variable returns a value that is either `session.regular` or `session.extended`. It reflects the session setting on the chart for that symbol. If the “Chart settings/Symbol/Session” field is set to “Extended”, it will only return “extended” if the symbol and the user’s feed allow for extended sessions. It is used when a session type is expected, for example as the argument for the `session` parameter in `ticker.new()`.
- [Session state built-ins](#) provide information on the trading session a bar belongs to.



## 4.7 Colors

- *Introduction*
- *Constant colors*
- *Conditional coloring*
- *Calculated colors*
- *Mixing transparencies*
- *Tips*

### 4.7.1 Introduction

Script visuals can play a critical role in the usability of the indicators we write in Pine Script™. Well-designed plots and drawings make indicators easier to use and understand. Good visual designs establish a visual hierarchy that allows the more important information to stand out, and the less important one to not get in the way.

Using colors in Pine can be as simple as you want, or as involved as your concept requires. The 4,294,967,296 possible assemblies of color and transparency available in Pine Script™ can be applied to:

- Any element you can plot or draw in an indicator's visual space, be it lines, fills, text or candles.
- The background of a script's visual space, whether the script is running in its own pane, or in overlay mode on the chart.
- The color of bars or the body of candles appearing on a chart.

A script can only color the elements it places in its own visual space. The only exception to this rule is that a pane indicator can color chart bars or candles.

Pine Script™ has built-in colors such as `color.green`, as well as functions like `color.rgb()` which allow you to dynamically generate any color in the RGBA color space.

### Transparency

Each color in Pine Script™ is defined by four values:

- Its red, green and blue components (0-255), following the [RGB color model](#).
- Its transparency (0-100), often referred to as the Alpha channel outside Pine, as defined in the [RGBA color model](#). Even though transparency is expressed in the 0-100 range, its value can be a “float” when used in functions, which gives you access to the 256 underlying values of the alpha channel.

The transparency of a color defines how opaque it is: zero is fully opaque, 100 makes the color—whichever it is—invisible. Modulating transparency can be crucial in more involved color visuals or when using backgrounds, to control which colors dominate the others, and how they mix together when superimposed.

## Z-index

When you place elements in a script's visual space, they have relative depth on the *z* axis; some will appear on top of others. The *z-index* is a value that represents the position of elements on the *z* axis. Elements with the highest z-index appear on top.

Elements drawn in Pine Script™ are divided in groups. Each group has its own position in the *z* space, and **within the same group**, elements created last in the script's logic will appear on top of other elements from the same group. An element of one group cannot be placed outside the region of the *z* space attributed to its group, so a plot can never appear on top of a table, for example, because tables have the highest z-index.

This list contains the groups of visual elements, ordered by increasing z-index, so background colors are always at the bottom of *z* space, and tables will always appear on top of all other elements:

- Background colors
- Fills
- Plots
- Hlines
- LineFills
- Lines
- Boxes
- Labels
- Tables

Note that by using `explicit_plot_zorder = true` in [indicator\(\)](#) or [strategy\(\)](#), you can control the relative z-index of `plot*()`, `hline()` and `fill()` visuals using their sequential order in the script.

## 4.7.2 Constant colors

There are 17 built-in colors in Pine Script™. This table lists their names, hexadecimal equivalent, and RGB values as arguments to `color.rgb()`:

Name	Hex	RGB values
color.aqua	#00BCD4	color.rgb(0, 188, 212)
color.black	#363A45	color.rgb(54, 58, 69)
color.blue	#2196F3	color.rgb(33, 150, 243)
color.fuchsia	#E040FB	color.rgb(224, 64, 251)
color.gray	#787B86	color.rgb(120, 123, 134)
color.green	#4CAF50	color.rgb(76, 175, 80)
color.lime	#00E676	color.rgb(0, 230, 118)
color.maroon	#880E4F	color.rgb(136, 14, 79)
color.navy	#311B92	color.rgb(49, 27, 146)
color.olive	#808000	color.rgb(128, 128, 0)
color.orange	#FF9800	color.rgb(255, 152, 0)
color.purple	#9C27B0	color.rgb(156, 39, 176)
color.red	#FF5252	color.rgb(255, 82, 82)
color.silver	#B2B5BE	color.rgb(178, 181, 190)
color.teal	#00897B	color.rgb(0, 137, 123)
color.white	#FFFFFF	color.rgb(255, 255, 255)
color.yellow	#FFEB3B	color.rgb(255, 235, 59)

In the following script, all plots use the same `color.olive` color with a transparency of 40, but expressed in different ways. All five methods are functionally equivalent:



```

1 // @version=5
2 indicator("", "", true)
3 // — Transparency (#99) is included in the hex value.
4 plot(ta.sma(close, 10), "10", #80800099)
5 // — Transparency is included in the color-generating function's arguments.
6 plot(ta.sma(close, 30), "30", color.new(color.olive, 40))
7 plot(ta.sma(close, 50), "50", color.rgb(128, 128, 0, 40))
8     // — Use `transp` parameter (deprecated and advised against)
9 plot(ta.sma(close, 70), "70", color.olive, transp = 40)
10 plot(ta.sma(close, 90), "90", #808000, transp = 40)

```

**Note:** The last two `plot()` calls specify transparency using the `transp` parameter. This use should be avoided as the `transp` is deprecated in Pine Script™ v5. Using the `transp` parameter to define transparency is not as flexible because it requires an argument of *input integer* type, which entails it must be known before the script is executed, and so cannot be calculated dynamically, as your script executes bar to bar. Additionally, if you use a `color` argument that already includes transparency information, as is done in the next three `plot()` calls, any argument used for the `transp` parameter would have no effect. This is also true for other functions with a `transp` parameter.

The colors in the previous script do not vary as the script executes bar to bar. Sometimes, however, colors need to be created as the script executes on each bar because they depend on conditions that are unknown at compile time, or when the script begins execution on bar zero. For those cases, programmers have two options:

1. Use conditional statements to select colors from a few pre-determined base colors.
2. Build new colors dynamically, by calculating them as the script executes bar to bar, to implement color gradients, for example.

### 4.7.3 Conditional coloring

Let's say you want to color a moving average in different colors, depending on some conditions you define. To do so, you can use a conditional statement that will select a different color for each of your states. Let's start by coloring a moving average in a bull color when it's rising, and in a bear color when it's not:



```

1 //@version=5
2 indicator("Conditional colors", "", true)
3 int lengthInput = input.int(20, "Length", minval = 2)
4 color maBullColorInput = input.color(color.green, "Bull")
5 color maBearColorInput = input.color(color.maroon, "Bear")
6 float ma = ta.sma(close, lengthInput)
7 // Define our states.
8 bool maRising = ta.rising(ma, 1)
9 // Build our color.
10 color c_ma = maRising ? maBullColorInput : maBearColorInput
11 plot(ma, "MA", c_ma, 2)

```

Note that:

- We provide users of our script a selection of colors for our bull/bear colors.
- We define an `maRising` boolean variable which will hold `true` when the moving average is higher on the current bar than it was on the last.
- We define a `c_ma` color variable that is assigned one of our two colors, depending on the value of the `maRising` boolean. We use the `? :` ternary operator to write our conditional statement.

You can also use conditional colors to avoid plotting under certain conditions. Here, we plot high and low pivots using a line, but we do not want to plot anything when a new pivot comes in, to avoid the joints that would otherwise appear in pivot transitions. To do so, we test for pivot changes and use `na` as the color value when a change is detected, so that no line is plotted on that bar:



```

1 //@version=5
2 indicator("Conditional colors", "", true)
3 int legsInput = input.int(5, "Pivot Legs", minval = 1)
4 color pHiColorInput = input.color(color.olive, "High pivots")

```

(continues on next page)

(continued from previous page)

```

5 color pLoColorInput = input.color(color.orange, "Low pivots")
6 // Initialize the pivot level variables.
7 var float pHi = na
8 var float pLo = na
9 // When a new pivot is detected, save its value.
10 pHi := nz(ta.pivothigh(legsInput, legsInput), pHi)
11 pLo := nz(ta.pivotlow( legsInput, legsInput), pLo)
12 // When a new pivot is detected, do not plot a color.
13 plot(pHi, "High", ta.change(pHi) ? na : pHiColorInput, 2, plot.style_line)
14 plot(pLo, "Low", ta.change(pLo) ? na : pLoColorInput, 2, plot.style_line)

```

To understand how this code works, one must first know that `ta.pivothigh()` and `ta.pivotlow()`, used as they are here without an argument to the `source` parameter, will return a value when they find a high/low pivot, otherwise they return `na`.

When we test the value returned by the pivot function for `na` using the `nz()` function, we allow the value returned to be assigned to the `pHi` or `pLo` variables only when it is not `na`, otherwise the previous value of the variable is simply reassigned to it, which has no impact on its value. Keep in mind that previous values of `pHi` and `pLo` are preserved bar to bar because we use the `var` keyword when initializing them, which causes the initialization to only occur on the first bar.

All that's left to do next is, when we plot our lines, to insert a ternary conditional statement that will yield `na` for the color when the pivot value changes, or the color selected in the script's inputs when the pivot level does not change.

#### 4.7.4 Calculated colors

Using functions like `color.new()`, `color.rgb()` and `color.from_gradient()`, one can build colors on the fly, as the script executes bar to bar.

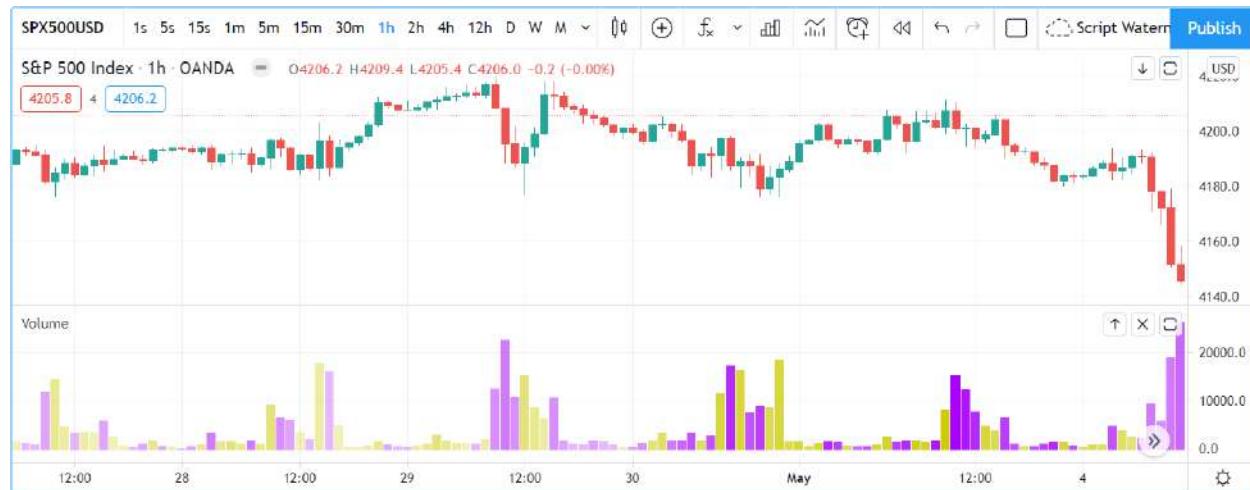
`color.new()` is most useful when you need to generate different transparency levels from a base color.

`color.rgb()` is useful when you need to build colors dynamically from red, green, blue, or transparency components. While `color.rgb()` creates a color, its sister functions `color.r()`, `color.g()`, `color.b()` and `color.t()` can be used to extract the red, green, blue or transparency values from a color, which can in turn be used to generate a variant.

`color.from_gradient()` is useful to create linear gradients between two base colors. It determines which intermediary color to use by evaluating a source value against minimum and maximum values.

### color.new()

Let's put `color.new(color, transp)` to use to create different transparencies for volume columns using one of two bull/bear base colors:



```

1 // @version=5
2 indicator("Volume")
3 // We name our color constants to make them more readable.
4 var color GOLD_COLOR = #CCCC00ff
5 var color VIOLET_COLOR = #AA00FFff
6 color bullColorInput = input.color(GOLD_COLOR, "Bull")
7 color bearColorInput = input.color(VIOLET_COLOR, "Bear")
8 int levelsInput = input.int(10, "Gradient levels", minval = 1)
9 // We initialize only once on bar zero with `var`, otherwise the count would reset to
10 // zero on each bar.
11 var float riseFallCnt = 0
12 // Count the rises/falls, clamping the range to: 1 to `i_levels`.
13 riseFallCnt := math.max(1, math.min(levelsInput, riseFallCnt + math.sign(volume -
14 nz(volume[1]))))
15 // Rescale the count on a scale of 80, reverse it and cap transparency to <80 so that
16 // colors remains visible.
17 float transparency = 80 - math.abs(80 * riseFallCnt / levelsInput)
18 // Build the correct transparency of either the bull or bear color.
19 color volumeColor = color.new(close > open ? bullColorInput : bearColorInput,
20 transparency)
21 plot(volume, "Volume", volumeColor, 1, plot.style_columns)

```

Note that:

- In the next to last line of our script, we dynamically calculate the column color by varying both the base color used, depending on whether the bar is up or down, **and** the transparency level, which is calculated from the cumulative rises or falls of volume.
- We offer the script user control over not only the base bull/bear colors used, but also on the number of brightness levels we use. We use this value to determine the maximum number of rises or falls we will track. Giving users the possibility to manage this value allows them to adapt the indicator's visuals to the timeframe or market they use.
- We take care to control the maximum level of transparency we use so that it never goes higher than 80. This ensures our colors always retain some visibility.
- We also set the minimum value for the number of levels to 1 in the inputs. When the user selects 1, the volume columns will be either in bull or bear color of maximum brightness—or transparency zero.

## color.rgb()

In our next example we use `color.rgb(red, green, blue, transp)` to build colors from RGBA values. We use the result in a holiday season gift for our friends, so they can bring their TradingView charts to parties:



```

1 // @version=5
2 indicator("Holiday candles", "", true)
3 float r = math.random(0, 255)
4 float g = math.random(0, 255)
5 float b = math.random(0, 255)
6 float t = math.random(0, 100)
7 color holidayColor = color.rgb(r, g, b, t)
8 plotcandle(open, high, low, close, color = c_holiday, wickcolor = holidayColor, ↴
    bordercolor = c_holiday)

```

Note that:

- We generate values in the zero to 255 range for the red, green and blue channels, and in the zero to 100 range for transparency. Also note that because `math.random()` returns float values, the float 0.0-100.0 range provides access to the full 0-255 transparency values of the underlying alpha channel.
- We use the `math.random(min, max, seed)` function to generate pseudo-random values. We do not use an argument for the third parameter of the function: `seed`. Using it is handy when you want to ensure the repeatability of the function's results. Called with the same seed, it will produce the same sequence of values.

## color.from\_gradient()

Our last examples of color calculations will use `color.from_gradient(value, bottom_value, top_value, bottom_color, top_color)`. Let's first use it in its simplest form, to color a CCI signal in a version of the indicator that otherwise looks like the built-in:



```

1 // @version=5
2 indicator(title="CCI line gradient", precision=2, timeframe="")
3 var color GOLD_COLOR    = #CCCC00
4 var color VIOLET_COLOR = #AA00FF

```

(continues on next page)

(continued from previous page)

```

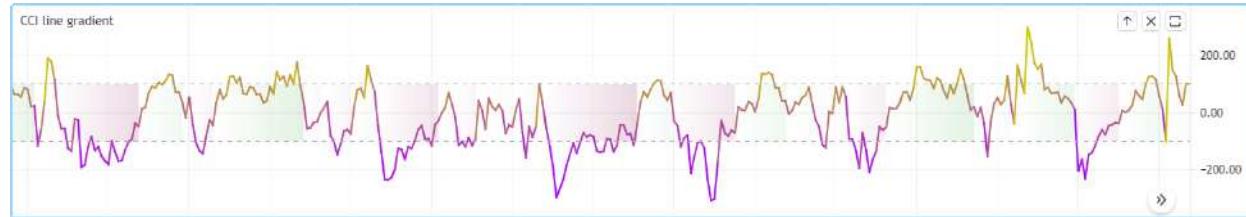
5 var color BEIGE_COLOR = #9C6E1B
6 float srcInput = input.source(close, title="Source")
7 int lenInput = input.int(20, "Length", minval = 5)
8 color bullColorInput = input.color(GOLD_COLOR, "Bull")
9 color bearColorInput = input.color(BEIGE_COLOR, "Bear")
10 float signal = ta.cci(srcInput, lenInput)
11 color signalColor = color.from_gradient(signal, -200, 200, bearColorInput, ↵
12   bullColorInput)
13 plot(signal, "CCI", signalColor)
14 bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed)
15 bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed)
16 fill(bandTopPlotID, bandBotPlotID, color.new(BEIGE_COLOR, 90), "Background")

```

Note that:

- To calculate the gradient, `color.from_gradient()` requires minimum and maximum values against which the argument used for the `value` parameter will be compared. The fact that we want a gradient for an unbounded signal like CCI (i.e., without fixed boundaries such as RSI, which always oscillates between 0-100), does not entail we cannot use `color.from_gradient()`. Here, we solve our conundrum by providing values of -200 and 200 as arguments. They do not represent the real minimum and maximum values for CCI, but they are at levels from which we do not mind the colors no longer changing, as whenever the series is outside the `bottom_value` and `top_value` limits, the colors used for `bottom_color` and `top_color` will apply.
- The color progression calculated by `color.from_gradient()` is linear. If the value of the series is halfway between the `bottom_value` and `top_value` arguments, the generated color's RGBA components will also be halfway between those of `bottom_color` and `top_color`.
- Many common indicator calculations are available in Pine Script™ as built-in functions. Here we use `ta.cci()` instead of calculating it the long way.

The argument used for `value` in `color.from_gradient()` does not necessarily have to be the value of the line we are calculating. Anything we want can be used, as long as arguments for `bottom_value` and `top_value` can be supplied. Here, we enhance our CCI indicator by coloring the band using the number of bars since the signal has been above/below the centerline:



```

1 //@version=5
2 indicator(title="CCI line gradient", precision=2, timeframe="")
3 var color GOLD_COLOR = #CCCC00
4 var color VIOLET_COLOR = #AA00FF
5 var color GREEN_BG_COLOR = color.new(color.green, 70)
6 var color RED_BG_COLOR = color.new(color.maroon, 70)
7 float srcInput = input.source(close, "Source")
8 int lenInput = input.int(20, "Length", minval = 5)
9 int stepsInput = input.int(50, "Gradient levels", minval = 1)
10 color bullColorInput = input.color(GOLD_COLOR, "Line: Bull", inline = "11")
11 color bearColorInput = input.color(VIOLET_COLOR, "Bear", inline = "11")
12 color bullBgColorInput = input.color(GREEN_BG_COLOR, "Background: Bull", inline = "12" ↵
13   ")
14 color bearBgColorInput = input.color(RED_BG_COLOR, "Bear", inline = "12")

```

(continues on next page)

(continued from previous page)

```

14 // Plot colored signal line.
15 float signal = ta.cci(srcInput, lenInput)
16 color signalColor = color.from_gradient(signal, -200, 200, color.new(bearColorInput,_
17   ↴0), color.new(bullColorInput, 0))
18 plot(signal, "CCI", signalColor, 2)
19
20 // Detect crosses of the centerline.
21 bool signalX = ta.cross(signal, 0)
22 // Count no of bars since cross. Capping it to the no of steps from inputs.
23 int gradientStep = math.min(stepsInput, nz(ta.barssince(signalX)))
24 // Choose bull/bear end color for the gradient.
25 color endColor = signal > 0 ? bullBgColorInput : bearBgColorInput
26 // Get color from gradient going from no color to `c_endColor`
27 color bandColor = color.from_gradient(gradientStep, 0, stepsInput, na, endColor)
28 bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed)
29 bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed)
30 fill(bandTopPlotID, bandBotPlotID, bandColor, title = "Band")

```

Note that:

- The signal plot uses the same base colors and gradient as in our previous example. We have however increased the width of the line from the default 1 to 2. It is the most important component of our visuals; increasing its width is a way to give it more prominence, and ensure users are not distracted by the band, which has become busier than it was in its original, flat beige color.
- The fill must remain unobtrusive for two reasons. First, it is of secondary importance to the visuals, as it provides complementary information, i.e., the duration for which the signal has been in bull/bear territory. Second, since fills have a greater z-index than plots, the fill will cover the signal plot. For these reasons, we make the fill's base colors fairly transparent, at 70, so they do not mask the plots. The gradient used for the band starts with no color at all (see the `na` used as the argument to `bottom_color` in the `color.from_gradient()` call), and goes to the base bull/bear colors from the inputs, which the conditional, `c_endColor` color variable contains.
- We provide users with distinct bull/bear color selections for the line and the band.
- When we calculate the `gradientStep` variable, we use `nz()` on `ta.barssince()` because in early bars of the dataset, when the condition tested has not occurred yet, `ta.barssince()` will return `na`. Because we use `nz()`, the value returned is replaced with zero in those cases.

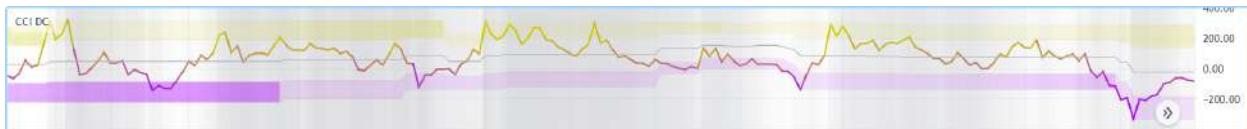
#### 4.7.5 Mixing transparencies

In this example we take our CCI indicator in another direction. We will build dynamically adjusting extremes zone buffers using a Donchian Channel (historical highs/lows) calculated from the CCI. We build the top/bottom bands by making them 1/4 the height of the DC. We will use a dynamically adjusting lookback to calculate the DC. To modulate the lookback, we will calculate a simple measure of volatility by keeping a ratio of a short-period ATR to a long one. When that ratio is higher than 50 of its last 100 values, we consider the volatility high. When the volatility is high/low, we decrease/increase the lookback.

Our aim is to provide users of our indicator with:

- The CCI line colored using a bull/bear gradient, as we illustrated in our most recent examples.
- The top and bottom bands of the Donchian Channel, filled in such a way that their color darkens as a historical high/low becomes older and older.
- A way to appreciate the state of our volatility measure, which we will do by painting the background with one color whose intensity increases when volatility increases.

This is what our indicator looks like using the light theme:



And with the dark theme:



```

1 // @version=5
2 indicator("CCI DC", precision = 6)
3 color GOLD_COLOR = #CCCC00ff
4 color VIOLET_COLOR = #AA00FFff
5 int lengthInput = input.int(20, "Length", minval = 5)
6 color bullColorInput = input.color(GOLD_COLOR, "Bull")
7 color bearColorInput = input.color(VIOLET_COLOR, "Bear")
8
9 // ----- Function clamps `val` between `min` and `max`.
10 clamp(val, min, max) =>
11     math.max(min, math.min(max, val))
12
13 // ----- Volatility expressed as 0-100 value.
14 float v = ta.atr(lengthInput / 5) / ta.atr(lengthInput * 5)
15 float vPct = ta.percentrank(v, lengthInput * 5)
16
17 // ----- Calculate dynamic lookback for DC. It increases/decreases on low/high
18 // volatility.
19 bool highVolatility = vPct > 50
20 var int lookBackMin = lengthInput * 2
21 var int lookBackMax = lengthInput * 10
22 var float lookBack = math.avg(lookBackMin, lookBackMax)
23 lookBack += highVolatility ? -2 : 2
24 lookBack := clamp(lookBack, lookBackMin, lookBackMax)
25
26 // ----- Dynamic lookback length Donchian channel of signal.
27 float signal = ta.cci(close, lengthInput)
28 // `lookBack` is a float; need to cast it to int to be used a length.
29 float hiTop = ta.highest(signal, int(lookBack))
30 float loBot = ta.lowest(signal, int(lookBack))
31 // Get margin of 25% of the DC height to build high and low bands.
32 float margin = (hiTop - loBot) / 4
33 float hiBot = hiTop - margin
34 float loTop = loBot + margin
35 // Center of DC.
36 float center = math.avg(hiTop, loBot)
37
38 // ----- Create colors.
39 color signalColor = color.from_gradient(signal, -200, 200, bearColorInput,
40 //                                         bullColorInput)
41 // Bands: Calculate transparencies so the longer since the hi/lo has changed,
42 //         the darker the color becomes. Cap highest transparency to 90.
43 float hiTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(hiTop)) +
44 //                                         1)) / 255), 60, 90)

```

(continues on next page)

(continued from previous page)

```

42 float loTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(loBot)) +_
43 ↪1)) / 255), 60, 90)
44 color hiColor = color.new(bullColorInput, hiTransp)
45 color loColor = color.new(bearColorInput, loTransp)
46 // Background: Rescale the 0-100 range of `vPct` to 0-25 to create 75-100_
47 ↪transparencies.
48 color bgColor = color.new(color.gray, 100 - (vPct / 4))
49
50 // ----- Plots
51 // Invisible lines for band fills.
52 hiTopPlotID = plot(hiTop, color = na)
53 hiBotPlotID = plot(hiBot, color = na)
54 loTopPlotID = plot(loTop, color = na)
55 loBotPlotID = plot(loBot, color = na)
56 // Plot signal and centerline.
57 p_signal = plot(signal, "CCI", signalColor, 2)
58 plot(center, "Centerline", color.silver, 1)
59
60 // Fill the bands.
61 fill(hiTopPlotID, hiBotPlotID, hiColor)
62 fill(loTopPlotID, loBotPlotID, loColor)
63
64 // ----- Background.
65 bgcolor(bgColor)

```

Note that:

- We clamp the transparency of the background to a 100-75 range so that it doesn't overwhelm. We also use a neutral color that will not distract too much. The darker the background is, the higher our measure of volatility.
- We also clamp the transparency values for the band fills between 60 and 90. We use 90 so that when a new high/low is found and the gradient resets, the starting transparency makes the color somewhat visible. We do not use a transparency lower than 60 because we don't want those bands to hide the signal line.
- We use the very handy `ta.percentrank()` function to generate a 0-100 value from our ATR ratio measuring volatility. It is useful to convert values whose scale is unknown into known values that can be used to produce transparencies.
- Because we must clamp values three times in our script, we wrote an `f_clamp()` function, instead of explicitly coding the logic three times.

## 4.7.6 Tips

### Designing usable colors schemes

If you write scripts intended for other traders, try to avoid colors that will not work well in some environments, whether it be for plots, labels, tables or fills. At a minimum, test your visuals to ensure they perform satisfactorily with both the light and dark TradingView themes; they are the most commonly used. Colors such as black and white, for example, should be avoided.

Build the appropriate inputs to provide script users the flexibility to adapt your script's visuals to their particular environments.

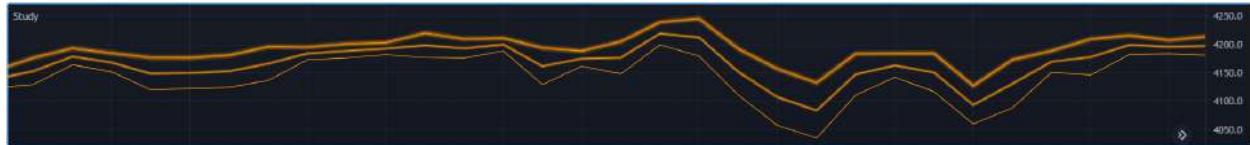
Take care to build a visual hierarchy of the colors you use that matches the relative importance of your script's visual components. Good designers understand how to achieve the optimal balance of color and weight so the eye is naturally drawn to the most important elements of the design. When you make everything stand out, nothing does. Make room for some elements to stand out by toning down the visuals surrounding it.

Providing a selection of color presets in your inputs — rather than a single color that can be changed — can help color-challenged users. Our [Technical Ratings](#) demonstrates one way of achieving this.

### Plot crisp lines

It is best to use zero transparency to plot the important lines in your visuals, to keep them crisp. This way, they will show through fills more precisely. Keep in mind that fills have a higher z-index than plots, so they are placed on top of them. A slight increase of a line's width can also go a long way in making it stand out.

If you want a special plot to stand out, you can also give it more importance by using multiple plots for the same line. These are examples where we modulate the successive width and transparency of plots to achieve this:



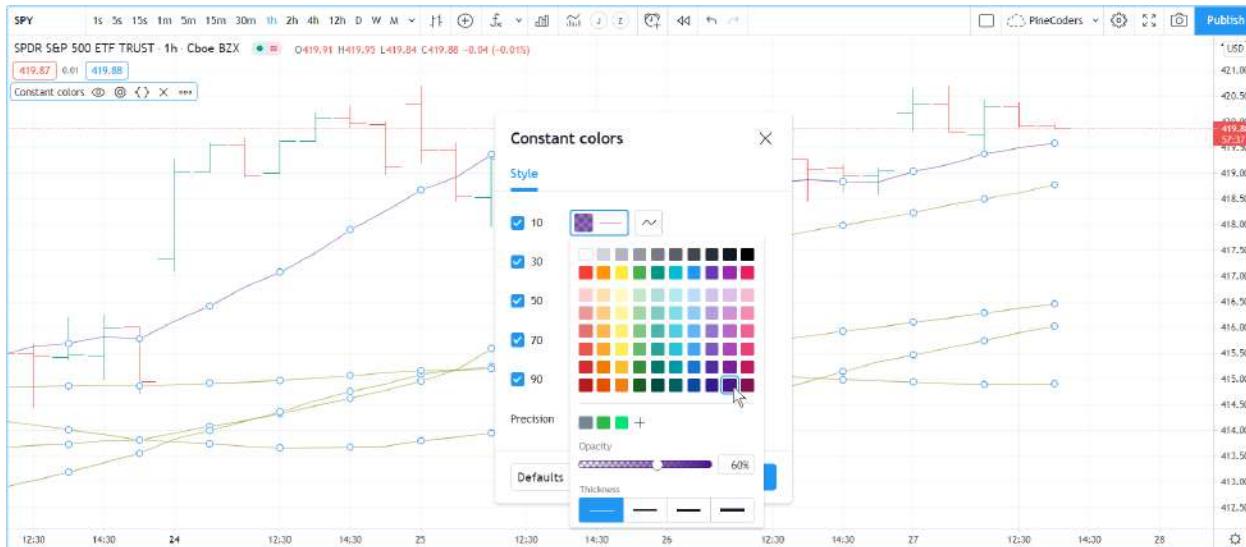
```
1 //@version=5
2 indicator("")
3 plot(high, "", color.new(color.orange, 80), 8)
4 plot(high, "", color.new(color.orange, 60), 4)
5 plot(high, "", color.new(color.orange, 00), 1)
6
7 plot(hl2, "", color.new(color.orange, 60), 4)
8 plot(hl2, "", color.new(color.orange, 00), 1)
9
10 plot(low, "", color.new(color.orange, 0), 1)
```

### Customize gradients

When building gradients, adapt them to the visuals they apply to. If you are using a gradient to color candles, for example, it is usually best to limit the number of steps in the gradient to ten or less, as it is more difficult for the eye to perceive intensity variations of discrete objects. As we did in our examples, cap minimum and maximum transparency levels so your visual elements remain visible and do not overwhelm when it's not necessary.

### Color selection through script settings

The type of color you use in your scripts has an impact on how users of your script will be able to change the colors of your script's visuals. As long as you don't use colors whose RGBA components have to be calculated at runtime, script users will be able to modify the colors you use by going to your script's "Settings/Style" tab. Our first example script on this page meets that criteria, and the following screenshot shows how we used the script's "Settings/Style" tab to change the color of the first moving average:



If your script uses a calculated color, i.e., a color where at least one of its RGBA components can only be known at runtime, then the “Settings/Style” tab will NOT offer users the usual color widgets they can use to modify your plot colors. Plots of the same script not using calculated colors will also be affected. In this script, for example, our first `plot()` call uses a calculated color, and the second one doesn’t:

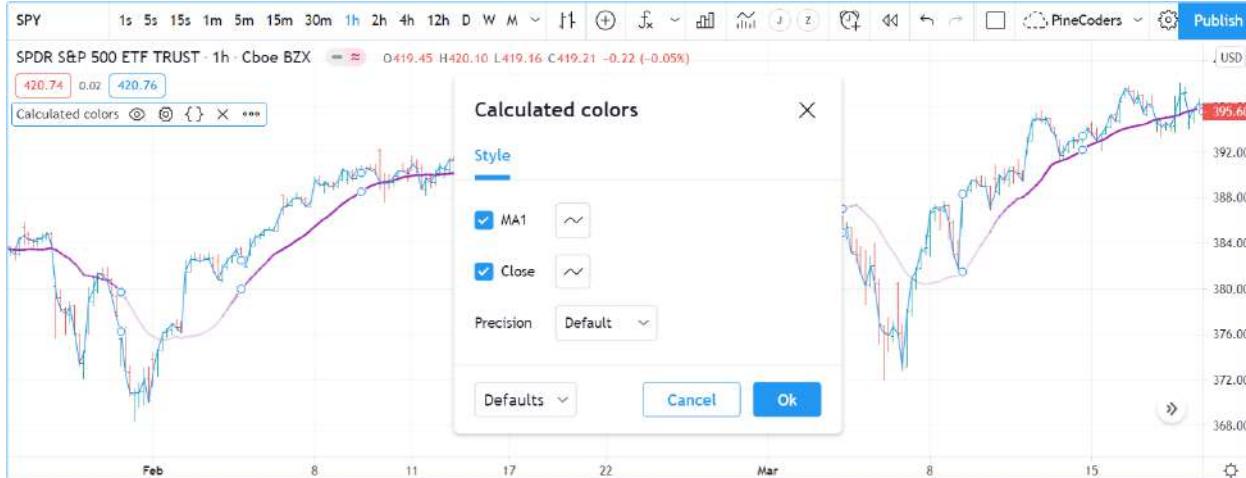
```

1 //@version=5
2 indicator("Calculated colors", "", true)
3 float ma = ta.sma(close, 20)
4 float maHeight = ta.percentrank(ma, 100)
5 float transparency = math.min(80, 100 - maHeight)
6 // This plot uses a calculated color.
7 plot(ma, "MA1", color.rgb(156, 39, 176, transparency), 2)
8 // This plot does not use a calculated color.
9 plot(close, "Close", color.blue)

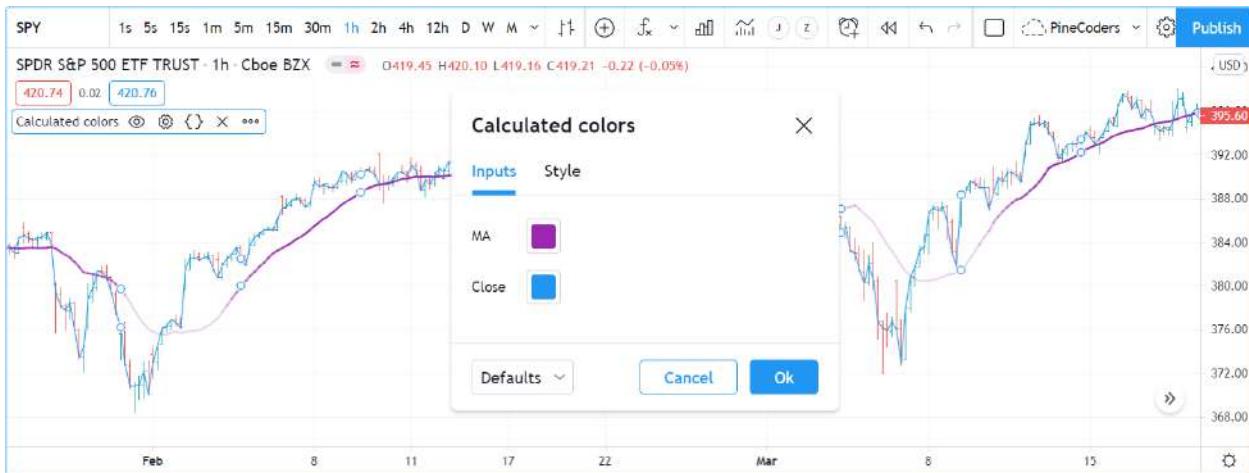
```

The color used in the first plot is a calculated color because its transparency can only be known at runtime. It is calculated using the relative position of the moving average in relation to its past 100 values. The greater percentage of past values are below the current value, the higher the 0-100 value of `maHeight` will be. Since we want the color to be the darkest when `maHeight` is 100, we subtract 100 from it to obtain the zero transparency then. We also cap the calculated transparency value to a maximum of 80 so that it always remains visible.

Because that calculated color is used in our script, the “Settings/Style” tab will not show any color widgets:



The solution to enable script users to control the colors used is to supply them with custom inputs, as we do here:



```

1 //@version=5
2 indicator("Calculated colors", "", true)
3 color maInput = input.color(color.purple, "MA")
4 color closeInput = input.color(color.blue, "Close")
5 float ma = ta.sma(close, 20)
6 float maHeight = ta.percentrank(ma, 100)
7 float transparency = math.min(80, 100 - maHeight)
8 // This plot uses a calculated color.
9 plot(ma, "MA1", color.new(maInput, transparency), 2)
10 // This plot does not use a calculated color.
11 plot(close, "Close", closeInput)

```

Notice how our script's “Settings” now show an “Inputs” tab, where we have created two color inputs. The first one uses `color.purple` as its default value. Whether the script user changes that color or not, it will then be used in a `color.new()` call to generate a calculated transparency in the `plot()` call. The second input uses as its default the built-in `color.blue` color we previously used in the `plot()` call, and simply use it as is in the second `plot()` call.



## 4.8 Fills

- *Introduction*
- `'plot()` and `hline()` fills`
- `Line fills`
- `Box and polyline fills`

## 4.8.1 Introduction

Some of Pine Script's visual outputs, including *plots*, *hlines*, *lines*, *boxes*, and *polylines*, allow one to fill the chart space they occupy with colors. Three different mechanisms facilitate filling the space between such outputs:

- The `fill()` function fills the space between two plots from `plot()` calls or two horizontal lines (`hline()`) from `hline()` calls with a specified color.
- Objects of the `linefill` type fill the space between `line` instances created with `line.new()`.
- Other drawing types, namely *boxes* and *polylines*, have built-in properties that allow the drawings to fill the visual spaces they occupy.

## 4.8.2 `plot()` and `hline()` fills

The `fill()` function fills the space between two plots or horizontal lines. It has the following two signatures:

```
fill(plot1, plot2, color, title, editable, show_last, fillgaps) → void
fill(hline1, hline2, color, title, editable, fillgaps) → void
```

The `plot1`, `plot2`, `hline1`, and `hline2` parameters accept *plot* or *hline* IDs returned by `plot()` and `hline()` function calls. The `fill()` function is the only built-in that can use these IDs.

This simple example demonstrates how the `fill()` function works with *plot* and *hline* IDs. It calls `plot()` and `hline()` three times to display arbitrary values on the chart. Each of these calls returns an ID, which the script assigns to variables for use in the `fill()` function. The values of `p1`, `p2`, and `p3` are “plot” IDs, whereas `h1`, `h2`, and `h3` reference “hline” IDs:



```
1 // @version=5
2 indicator("Example 1")
3
4 // Assign "plot" IDs to the `p1`, `p2`, and `p3` variables.
5 p1 = plot(math.sin(high), "Sine of `high`")
6 p2 = plot(math.cos(low), "Cosine of `low`")
7 p3 = plot(math.sin(close), "Sine of `close`")
8 // Fill the space between `p1` and `p2` with 90% transparent red.
9 fill(p1, p3, color.new(color.red, 90), "`p1`-'p3` fill")
10 // Fill the space between `p2` and `p3` with 90% transparent blue.
```

(continues on next page)

(continued from previous page)

```

11 fill(p2, p3, color.new(color.blue, 90), "`p2`-`p3` fill")
12
13 // Assign "hline" IDs to the `h1`, `h2`, and `h3` variables.
14 h1 = hline(0, "First level")
15 h2 = hline(1.0, "Second level")
16 h3 = hline(0.5, "Third level")
17 h4 = hline(1.5, "Fourth level")
18 // Fill the space between `h1` and `h2` with 90% transparent yellow.
19 fill(h1, h2, color.new(color.yellow, 90), "`h1`-`h2` fill")
20 // Fill the space between `h3` and `h4` with 90% transparent lime.
21 fill(h3, h4, color.new(color.lime, 90), "`h3`-`h4` fill")

```

It's important to note that the `fill()` function requires *either* two "plot" IDs or two "hline" IDs. One *cannot* mix and match these types in the function call. Consequently, programmers will sometimes need to use `plot()` where they otherwise might have used `hline()` if they want to fill the space between a consistent level and a fluctuating series.

For example, this script calculates an oscillator based on the percentage distance between the chart's `close` price and a 10-bar SMA, then plots it on the chart pane. In this case, we wanted to fill the area between the oscillator and zero. Although we can display the zero level with `hline()` since its value does not change, we cannot pass a "plot" and "hline" ID to the `fill()` function. Therefore, we must use a `plot()` call for the level to allow the script to fill the space:



```

1 // @version=5
2 indicator("Example 2")
3
4 // @variable The 10-bar moving average of `close` prices.
5 float ma = ta.sma(close, 10)
6 // @variable The distance from the `ma` to the `close` price, as a percentage of the
7 // `ma`.
8 float oscillator = 100 * (ma - close) / ma
9
10 // @variable The ID of the `oscillator` plot for use in the `fill()` function.
11 oscPlotID = plot(oscillator, "Oscillator")
12 // @variable The ID of the zero level plot for use in the `fill()` function.
13 // Requires a "plot" ID since the `fill()` function can't use "plot" and
14 // "hline" IDs at the same time.
15 zeroPlotID = plot(0, "Zero level", color.silver, 1, plot.style_circles)

```

(continues on next page)

(continued from previous page)

```

14
15 // Fill the space between the `oscPlotID` and `zeroPlotID` with 90% transparent blue.
16 fill(oscPlotID, zeroPlotID, color.new(color.blue, 90), "Oscillator fill")

```

The `color` parameter of the `fill()` function accepts a “series color” argument, meaning the fill’s color can change across chart bars. For example, this code fills the space between two moving average plots with 90% transparent green or red colors based on whether `ma1` is above `ma2`:



```

1 // @version=5
2 indicator("Example 3", overlay = true)
3
4 // @variable The 5-bar moving average of `close` prices.
5 float ma1 = ta.sma(close, 5)
6 // @variable The 20-bar moving average of `close` prices.
7 float ma2 = ta.sma(close, 20)
8
9 // @variable The 90% transparent color of the space between MA plots. Green if `ma1 > ma2`, red otherwise.
10 color fillColor = ma1 > ma2 ? color.new(color.green, 90) : color.new(color.red, 90)
11
12 // @variable The ID of the `ma1` plot for use in the `fill()` function.
13 ma1PlotID = plot(ma1, "5-bar SMA")
14 // @variable The ID of the `ma2` plot for use in the `fill()` function.
15 ma2PlotID = plot(ma2, "20-bar SMA")
16
17 // Fill the space between the `ma1PlotID` and `ma2PlotID` using the `fillColor`.
18 fill(ma1PlotID, ma2PlotID, fillColor, "SMA plot fill")

```

### 4.8.3 Line fills

While the `fill()` function allows a script to fill the space between two *plots or hlines*, it does not work with `line` objects. When a script needs to fill the space between `lines`, it requires a `linefill` object created by the `linefill.new()` function. The function has the following signature:

```
linefill.new(line1, line2, color) → series linefill
```

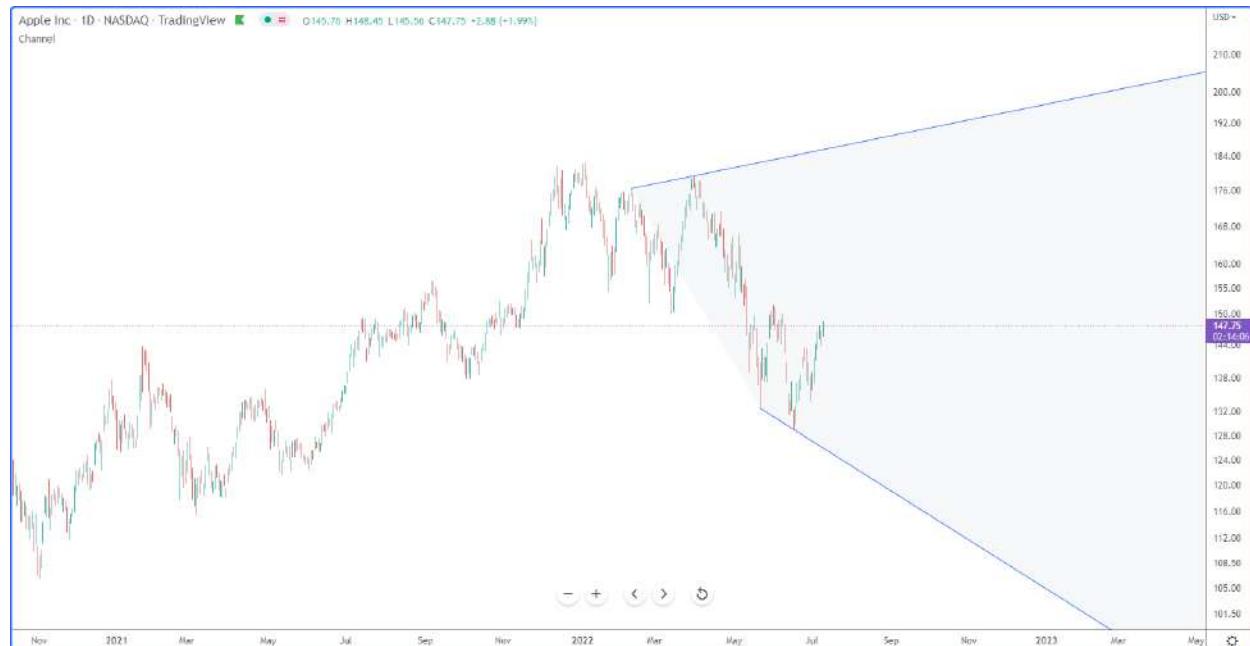
The `line1` and `line2` parameters accept `line` IDs. These IDs determine the chart region that the `linefill` object will fill with its specified `color`. A script can update the `color` property of a `linefill` ID returned by this function by calling `linefill.set_color()` with the ID as its `id` argument.

The behavior of linefills depends on the lines they reference. Scripts cannot move linefills directly, as the lines that a linefill uses determine the space it will fill. To retrieve the IDs of the `lines` referenced by a `linefill` object, use the `linefill.get_line1()` and `linefill.get_line2()` functions.

Any pair of `line` instances can only have *one* `linefill` between them. Successive calls to `linefill.new()` using the same `line1` and `line2` arguments will create a new `linefill` ID that *replaces* the previous one associated with them.

The example below demonstrates a simple use case for linefills. The script calculates a `pivotHigh` and `pivotLow` series using the built-in `ta.pivothigh()` and `ta.pivotlow()` functions with constant `leftbars` and `rightbars` arguments. On the last confirmed historical bar, the script draws two extended lines. The first line connects the two most recent non-na `pivotHigh` values, and the second connects the most recent non-na `pivotLow` values.

To emphasize the “channel” formed by these lines, the script fills the space between them using `linefill.new()`:



```

1 // @version=5
2 indicator("Linefill demo", "Channel", true)
3
4 // @variable The number bars to the left of a detected pivot.
5 int LEFT_BARS = 15
6 // @variable The number bars to the right for pivot confirmation.
7 int RIGHT_BARS = 5
8
9 // @variable The price of the pivot high point.

```

(continues on next page)

(continued from previous page)

```

10 float pivotHigh = ta.pivothigh(LEFT_BARS, RIGHT_BARS)
11 // @variable The price of the pivot low point.
12 float pivotLow = ta.pivotlow(LEFT_BARS, RIGHT_BARS)
13
14 // Initialize the chart points the lines will use.
15 var firstHighPoint = chart.point.new(na, na, na)
16 var secondHighPoint = chart.point.new(na, na, na)
17 var firstLowPoint = chart.point.new(na, na, na)
18 var secondLowPoint = chart.point.new(na, na, na)
19
20 // Update the `firstHighPoint` and `secondHighPoint` when `pivotHigh` is not `na`.
21 if not na(pivotHigh)
22     firstHighPoint := secondHighPoint
23     secondHighPoint := chart.point.from_index(bar_index - RIGHT_BARS, pivotHigh)
24 // Update the `firstLowPoint` and `secondLowPoint` when `pivotlow` is not `na`.
25 if not na(pivotLow)
26     firstLowPoint := secondLowPoint
27     secondLowPoint := chart.point.from_index(bar_index - RIGHT_BARS, pivotLow)
28
29 if barstate.islastconfirmedhistory
30     // @variable An extended line that passes through the `firstHighPoint` and_
31     // `secondHighPoint`.
32     line pivotHighLine = line.new(firstHighPoint, secondHighPoint, extend = extend.
33     // right)
34     // @variable An extended line that passes through the `firstLowPoint` and_
35     // `secondLowPoint`.
36     line pivotLowLine = line.new(firstLowPoint, secondLowPoint, extend = extend.right)
37     // @variable The color of the space between the lines.
38     color fillColor = switch
39         secondHighPoint.price > firstHighPoint.price and secondLowPoint.price >_
40         // firstLowPoint.price => color.lime
41         secondHighPoint.price < firstHighPoint.price and secondLowPoint.price <_
42         // firstLowPoint.price => color.red
43         =>
44             color.silver
45         // @variable A linefill that colors the space between the `pivotHighLine` and_
46         // `pivotLowLine`.
47         linefill channelFill = linefill.new(pivotHighLine, pivotLowLine, color.
48         new(fillColor, 90))

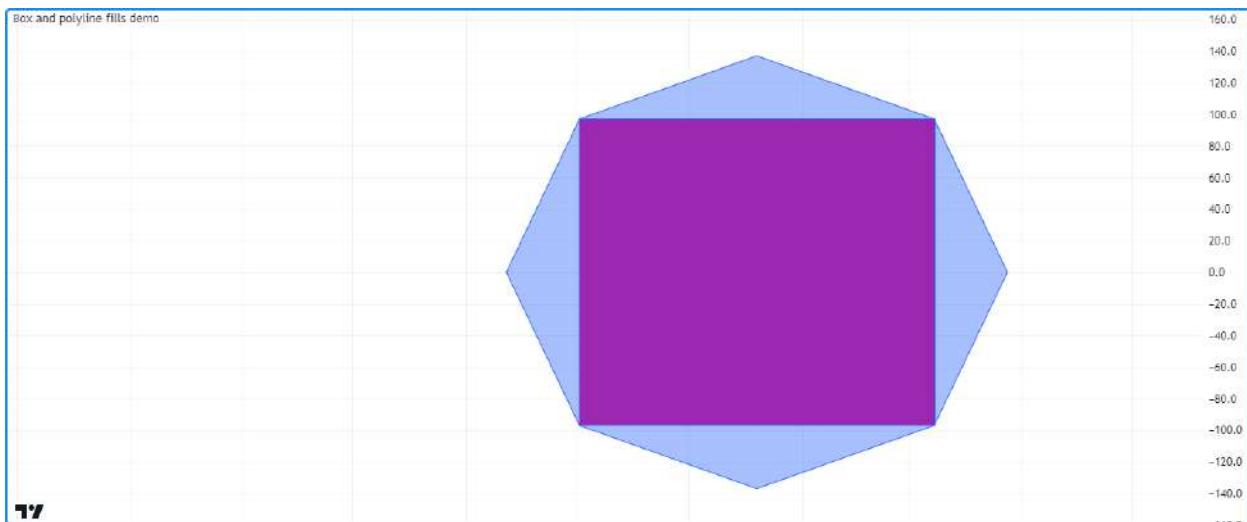
```

#### 4.8.4 Box and polyline fills

The `box` and `polyline` types allow scripts to draw geometric shapes and other formations on the chart. Scripts create `boxes` and `polylines` with the `box.new()` and `polyline.new()` functions, which include parameters that allow the drawings to fill their visual spaces.

To fill the space inside the borders of a `box` with a specified color, include a `bgcolor` argument in the `box.new()` function. To fill a polyline's visual space, pass a `fill_color` argument to the `polyline.new()` function.

For example, this script draws an octagon with a `polyline` and an inscribed rectangle with a `box` on the last confirmed historical bar. It determines the size of the drawings using the value from the `radius` variable, which corresponds to approximately one-fourth of the number of bars visible on the chart. We included `fill_color = color.new(color.blue, 60)` in the `polyline.new()` call to fill the octagon with a translucent blue color, and we used `bgcolor = color.purple` in the `box.new()` call to fill the inscribed rectangle with opaque purple:



```

1 // @version=5
2 indicator("Box and polyline fills demo")
3
4 //@variable The number of visible chart bars, excluding the leftmost and rightmost_
5 // bars.
6 var int barCount = 0
7 if time > chart.left_visible_bar_time and time < chart.right_visible_bar_time
8     barCount += 1
9
10 //@variable The approximate radius used to calculate the octagon and rectangle_
11 // coordinates.
12 int radius = math.ceil(barCount / 4)
13
14 if barstate.islastconfirmedhistory
15     //@variable An array of chart points. The polyline uses all points in this array,_
16 // but the box only needs two.
17     array<chart.point> points = array.new<chart.point>()
18     //@variable The counterclockwise angle of each point, in radians. Updates on each_
19 // loop iteration.
20     float angle = 0.0
21     //@variable The radians to add to the `angle` on each loop iteration.
22     float increment = 0.25 * math.pi
23     // Loop 8 times to calculate octagonal points.
24     for i = 0 to 7
25         //@variable The point's x-coordinate (bar offset).
26         int x = int(math.round(math.cos(angle) * radius))
27         //@variable The point's y-coordinate.
28         float y = math.round(math.sin(angle) * radius)
29         // Push a new chart point into the `points` array and increase the `angle`.
30         points.push(chart.point.from_index(bar_index - radius + x, y))
31         angle += increment
32
33     // Create a closed polyline to draw the octagon and fill it with translucent blue.
34     polyline.new(points, closed = true, fill_color = color.new(color.blue, 60))
35     // Create a box for the rectangle using index 3 and 7 for the top-left and bottom-
36 // right corners,
37     // and fill it with opaque purple.
38     box.new(points.get(3), points.get(7), bgcolor = color.purple)

```

See this manual's [Lines and boxes](#) page to learn more about working with these types.



## 4.9 Inputs

- *Introduction*
- *Input functions*
- *Input function parameters*
- *Input types*
- *Other features affecting Inputs*
- *Tips*

### 4.9.1 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:

```

1 //@version=5
2 indicator("MA", "", true)
3 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:

```

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

```

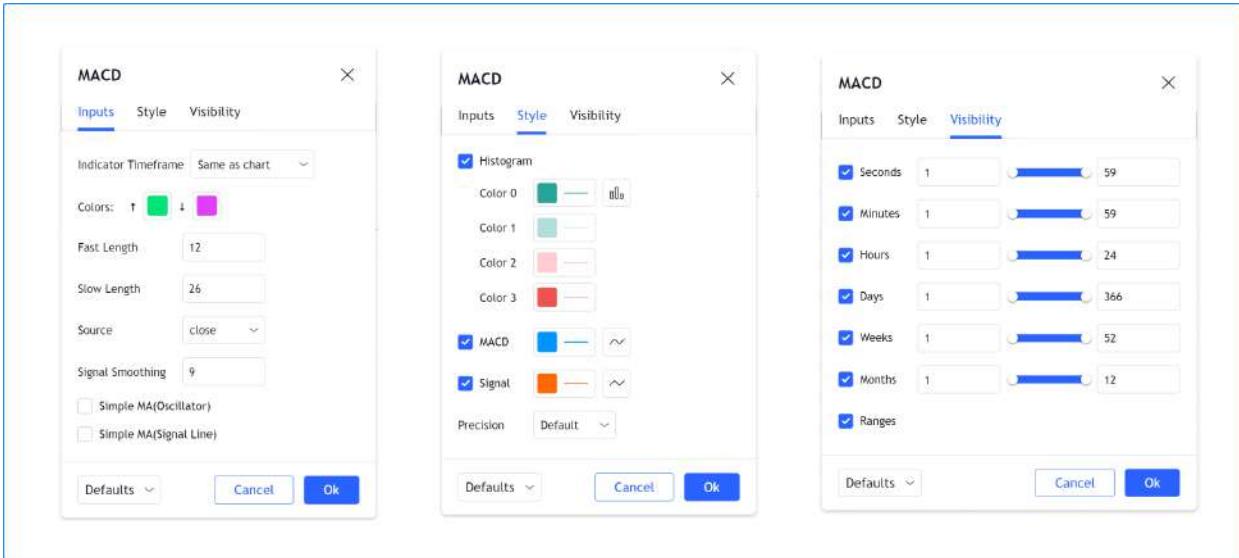
Inputs can only be accessed when a script is running on the chart. Script users access them through the script's “Settings” dialog box, which can be reached by either:

- 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.

### 4.9.2 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:

```

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

```

All values returned by `input.*()` functions except “source” ones are “input” qualified values. See our User Manual’s section on [type qualifiers](#) for more information.

### 4.9.3 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 “const” arguments (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 an `input.*()` function is always qualified as “input” or “series”, it follows that the result of one `input.*()` function call cannot be used as an argument in a subsequent `input.*()` call because the “input” qualifier is stronger than “const”.

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.

#### 4.9.4 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 | ↴  
→series float
```

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 qualified type returned by the function when used with `defval` arguments of different types:

```
1 // @version=5  
2 indicator(`input()", "", true)  
3 a = input(1, "input int")  
4 b = input(1.0, "input float")  
5 c = input(true, "input bool")  
6 d = input(color.orange, "input color")  
7 e = input("1", "input string")  
8 f = input(close, "series float")  
9 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 int  
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:

```

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

```

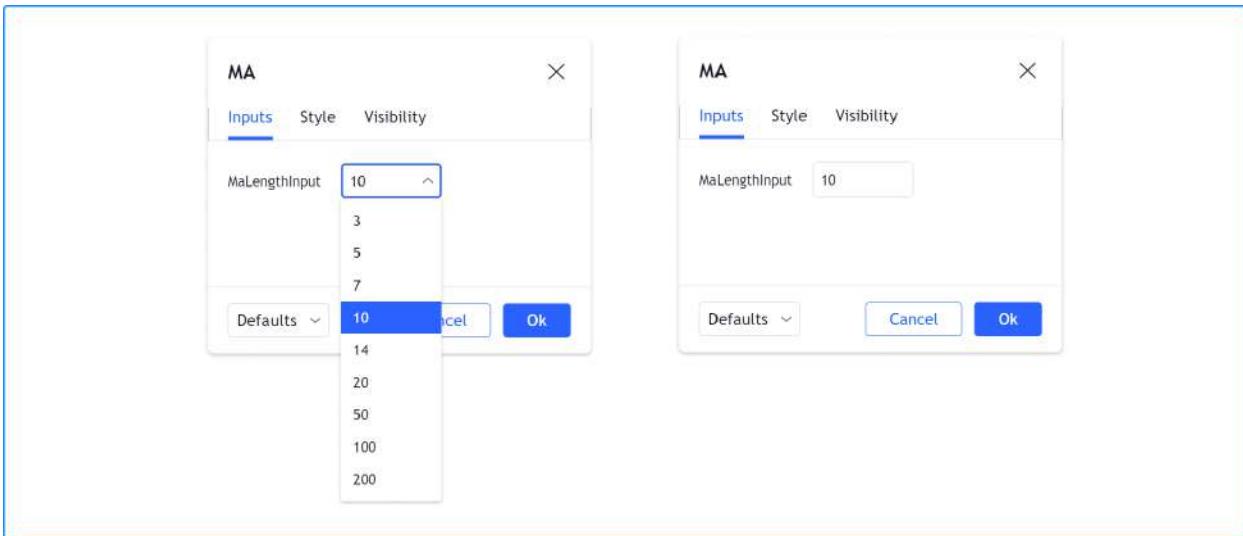
This one uses the `minval` parameter to limit the length:

```

1 //@version=5
2 indicator("MA", "", true)
3 maLengthInput = input.int(10, minval = 2)
4 ma = ta.sma(close, maLengthInput)
5 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.float(defval, title, minval, maxval, step, tooltip, inline, group, confirm) →
input int
input.float(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:

```

1 //@version=5
2 indicator("MA", "", true)
3 maLengthInput = input.int(10, minval = 1)
4 bbFactorInput = input.float(1.5, minval = 0, step = 0.5)
5 ma      = ta.sma(close, maLengthInput)
6 bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
7 bbHi   = ma + bbWidth
8 bbLo   = ma - bbWidth
9 plot(ma)

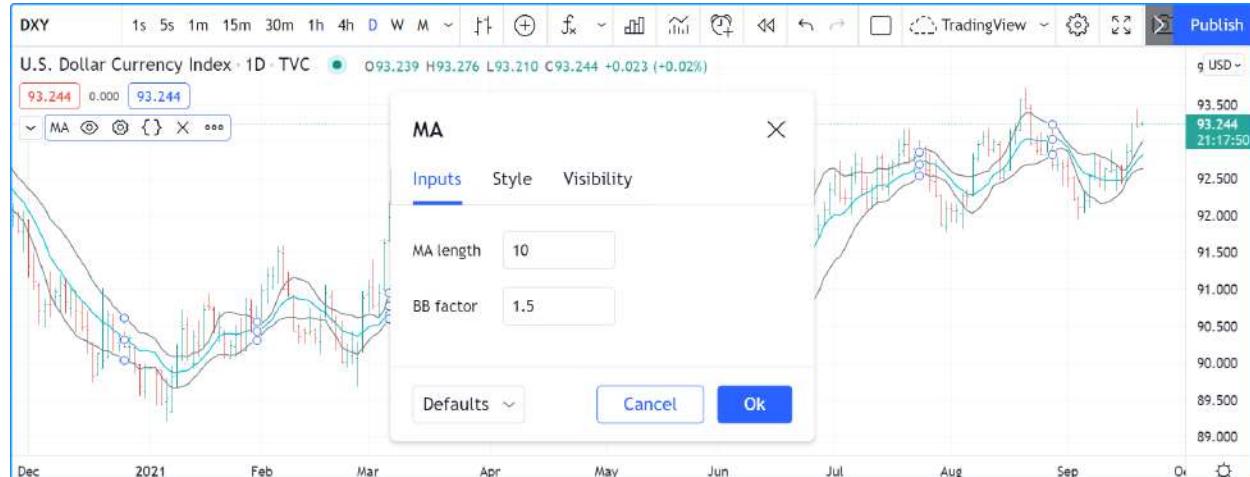
```

(continues on next page)

(continued from previous page)

```
10 plot(bbHi, "BB Hi", color.gray)
11 plot(bbLo, "BB Lo", color.gray)
```

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



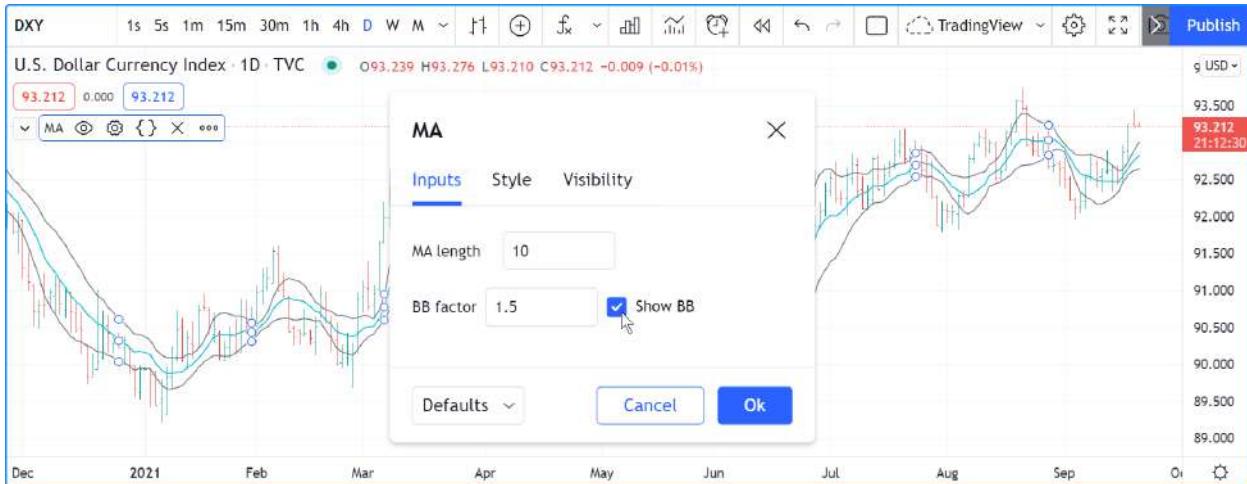
### Boolean input

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:

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

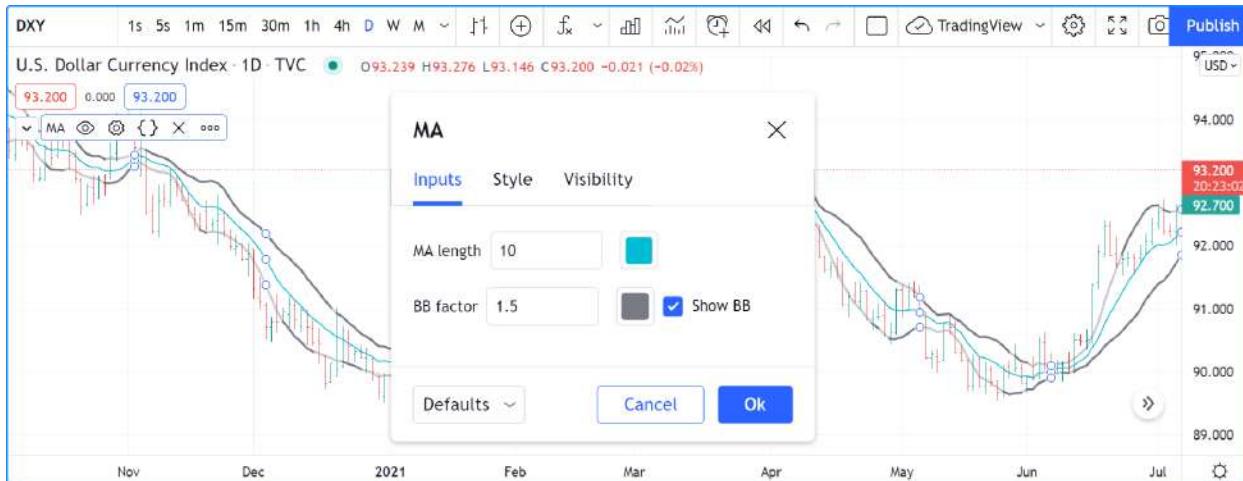
```
1 //@version=5
2 indicator("MA", "", true)
3 maLengthInput = input.int(10, "MA length", inline = "01", minval = 1)
4 maColorInput = input.color(color.aqua, "", inline = "01")
5 bbFactorInput = input.float(1.5, "BB factor", inline = "02", minval = 0, step_
6 ↗= 0.5)
7 bbColorInput = input.color(color.gray, "", inline = "02")
8 showBBInput = input.bool(true, "Show BB", inline = "02")
9 ma = ta.sma(close, maLengthInput)
10 bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
11 bbHi = ma + bbWidth
12 bbLo = ma - bbWidth
13 bbHiColor = color.new(bbColorInput, high > bbHi ? 60 : 0)
14 bbLoColor = color.new(bbColorInput, low < bbLo ? 60 : 0)
15 plot(ma, "MA", maColorInput)
16 plot(showBBInput ? bbHi : na, "BB Hi", bbHiColor, 2)
17 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:

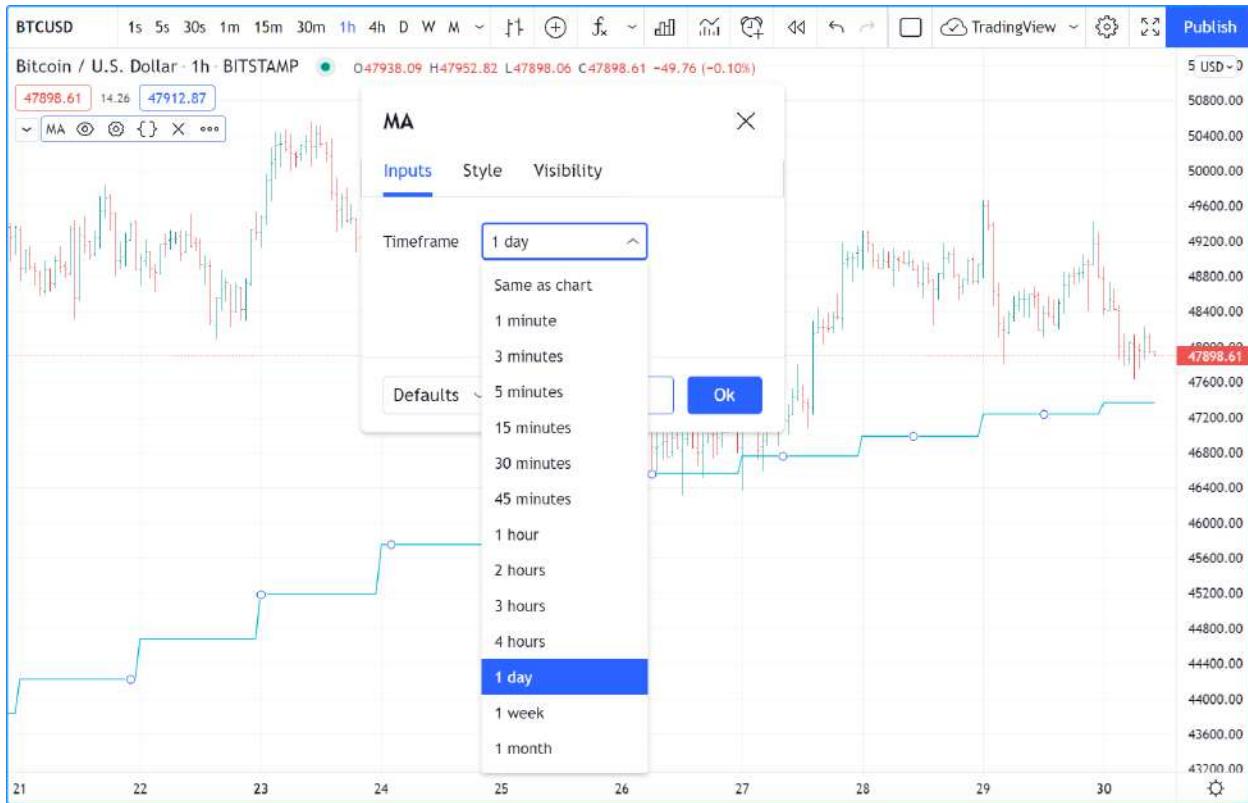
```

1 //@version=5
2 indicator("MA", "", true)
3 tfInput = input.timeframe("D", "Timeframe")
4 ma = ta.sma(close, 20)
5 securityNoRepaint(sym, tf, src) =>
6     request.security(sym, tf, src[barstate.isrealtime ? 1 : 0]) [barstate.isrealtime ?_
→0 : 1]
7 maHTF = securityNoRepaint(syminfo.tickerid, tfInput, ma)
8 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 favorited 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:

```

1 //@version=5
2 indicator("MA", "", true)
3 tfInput = input.timeframe("D", "Timeframe")
4 symbolInput = input.symbol("", "Symbol")
5 ma = ta.sma(close, 20)
6 securityNoRepaint(sym, tf, src) =>
7     request.security(sym, tf, src[barstate.isrealtime ? 1 : 0]) [barstate.isrealtime ?_
8     ↪ 0 : 1]
9 maHTF = securityNoRepaint(symbolInput, tfInput, ma)
10 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.

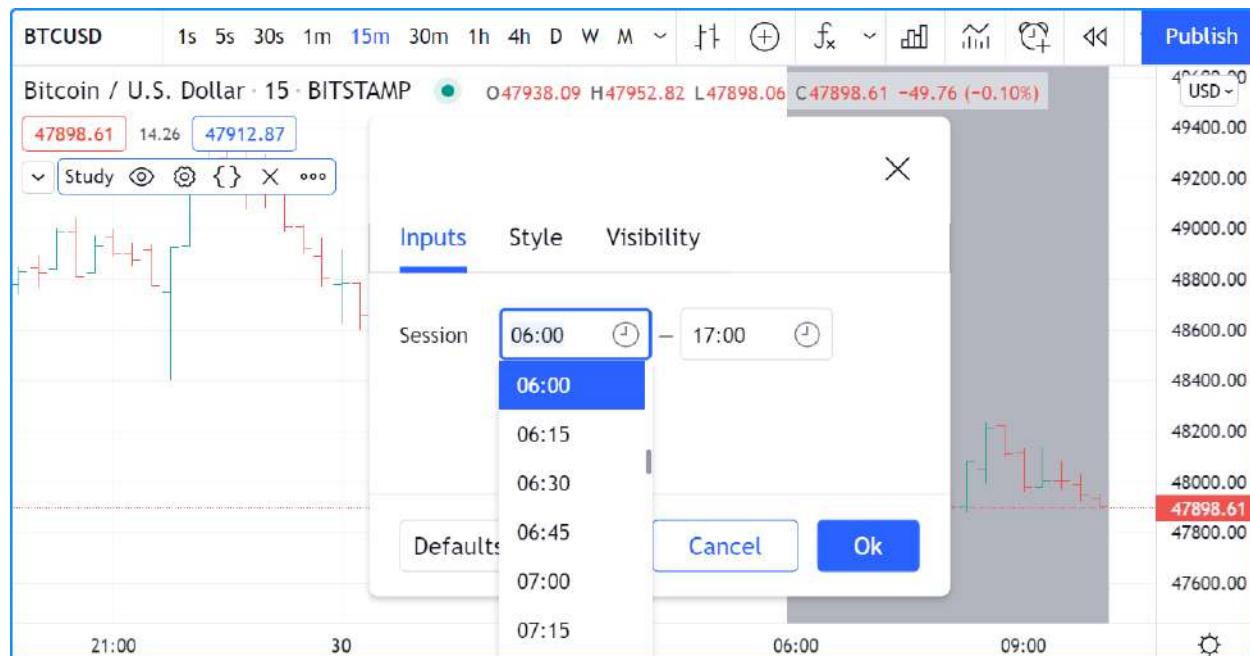
The value returned by `input.session()` is a valid string in session format. See the manual’s page on [sessions](#) for more information.

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

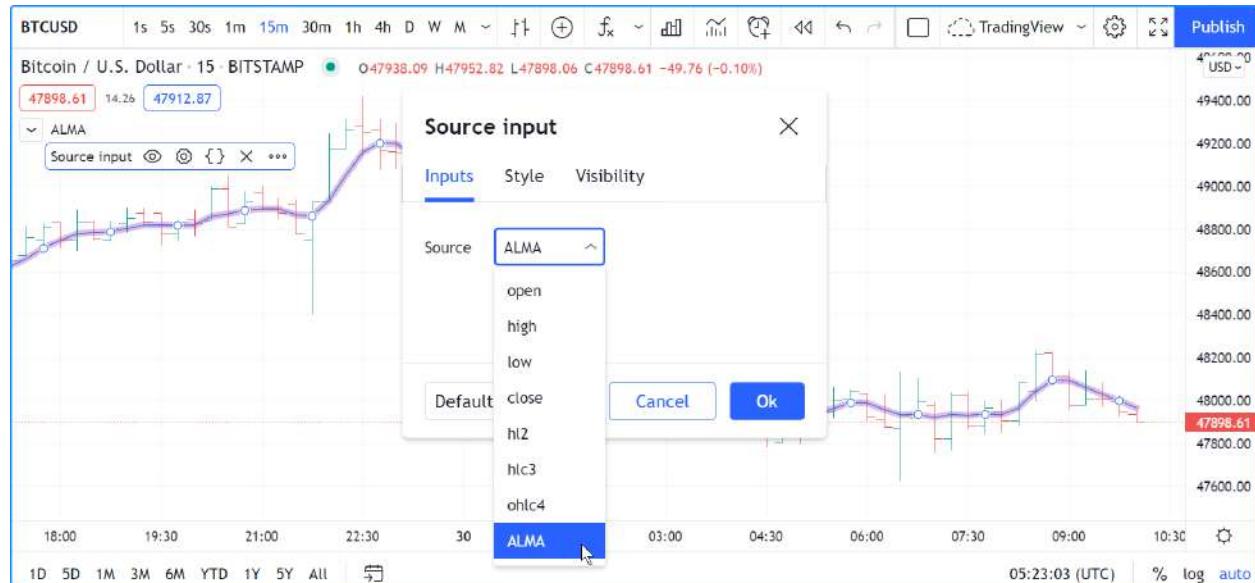
Source inputs are useful to provide a selection of two types of sources:

- 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 or more 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:

```
1 //@version=5
2 indicator("Source input", "", true)
3 srcInput = input.source(high, "Source")
4 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:

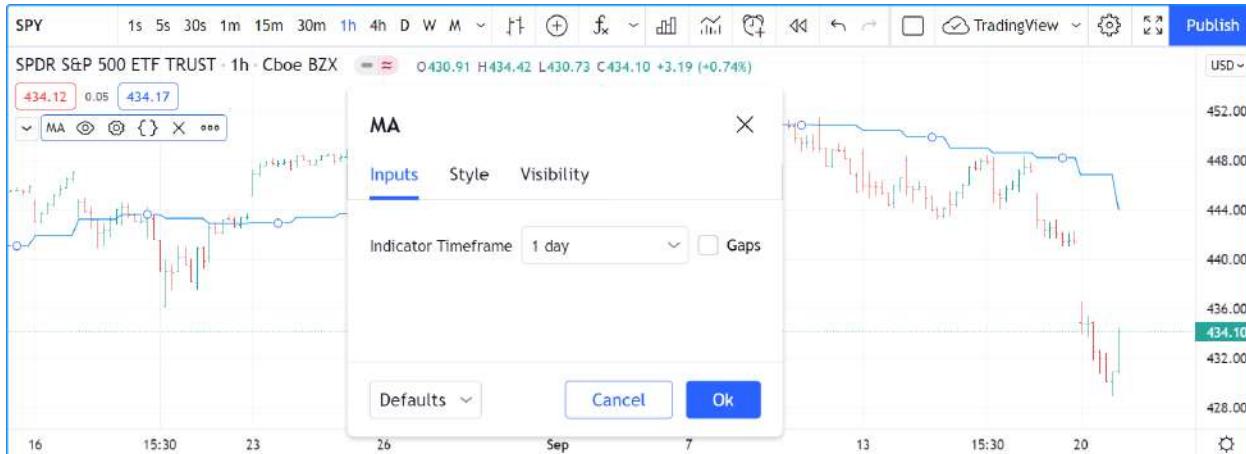
```
1 //@version=5
2 indicator("Time input", "T", true)
3 timeAndDateInput = input.time(timestamp("1 Aug 2021 00:00 +0300"), "Date and time")
4 barIsLater = time > timeAndDateInput
5 plotchar(barIsLater, "barIsLater", "↑", location.top, size = size.tiny)
```

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

## 4.9.5 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:

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



## 4.9.6 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 multiple `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:

```
1 //@version=5
2 indicator("Aligned inputs", "", true)
3
4 var GRP1 = "Not aligned"
```

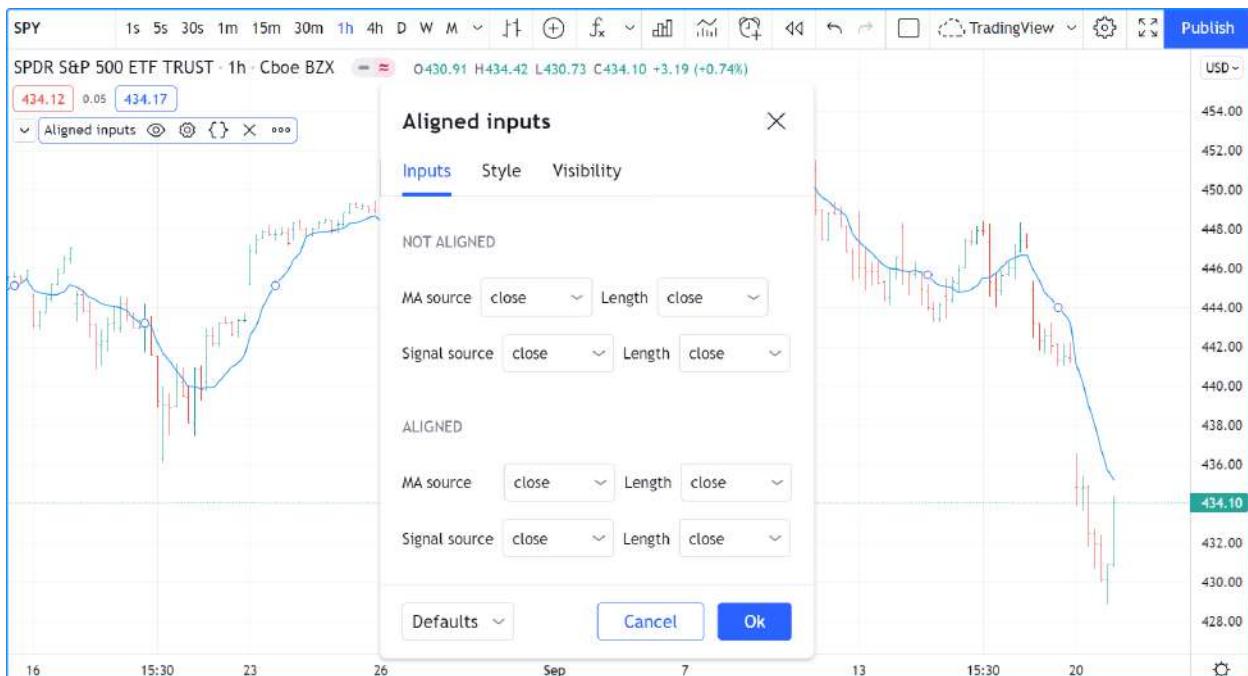
(continues on next page)

(continued from previous page)

```

5 ma1SourceInput = input(close, "MA source", inline = "11", group = GRP1)
6 ma1LengthInput = input(close, "Length", inline = "11", group = GRP1)
7 long1SourceInput = input(close, "Signal source", inline = "12", group = GRP1)
8 long1LengthInput = input(close, "Length", inline = "12", group = GRP1)
9
10 var GRP2 = "Aligned"
11 // The three spaces after "MA source" are Unicode EN spaces (U+2002).
12 ma2SourceInput = input(close, "MA source    ", inline = "21", group = GRP2)
13 ma2LengthInput = input(close, "Length", inline = "21", group = GRP2)
14 long2SourceInput = input(close, "Signal source", inline = "22", group = GRP2)
15 long2LengthInput = input(close, "Length", inline = "22", group = GRP2)
16
17 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.

 TradingView



## 4.10 Levels

- `'hline(`levels`
- `Fills between levels`

### 4.10.1 `hline(` levels

Levels are lines plotted using the `hline()` function. It is designed to plot **horizontal** levels using a **single color**, i.e., it does not change on different bars. See the [Levels](#) section of the page on `plot()` for alternative ways to plot levels when `hline()` won't do what you need.

The function has the following signature:

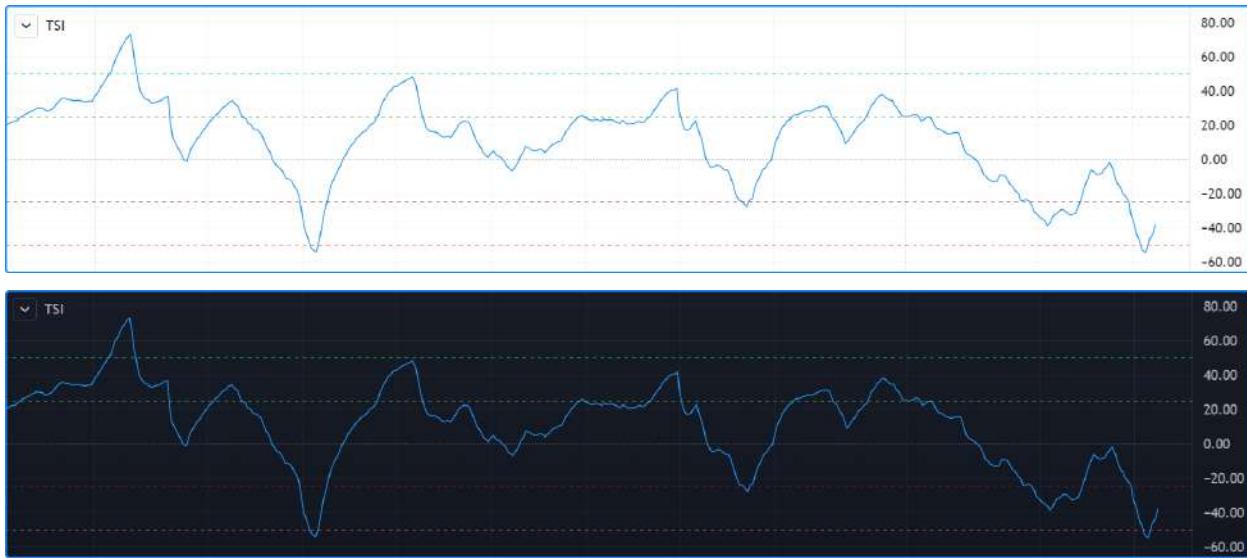
```
hline(price, title, color, linestyle, linewidth, editable) → hline
```

`hline()` has a few constraints when compared to `plot()`:

- Since the function's objective is to plot horizontal lines, its `price` parameter requires an “input int/float” argument, which means that “series float” values such as `close` or dynamically-calculated values cannot be used.
- Its `color` parameter requires an “input int” argument, which precludes the use of dynamic colors, i.e., colors calculated on each bar — or “series color” values.
- Three different line styles are supported through the `linestyle` parameter: `hline.style_solid`, `hline.style_dotted` and `hline.style_dashed`.

Let's see `hline()` in action in the “True Strength Index” indicator:

```
1 //@version=5
2 indicator("TSI")
3 myTSI = 100 * ta.tsi(close, 25, 13)
4
5 hline( 50, "+50", color.lime)
6 hline( 25, "+25", color.green)
7 hline( 0, "Zero", color.gray, linestyle = hline.style_dotted)
8 hline(-25, "-25", color.maroon)
9 hline(-50, "-50", color.red)
10
11 plot(myTSI)
```



Note that:

- We display 5 levels, each of a different color.
- We use a different line style for the zero centerline.
- We choose colors that will work well on both light and dark themes.
- The usual range for the indicator's values is +100 to -100. Since the `ta.tsi()` built-in returns values in the +1 to -1 range, we make the adjustment in our code.

#### 4.10.2 Fills between levels

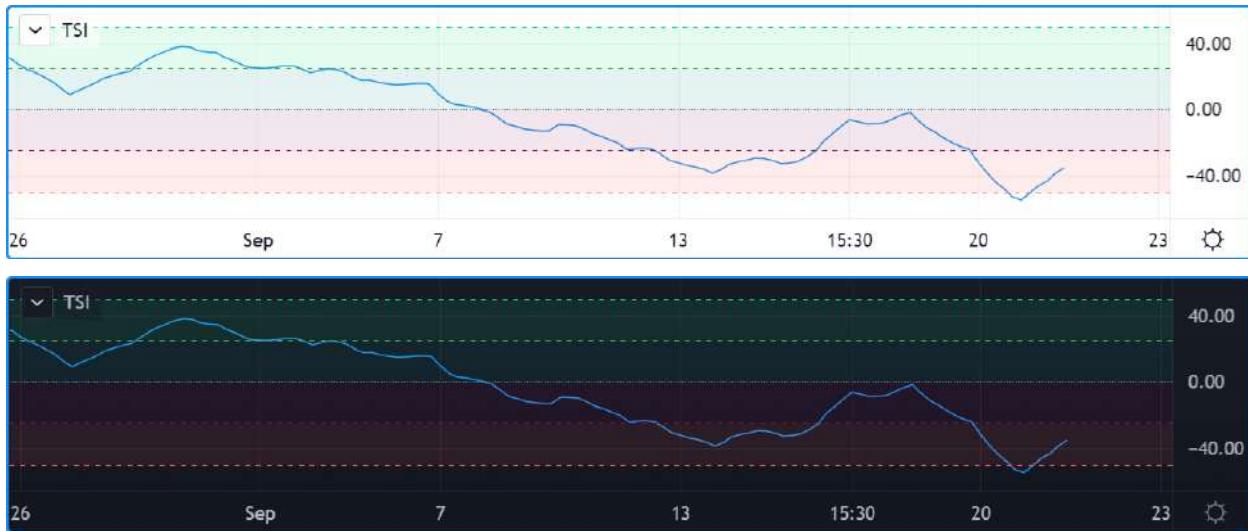
The space between two levels plotted with `hline()` can be colored using `fill()`. Keep in mind that **both** plots must have been plotted with `hline()`.

Let's put some background colors in our TSI indicator:

```

1 //@version=5
2 indicator("TSI")
3 myTSI = 100 * ta.tsi(close, 25, 13)
4
5 plus50Hline  = hline( 50, "+50", color.lime)
6 plus25Hline  = hline( 25, "+25", color.green)
7 zeroHline    = hline( 0, "Zero", color.gray, linestyle = hline.style_dotted)
8 minus25Hline = hline(-25, "-25", color.maroon)
9 minus50Hline = hline(-50, "-50", color.red)
10
11 // ----- Function returns a color in a light shade for use as a background.
12 fillColor(color col) =>
13     color.new(col, 90)
14
15 fill(plus50Hline, plus25Hline, fillColor(color.lime))
16 fill(plus25Hline, zeroHline,   fillColor(color.teal))
17 fill(zeroHline,   minus25Hline, fillColor(color.maroon))
18 fill(minus25Hline, minus50Hline, fillColor(color.red))
19
20 plot(myTSI)

```



Note that:

- We have now used the return value of our `hline()` function calls, which is of the `hline` special type. We use the `plus50Hline`, `plus25Hline`, `zeroHline`, `minus25Hline` and `minus50Hline` variables to store those “hline” IDs because we will need them in our `fill()` calls later.
- To generate lighter color shades for the background colors, we declare a `fillColor()` function that accepts a color and returns its 90 transparency. We use calls to that function for the `color` arguments in our `fill()` calls.
- We make our `fill()` calls for each of the four different fills we want, between four different pairs of levels.
- We use `color.teal` in our second fill because it produces a green that fits the color scheme better than the `color.green` used for the 25 level.



## 4.11 Libraries

- *Introduction*
- *Creating a library*
- *Publishing a library*
- *Using a library*

## 4.11.1 Introduction

Pine Script™ libraries are publications containing functions that can be reused in indicators, strategies, or in other libraries. They are useful to define frequently-used functions so their source code does not have to be included in every script where they are needed.

A library must be published (privately or publicly) before it can be used in another script. All libraries are published open-source. Public scripts can only use public libraries and they must be open-source. Private scripts or personal scripts saved in the Pine Script™ Editor can use public or private libraries. A library can use other libraries, or even previous versions of itself.

Library programmers should be familiar with Pine Script™'s typing nomenclature, scopes and user-defined functions. If you need to brush up on qualified types, see the User Manual's page on the [Type system](#). For more information on user-defined functions and scopes, see the [User-defined functions](#) page.

You can browse the library scripts published publicly by members in TradingView's [Community Scripts](#).

## 4.11.2 Creating a library

A library is a special kind of script that begins with the `library()` declaration statement, rather than `indicator()` or `strategy()`. A library contains exportable function definitions, which constitute the only visible part of the library when it is used by another script. Libraries can also use other Pine Script™ code in their global scope, like a normal indicator. This code will typically serve to demonstrate how to use the library's functions.

A library script has the following structure, where one or more exportable functions must be defined:

```

1 // @version=5
2
3 // @description <library_description>
4 library(title, overlay)
5
6 <script_code>
7
8 // @function <function_description>
9 // @param <parameter> <parameter_description>
10 // @returns <return_value_description>
11 export <function_name>([simple/series] <parameter_type> <parameter_name> [= <default_
12   ↴value>] [, ...]) =>
13     <function_code>
14 <script_code>
```

Note that:

- The `// @description`, `// @function`, `// @param` and `// @returns` *compiler annotations* are optional but we highly recommend you use them. They serve a double purpose: document the library's code and populate the default library description which authors can use when publishing the library.
- The `export` keyword is mandatory.
- `<parameter_type>` is mandatory, contrary to user-defined function parameter definitions in indicators or strategies, which are typeless.
- `<script_code>` can be any code you would normally use in an indicator, including inputs or plots.

This is an example library:

```
1 // @version=5
2
3 // @description Provides functions calculating the all-time high/low of values.
4 library("AllTimeHighLow", true)
5
6 // @function Calculates the all-time high of a series.
7 // @param val Series to use (`high` is used if no argument is supplied).
8 // @returns The all-time high for the series.
9 export hi(float val = high) =>
10    var float ath = val
11    ath := math.max(ath, val)
12
13 // @function Calculates the all-time low of a series.
14 // @param val Series to use (`low` is used if no argument is supplied).
15 // @returns The all-time low for the series.
16 export lo(float val = low) =>
17    var float atl = val
18    atl := math.min(atl, val)
19
20 plot(hi())
21 plot(lo())
```

### Library functions

Function definitions in libraries are slightly different than those of user-defined functions in indicators and strategies. There are constraints as to what can be included in the body of library functions.

In library function signatures (their first line):

- The `export` keyword is mandatory.
- The type of argument expected for each parameter must be explicitly mentioned.
- A `simple` or `series` keyword can restrict the allowable qualified types of arguments (the next section explains their use).

These are the constraints imposed on library functions:

- They cannot use variables from the library's global scope unless they are qualified as "const". This means you cannot use global variables initialized from script inputs, for example, or globally declared arrays.
- `request.*()` calls are not allowed.
- `input.*()` calls are not allowed.
- `plot*(), fill()` and `bcolor()` calls are not allowed.

Library functions always return a result that is qualified as "simple" or "series". You cannot use them where "const" or "input" qualified values are required, as is the case with some built-in functions. For example, a library function cannot be used to calculate an argument for the `show_last` parameter in a `plot()` call because it requires an "input int" value.

## Qualified type control

The qualified types of arguments supplied in calls to library functions are autodetected based on how each argument is used inside the function. If the argument can be used as a “series”, it is qualified as such. If it cannot, an attempt is made with the “simple” type qualifier. This explains why this code:

```
1 export myEma(int x) =>
2     ta.ema(close, x)
```

will work when called using `myCustomLibrary.myEma(20)`, even though `ta.ema()`’s `length` parameter requires a “simple int” argument. When the Pine Script™ compiler detects that a “series” `length` cannot be used with `ta.ema()`, it tries the “simple” qualifier, which in this case is allowed.

While library functions cannot return “const” or “input” values, they can be written to produce “simple” results. This makes them useful in more contexts than functions returning “series” results, as some built-in functions do not allow “series” arguments. For example, `request.security()` requires a “simple string” for its `symbol` parameter. If we wrote a library function to assemble the argument to `symbol` in the following way, the function’s result would not work because it is of the “series string” qualified type:

```
1 export makeTickerid(string prefix, string ticker) =>
2     prefix + ":" + ticker
```

However, by restricting the parameter qualifiers to “simple”, we can force the function to yield a “simple” result. We can achieve this by prefixing the parameters’ type with the `simple` keyword:

```
1 export makeTickerid(simple string prefix, simple string ticker) =>
2     prefix + ":" + ticker
```

Note that for the function to return a “simple” value, no “series” values can be used in its calculation; otherwise the result will be a “series” value.

One can also use the `series` keyword to prefix the type of a library function parameter. However, because arguments are qualified as “series” by default, using the `series` modifier is redundant.

## User-defined types and objects

You can export *user-defined types (UDTs)* from libraries, and library functions can return *objects*.

To export a UDT, prefix its definition with the `export` keyword just as you would export a function:

```
1 // @version=5
2 library("Point")
3
4 export type point
5     int x
6     float y
7     bool isHi
8     bool wasBreached = false
```

A script importing that library and creating an object from its `point` UDT would look somewhat like this:

Note that:

- This code won’t compile because no “Point” library is published, and the script doesn’t display anything.
- `userName` would need to be replaced by the TradingView user name of the library’s publisher.
- We use the built-in `new()` method to create an object from the `point` UDT.

- We prefix the reference to the library's point UDT with the pt alias defined in the import statement, just like we would when using a function from an imported library.

UDTs used in a library **must** be exported if any of its exported functions use a parameter or returns a result of that user-defined type.

When a library only uses a UDT internally, it does not have to be exported. The following library uses the point UDT internally, but only its drawPivots() function is exported, which does not use a parameter nor return a result of point type:

```

1 // @version=5
2 library("PivotLabels", true)
3
4 // We use this `point` UDT in the library, but it does NOT require exporting because:
5 //   1. The exported function's parameters do not use the UDT.
6 //   2. The exported function does not return a UDT result.
7 type point
8     int x
9     float y
10    bool isHi
11    bool wasBreached = false
12
13
14 fillPivotsArray(qtyLabels, leftLegs, rightLegs) =>
15     // Create an array of the specified qty of pivots to maintain.
16     var pivotsArray = array.new<point>(math.max(qtyLabels, 0))
17
18     // Detect pivots.
19     float pivotHi = ta.pivothigh(leftLegs, rightLegs)
20     float pivotLo = ta.pivotlow(leftLegs, rightLegs)
21
22     // Create a new `point` object when a pivot is found.
23     point foundPoint = switch
24         pivotHi => point.new(time[rightLegs], pivotHi, true)
25         pivotLo => point.new(time[rightLegs], pivotLo, false)
26         => na
27
28     // Add new pivot info to the array and remove the oldest pivot.
29     if not na(foundPoint)
30         array.push(pivotsArray, foundPoint)
31         array.shift(pivotsArray)
32
33     array<point> result = pivotsArray
34
35
36 detectBreaches(pivotsArray) =>
37     // Detect breaches.
38     for [i, eachPoint] in pivotsArray
39         if not na(eachPoint)
40             if not eachPoint.wasBreached
41                 bool hiWasBreached =      eachPoint.isHi and high[1] <= eachPoint.y_
42                 ↪and high > eachPoint.y
43                 bool loWasBreached = not eachPoint.isHi and low[1]  >= eachPoint.y_
44                 ↪and low  < eachPoint.y
45                 if hiWasBreached or loWasBreached
46                     // This pivot was breached; change its `wasBreached` field.
47                     point p = array.get(pivotsArray, i)
48                     p.wasBreached := true

```

(continues on next page)

(continued from previous page)

```

47         array.set(pivotsArray, i, p)
48
49
50 drawLabels(pivotsArray) =>
51     for eachPoint in pivotsArray
52         if not na(eachPoint)
53             label.new(
54                 eachPoint.x,
55                 eachPoint.y,
56                 str.tostring(eachPoint.y, format.mintick),
57                 xloc.bar_time,
58                 color = eachPoint.wasBreached ? color.gray : eachPoint.isHi ? color.
59 ←teal : color.red,
60                 style = eachPoint.isHi ? label.style_label_down: label.style_label_up,
61                 textcolor = eachPoint.wasBreached ? color.silver : color.white)
62
63 // @function      Displays a label for each of the last `qtyLabels` pivots.
64 //                  Colors high pivots in green, low pivots in red, and breached_
65 ←pivots in gray.
66 // @param qtyLabels (simple int) Quantity of last labels to display.
67 // @param leftLegs  (simple int) Left pivot legs.
68 // @param rightLegs (simple int) Right pivot legs.
69 // @returns        Nothing.
70 export drawPivots(int qtyLabels, int leftLegs, int rightLegs) =>
71     // Gather pivots as they occur.
72     pointsArray = fillPivotsArray(qtyLabels, leftLegs, rightLegs)
73
74     // Mark breached pivots.
75     detectBreaches(pointsArray)
76
77     // Draw labels once.
78     if barstate.islastconfirmedhistory
79         drawLabels(pointsArray)
80
81 // Example use of the function.
82 drawPivots(20, 10, 5)

```

If the TradingView user published the above library, it could be used like this:

### 4.11.3 Publishing a library

Before you or other Pine Script™ programmers can reuse any library, it must be published. If you want to share your library with all TradingViewers, publish it publicly. To use it privately, use a private publication. As with indicators or strategies, the active chart when you publish a library will appear in both its widget (the small placeholder denoting libraries in the TradingView scripts stream) and script page (the page users see when they click on the widget).

Private libraries can be used in public Protected or Invite-only scripts.

After adding our example library to the chart and setting up a clean chart showing our library plots the way we want them, we use the Pine Editor's "Publish Script" button. The "Publish Library" window comes up:

The screenshot shows the "Publish Library" interface. On the left, there's a code editor with the title "AllTimeHighLow". Below it, two functions are listed: `hi(val)` and `lo(val)`. Each function has its description, parameters, and returns information. On the right, there are sections for "Privacy settings" (set to "Public"), "Visibility" (set to "Open"), "Category" (selected as "Statistics and Metrics"), and "Tags" (with tags "all-time", "high", and "low"). A large blue button at the bottom right says "Publish Public Library".

Note that:

- We leave the library's title as is (the `title` argument in our `library()` declaration statement is used as the default). While you can change the publication's title, it is preferable to keep its default value because the `title` argument is used to reference imported libraries in the `import` statement. It makes life easier for library users when your publication's title matches the actual name of the library.
- A default description is built from the *compiler annotations* we used in our library. We will publish the library without retouching it.
- We chose to publish our library publicly, so it will be visible to all TradingViewers.
- We do not have the possibility of selecting a visibility type other than “Open” because libraries are always open-source.
- The list of categories for libraries is different than for indicators and strategies. We have selected the “Statistics and Metrics” category.
- We have added some custom tags: “all-time”, “high” and “low”.

The intended users of public libraries being other Pine programmers; the better you explain and document your library's functions, the more chances others will use them. Providing examples demonstrating how to use your library's functions in your publication's code will also help.

## House Rules

Pine libraries are considered “public domain” code in our [House Rules on Script Publishing](#), which entails that permission is not required from their author if you call their functions or reuse their code in your open-source scripts. However, if you intend to reuse code from a Pine Script™ library’s functions in a public protected or invite-only publication, explicit permission for reuse in that form is required from its author.

Whether using a library’s functions or reusing its code, you must credit the author in your publication’s description. It is also good form to credit in open-source comments.

### 4.11.4 Using a library

Using a library from another script (which can be an indicator, a strategy or another library), is done through the `import` statement:

```
import <username>/<libraryName>/<libraryVersion> [as <alias>]
```

where:

- The `<username>/<libraryName>/<libraryVersion>` path will uniquely identify the library.
- The `<libraryVersion>` must be specified explicitly. To ensure the reliability of scripts using libraries, there is no way to automatically use the latest version of a library. Every time a library update is published by its author, the library’s version number increases. If you intend to use the latest version of the library, the `<libraryVersion>` value will require updating in the `import` statement.
- The `as <alias>` part is optional. When used, it defines the namespace that will refer to the library’s functions. For example, if you import a library using the `allTime` alias as we do in the example below, you will refer to that library’s functions as `allTime.<function_name>()`. When no alias is defined, the library’s name becomes its namespace.

To use the library we published in the previous section, our next script will require an `import` statement:

```
import PineCoders/AllTimeHighLow/1 as allTime
```

As you type the user name of the library’s author, you can use the Editor’s `ctrl + space / cmd + space` “Auto-complete” command to display a popup providing selections that match the available libraries:



This is an indicator that reuses our library:

```
1 //@version=5
2 indicator("Using AllTimeHighLow library", "", true)
3 import PineCoders/AllTimeHighLow/1 as allTime
4
5 plot(allTime.hi())
6 plot(allTime.lo())
7 plot(allTime.hi(close))
```

Note that:

- We have chosen to use the “allTime” alias for the library’s instance in our script. When typing that alias in the Editor, a popup will appear to help you select the particular function you want to use from the library.
- We use the library’s `hi()` and `lo()` functions without an argument, so the default `high` and `low` built-in variables will be used for their series, respectively.
- We use a second call to `allTime.hi()`, but this time using `close` as its argument, to plot the highest `close` in the chart’s history.



## 4.12 Lines and boxes

- *Introduction*
- *Lines*
- *Boxes*
- *Polylines*
- *Realtime behavior*
- *Limitations*

### 4.12.1 Introduction

Pine Script™ facilitates drawing lines, boxes, and other geometric formations from code using the `line`, `box`, and `polyline` types. These types provide utility for programmatically drawing support and resistance levels, trend lines, price ranges, and other custom formations on a chart.

Unlike `plots`, the flexibility of these types makes them particularly well-suited for visualizing current calculated data at virtually any available point on the chart, irrespective of the chart bar the script executes on.

`Lines`, `boxes`, and `polylines` are *objects*, like `labels`, `tables`, and other *special types*. Scripts reference objects of these types using IDs, which act like *pointers*. As with other objects, `line`, `box`, and `polyline` IDs are qualified as “series” values, and all functions that manage these objects accept “series” arguments.

---

**Note:** Using the types we discuss on this page often involves `arrays`, especially when working with `polylines`, which require an `array` of `chart.point` instances. We therefore recommend you become familiar with `arrays` to make the most of these drawing types in your scripts.

---

Lines drawn by a script may be vertical, horizontal, or angled. Boxes are always rectangular. Polylines sequentially connect multiple vertical, horizontal, angled, or curved line segments. Although all of these drawing types have different characteristics, they do have some things in common:

- *Lines*, *boxes*, and *polylines* can have coordinates at any available location on the chart, including ones at future times beyond the last chart bar.
- Objects of these types can use `chart.point` instances to set their coordinates.
- The x-coordinates of each object can be bar index or time values, depending on their specified `xloc` property.
- Each object can have one of multiple predefined line styles.
- Scripts can call the functions that manage these objects from within the scopes of *loops* and *conditional structures*, allowing iterative and conditional control of their drawings.
- There are limits on the number of these objects that a script can reference and display on the chart. A single script instance can display up to 500 lines, 500 boxes, and 100 polylines. Users can specify the maximum number allowed for each type via the `max_lines_count`, `max_boxes_count`, and `max_polylines_count` parameters of the script's `indicator()` or `strategy()` declaration statement. If unspecified, the default is ~50. As with `label` and `table` types, lines, boxes, and polylines utilize a *garbage collection* mechanism that deletes the oldest objects on the chart when the total number of drawings exceeds the script's limit.

**Note:** On TradingView charts, a complete set of *Drawing Tools* allows users to create and modify drawings using mouse actions. While they may sometimes resemble drawing objects created with Pine Script™ code, they are **unrelated** entities. Pine scripts cannot interact with drawing tools from the chart user interface, and mouse actions do not directly affect Pine drawing objects.

## 4.12.2 Lines

The built-ins in the `line.*` namespace control the creation and management of `line` objects:

- The `line.new()` function creates a new line.
- The `line.set_*` () functions modify line properties.
- The `line.get_*` () functions retrieve values from a line instance.
- The `line.copy()` function clones a line instance.
- The `line.delete()` function deletes an existing line instance.
- The `line.all` variable references a read-only `array` containing the IDs of all lines displayed by the script. The array's `size` depends on the `max_lines_count` of the `indicator()` or `strategy()` declaration statement and the number of lines the script has drawn.

Scripts can call `line.set_*` (), `line.get_*` (), `line.copy()`, and `line.delete()` built-ins as functions or *methods*.

### Creating lines

The `line.new()` function creates a new `line` instance to display on the chart. It has the following signatures:

```
line.new(first_point, second_point, xloc, extend, color, style, width) → series line
line.new(x1, y1, x2, y2, xloc, extend, color, style, width) → series line
```

The first overload of this function contains the `first_point` and `second_point` parameters. The `first_point` is a `chart.point` representing the start of the line, and the `second_point` is a `chart.point` representing the line's end. The function copies the information from these *chart points* to determine the line's coordinates. Whether it uses the `index` or `time` fields from the `first_point` and `second_point` as x-coordinates depends on the function's `xloc` value.

The second overload specifies `x1`, `y1`, `x2`, and `y2` values independently, where `x1` and `x2` are `int` values representing the starting and ending x-coordinates of the line, and `y1` and `y2` are `float` values representing the y-coordinates. Whether the line considers the `x` values as bar indices or timestamps depends on the `xloc` value in the function call.

Both overloads share the same additional parameters:

### `xloc`

Controls whether the x-coordinates of the new line use bar index or time values. Its default value is `xloc.bar_index`.

When calling the first overload, using an `xloc` value of `xloc.bar_index` tells the function to use the `index` fields of the `first_point` and `second_point`, and a value of `xloc.bar_time` tells the function to use the `time` fields of the points.

When calling the second overload, an `xloc` value of `xloc.bar_index` prompts the function to treat the `x1` and `x2` arguments as bar index values. When using `xloc.bar_time`, the function will treat `x1` and `x2` as time values.

When the specified x-coordinates represent *bar index* values, it's important to note that the minimum x-coordinate allowed is `bar_index = 9999`. For larger offsets, one can use `xloc.bar_time`.

### `extend`

Determines whether the drawn line will infinitely extend beyond its defined start and end coordinates. It accepts one of the following values: `extend.left`, `extend.right`, `extend.both`, or `extend.none` (default).

### `color`

Specifies the color of the line drawing. The default is `color.blue`.

### `style`

Specifies the line's style, which can be any of the options listed in this page's *Line styles* section. The default value is `line.style_solid`.

### `width`

Controls the width of the line, in pixels. The default value is 1.

The example below demonstrates how one can draw lines in their simplest form. This script draws a new vertical line connecting the `open` and `close` prices at the horizontal center of each chart bar:



```

1 //@version=5
2 indicator("Creating lines demo", overlay = true)
3
4 //@variable The `chart.point` for the start of the line. Contains `index` and `time`_
5 ↵information.
6 firstPoint = chart.point.now(open)

```

(continues on next page)

(continued from previous page)

```

6 // @variable The `chart.point` for the end of the line. Contains `index` and `time`_  

7 // information.  

8 secondPoint = chart.point.now(close)  

9  

10 // Draw a basic line with a `width` of 5 connecting the `firstPoint` to the_  

11 // `secondPoint`.  

12 // This line uses the `index` field from each point for its x-coordinates.  

13 line.new(firstPoint, secondPoint, width = 5)  

14  

15 // Color the background on the unconfirmed bar.  

16 bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed_  

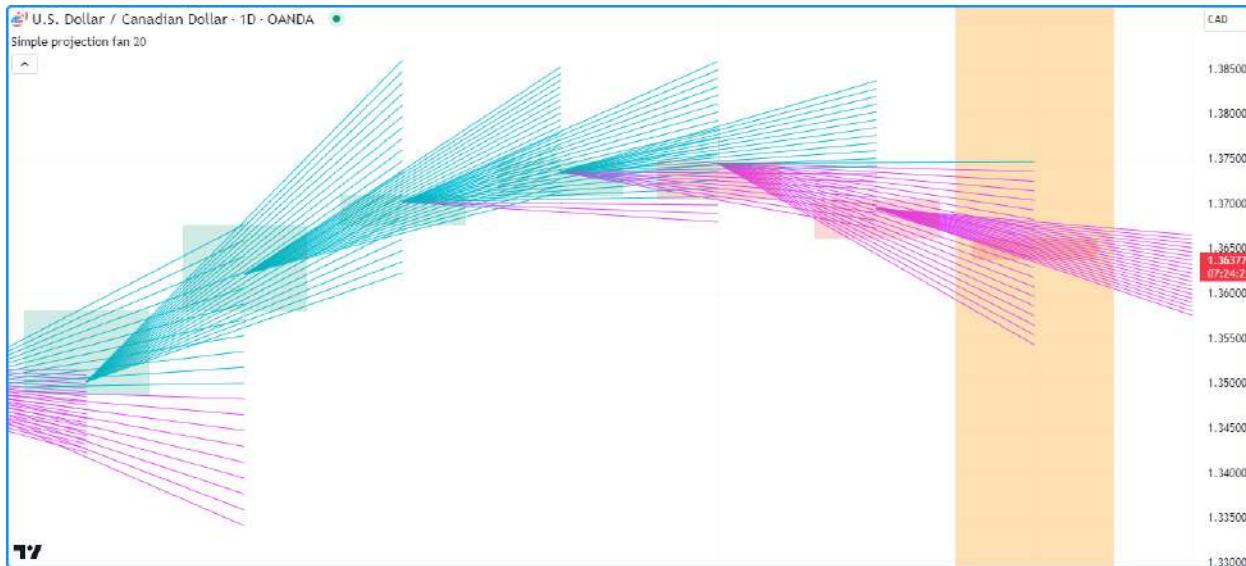
17 // bar highlight")

```

**Note that:**

- If the `firstPoint` and `secondPoint` reference identical coordinates, the script will *not* display a line since there is no distance between them to draw. However, the line ID will still exist.
- The script will only display approximately the last 50 lines on the chart, as it does not have a specified `max_lines_count` in the `indicator()` function call. Line drawings persist on the chart until deleted using `line.delete()` or removed by the garbage collector.
- The script *redraws* the line on the open chart bar (i.e., the bar with an orange background highlight) until it closes. After the bar closes, it will no longer update the drawing.

Let's look at a more involved example. This script uses the previous bar's `hl2` price and the current bar's `high` and `low` prices to draw a fan with a user-specified number of lines projecting a range of hypothetical price values for the following chart bar. It calls `line.new()` within a `for` loop to create `linesPerBar` lines on each bar:



```

1 // @version=5  

2 indicator("Creating lines demo", "Simple projection fan", true, max_lines_count = 500)  

3  

4 // @variable The number of fan lines drawn on each chart bar.  

5 int linesPerBar = input.int(20, "Line drawings per bar", 2, 100)  

6  

7 // @variable The distance between each y point on the current bar.  

8 float step = (high - low) / (linesPerBar - 1)

```

(continues on next page)

(continued from previous page)

```

9 // @variable The `chart.point` for the start of each line. Does not contain `time`_  

10 // information.  

11 firstPoint = chart.point.from_index(bar_index - 1, h12[1])  

12 // @variable The `chart.point` for the end of each line. Does not contain `time`_  

13 // information.  

14 secondPoint = chart.point.from_index(bar_index + 1, float(na))  

15 // @variable The stepped y value on the current bar for `secondPoint.price`_  

16 // calculation, starting from the `low`.  

17 float barValue = low  

18 // Loop to draw the fan.  

19 for i = 1 to linesPerBar  

20     // Update the `price` of the `secondPoint` using the difference between the_  

21     // `barValue` and `firstPoint.price`.  

22     secondPoint.price := 2.0 * barValue - firstPoint.price  

23     // @variable Is `color.aqua` when the line's slope is positive, `color.fuchsia`_  

24     // otherwise.  

25     color lineColor = secondPoint.price > firstPoint.price ? color.aqua : color.  

26     fuchsia  

27     // Draw a new `lineColor` line connecting the `firstPoint` and `secondPoint`_  

28     // coordinates.  

29     // This line uses the `index` field from each point for its x-coordinates.  

30     line.new(firstPoint, secondPoint, color = lineColor)  

31     // Add the `step` to the `barValue`.  

32     barValue += step  

33  

34 // Color the background on the unconfirmed bar.  

35 bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed_  

36 // bar highlight")
```

**Note that:**

- We've included `max_lines_count = 500` in the `indicator()` function call, meaning the script preserves up to 500 lines on the chart.
- Each `line.new()` call *copies* the information from the `chart.point` referenced by the `firstPoint` and `secondPoint` variables. As such, the script can change the `price` field of the `secondPoint` on each loop iteration without affecting the `y`-coordinates in other lines.

## Modifying lines

The `line.*` namespace contains multiple *setter* functions that modify the properties of `line` instances:

- `line.set_first_point()` and `line.set_second_point()` respectively update the start and end points of the `id` line using information from the specified `point`.
- `line.set_x1()` and `line.set_x2()` set one of the `x`-coordinates of the `id` line to a new `x` value, which can represent a bar index or time value depending on the line's `xloc` property.
- `line.set_y1()` and `line.set_y2()` set one of the `y`-coordinates of the `id` line to a new `y` value.
- `line.set_xy1()` and `line.set_xy2()` update one of the `id` line's points with new `x` and `y` values.
- `line.set_xloc()` sets the `xloc` of the `id` line and updates both of its `x`-coordinates with new `x1` and `x2` values.
- `line.set_extend()` sets the `extend` property of the `id` line.
- `line.set_color()` updates the `id` line's `color` value.

- `line.set_style()` changes the style of the `id` line.
- `line.set_width()` sets the width of the `id` line.

All setter functions directly modify the `id` line passed into the call and do not return any value. Each setter function accepts “series” arguments, as a script can change a line’s properties throughout its execution.

The following example draws lines connecting the opening price of a timeframe to its closing price. The script uses the `var` keyword to declare the `periodLine` and the variables that reference `chart.point` values (`openPoint` and `closePoint`) only on the *first* chart bar, and it assigns new values to these variables over its execution. After detecting a `change` on the timeframe, it sets the `color` of the existing `periodLine` using `line.set_color()`, creates new values for the `openPoint` and `closePoint` using `chart.point.now()`, then assigns a `new line` using those points to the `periodLine`.

On other bars where the `periodLine` value is not `na`, the script assigns a new `chart.point` to the `closePoint`, then uses `line.set_second_point()` and `line.set_color()` as *methods* to update the line’s properties:



```

1 // @version=5
2 indicator("Modifying lines demo", overlay = true)
3
4 // @variable The size of each period.
5 string timeframe = input.timeframe("D", "Timeframe")
6
7 // @variable A line connecting the period's opening and closing prices.
8 var line periodLine = na
9
10 // @variable The first point of the line. Contains `time` and `index` information.
11 var chart.point openPoint = chart.point.now(open)
12 // @variable The closing point of the line. Contains `time` and `index` information.
13 var chart.point closePoint = chart.point.now(close)
14
15 if timeframe.change(timeframe)
16     // @variable The final color of the `periodLine`.
17     color finalColor = switch
18         closePoint.price > openPoint.price => color.green
19         closePoint.price < openPoint.price => color.red
20         =>                                         color.gray
21
22     // Update the color of the current `periodLine` to the `finalColor`.

```

(continues on next page)

(continued from previous page)

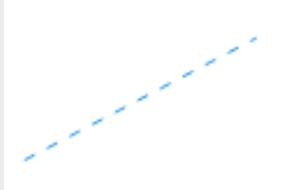
```
23     line.set_color(periodLine, finalColor)
24
25     // Assign new points to the `openPoint` and `closePoint`.
26     openPoint := chart.point.now(open)
27     closePoint := chart.point.now(close)
28     // Assign a new line to the `periodLine`. Uses `time` fields from the `openPoint` ↵
29     // and `closePoint` as x-coordinates.
30     periodLine := line.new(openPoint, closePoint, xloc.bar_time, style = line.style_ ↵
31     arrow_right, width = 3)
32
33 else if not na(periodLine)
34     // Assign a new point to the `closePoint`.
35     closePoint := chart.point.now(close)
36
37     //@variable The color of the developing `periodLine`.
38     color developingColor = switch
39         closePoint.price > openPoint.price => color.aqua
40         closePoint.price < openPoint.price => color.fuchsia
41         => color.gray
42
43     // Update the coordinates of the line's second point using the new `closePoint`.
44     // It uses the `time` field from the point for its new x-coordinate.
45     periodLine.set_second_point(closePoint)
46     // Update the color of the line using the `developingColor`.
47     periodLine.set_color(developingColor)
```

**Note that:**

- Each line drawing in this example uses the `line.style_arrow_right` style. See the *Line styles* section below for an overview of all available style settings.

## Line styles

Users can control the style of their scripts' line drawings by passing one of the following variables as the `style` argument in their `line.new()` or `line.set_style()` function calls:

Argument	Line	Argument	Line
line. style_solid		line. style_arrow_left	
line. style_dotted		line. style_arrow_right	
line. style_dashed		line. style_arrow_both	

**Note that:**

- *Polyline*s can also use any of these variables as their `line_style` value. See the [Creating polylines](#) section of this page.

**Reading line values**

The `line.*` namespace includes *getter* functions, which allow a script to retrieve values from a `line` object for further use:

- `line.get_x1()` and `line.get_x2()` respectively get the first and second x-coordinate from the `id` line. Whether the value returned represents a bar index or time value depends on the line's `xloc` property.
- `line.get_y1()` and `line.get_y2()` respectively get the `id` line's first and second y-coordinate.
- `line.get_price()` retrieves the price (y-coordinate) from a line `id` at a specified `x` value, including at bar indices outside the line's start and end points. This function is only compatible with lines that use `xloc.bar_index` as the `xloc` value.

The script below draws a new line upon the onset of a `rising` or `falling` price pattern forming over `length` bars. It uses the `var` keyword to declare the `directionLine` variable on the first chart bar. The ID assigned to the `directionLine` persists over subsequent bars until the `newDirection` condition occurs, in which case the script assigns a new line to the variable.

On every bar, the script calls the `line.get_y2()`, `line.get_y1()`, `line.get_x2()`, and `line.get_x1()` getters as *methods* to retrieve values from the current `directionLine` and calculate its `slope`, which it uses to determine the color of each drawing and plot. It retrieves extended values of the `directionLine` from `beyond` its second point using `line.get_price()` and `plots` them on the chart:



```

1 // @version=5
2 indicator("Reading line values demo", overlay = true)
3
4 //@variable The number of bars for rising and falling calculations.
5 int length = input.int(2, "Length", 2)
6
7 //@variable A line that's drawn whenever `hlc3` starts rising or falling over
8 // `length` bars.
9 var line directionLine = na
10
11 //@variable Is `true` when `hlc3` is rising over `length` bars, `false` otherwise.
12 bool rising = ta.rising(hlc3, length)
13 //@variable Is `true` when `hlc3` is falling over `length` bars, `false` otherwise.
14 bool falling = ta.falling(hlc3, length)
15 //@variable Is `true` when a rising or falling pattern begins, `false` otherwise.
16 bool newDirection = (rising and not rising[1]) or (falling and not falling[1])
17
18 // Update the `directionLine` when `newDirection` is `true`. The line uses the
19 // default `xloc.bar_index`.
20 if newDirection
21     directionLine := line.new(bar_index - length, hlc3[length], bar_index, hlc3,
22 //width = 3)
23
24 //@variable The slope of the `directionLine`.
25 float slope = (directionLine.get_y2() - directionLine.get_y1()) / (directionLine.get_
26 //x2() - directionLine.get_x1())
27 //@variable The value extrapolated from the `directionLine` at the `bar_index`.
28 float lineValue = line.get_price(directionLine, bar_index)
29
30 //@variable Is `color.green` when the `slope` is positive, `color.red` otherwise.
31 color slopeColor = slope > 0 ? color.green : color.red
32
33 // Update the color of the `directionLine`.
34 directionLine.set_color(slopeColor)
35 // Plot the `lineValue`.
36 plot(lineValue, "Extrapolated value", slopeColor, 3, plot.style_circles)

```

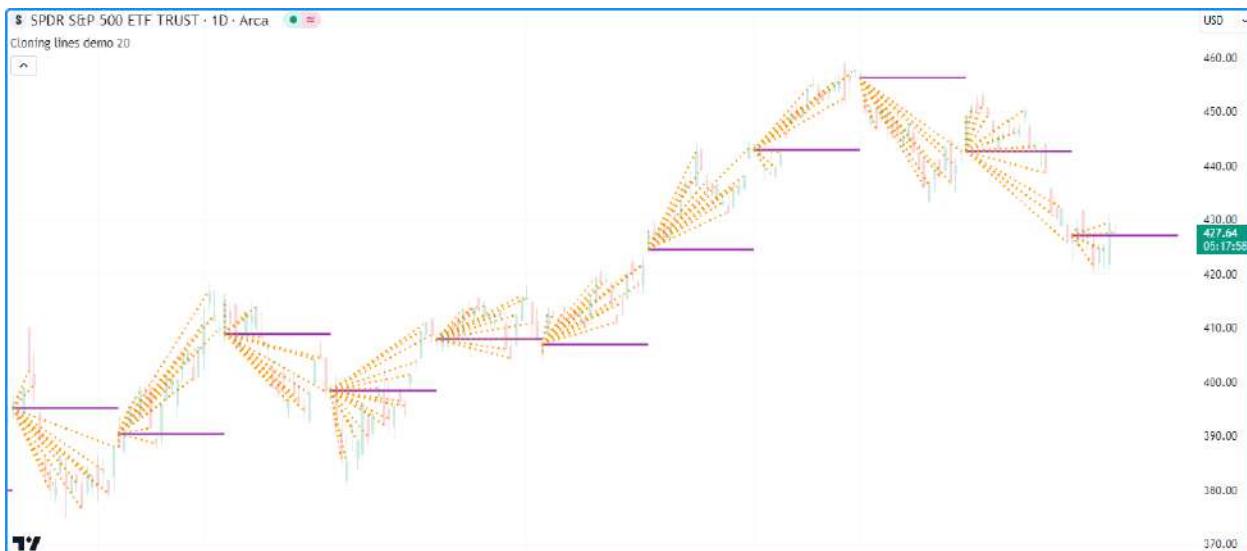
**Note that:**

- This example calls the second overload of the `line.new()` function, which uses `x1`, `y1`, `x2`, and `y2` parameters to define the start and end points of the line. The `x1` value is `length` bars behind the current `bar_index`, and the `y1` value is the `hlC3` value at that index. The `x2` and `y2` in the function call use the current bar's `bar_index` and `hlC3` values.
  - The `line.get_price()` function call treats the `directionLine` as though it extends infinitely, regardless of its `extend` property.
  - The script only displays approximately the last 50 lines on the chart, but the `plot` of extrapolated values spans throughout the chart's history.

## Cloning lines

Scripts can clone a line `id` and all its properties with the `line.copy()` function. Any changes to the copied line instance do not affect the original.

For example, this script creates a horizontal line at the the bar's `open` price once every `length` bars, which it assigns to a `mainLine` variable. On all other bars, it creates a `copiedLine` using `line.copy()` and calls `line.set_*()` functions to modify its properties. As we see below, altering the `copiedLine` does not affect the `mainLine` in any way:



```
1 // @version=5
2 indicator("Cloning lines demo", overlay = true, max_lines_count = 500)
3
4 //@variable The number of bars between each new mainLine assignment.
5 int length = input.int(20, "Length", 2, 500)
6
7 //@variable The first `chart.point` used by the `mainLine`. Contains `index` and
8 // `time` information.
9 firstPoint = chart.point.now(open)
10 //@variable The second `chart.point` used by the `mainLine`. Does not contain `time` ↴
11 // information.
12 secondPoint = chart.point.from_index(bar_index + length, open)
13
14 //@variable A horizontal line drawn at the `open` price once every `length` bars.
15 var line mainLine = na
16
17 if bar_index % length == 0
```

(continues on next page)

(continued from previous page)

```

16 // Assign a new line to the `mainLine` that connects the `firstPoint` to the_
17 // `secondPoint` .
18 // This line uses the `index` fields from both points as x-coordinates.
19 mainLine := line.new(firstPoint, secondPoint, color = color.purple, width = 2)
20
21 // @variable A copy of the `mainLine` . Changes to this line do not affect the original.
22 line copiedLine = line.copy(mainLine)
23
24 // Update the color, style, and second point of the `copiedLine` .
25 line.set_color(copiedLine, color.orange)
26 line.set_style(copiedLine, line.style_dotted)
27 line.set_second_point(copiedLine, chart.point.now(close))

```

### Note that:

- The `index` field of the `secondPoint` is `length` bars beyond the current `bar_index`. Since the maximum x-coordinate allowed with `xloc.bar_index` is `bar_index + 500`, we've set the `maxval` of the `length` input to 500.

### Deleting lines

To delete a line `id` drawn by a script, use the `line.delete()` function. This function removes the line instance from the script and its drawing on the chart.

Deleting line instances is often handy when one wants to only keep a specific number of lines on the chart at any given time or conditionally remove drawings as a chart progresses.

For example, this script draws a horizontal line with the `extend.right` property whenever an `RSI` crosses its `EMA`.

The script stores all line IDs in a `lines` array that it *uses as a queue* to only display the last `numberOfLines` on the chart. When the `size` of the array exceeds the specified `numberOfLines`, the script removes the array's oldest line ID using `array.shift()` and deletes it with `line.delete()`:



```

1 // @version=5
2
3 // @variable The maximum number of lines allowed on the chart.
4 const int MAX_LINES_COUNT = 500

```

(continues on next page)

(continued from previous page)

```

5   indicator("Deleting lines demo", "RSI cross levels", max_lines_count = MAX_LINES_
6   ↪COUNT)
7
8 // @variable The length of the RSI.
9 int rsiLength = input.int(14, "RSI length", 2)
10 // @variable The length of the RSI's EMA.
11 int emaLength = input.int(28, "RSI average length", 2)
12 // @variable The maximum number of lines to keep on the chart.
13 int numberofLines = input.int(20, "Lines on the chart", 0, MAX_LINES_COUNT)
14
15 // @variable An array containing the IDs of lines on the chart.
16 var array<line> lines = array.new<line>()
17
18 // @variable An `rsiLength` RSI of `close`.
19 float rsi = ta.rsi(close, rsiLength)
20 // @variable A `emaLength` EMA of the `rsi`.
21 float rsiMA = ta.ema(rsi, emaLength)
22
23 if ta.cross(rsi, rsiMA)
24     // @variable The color of the horizontal line.
25     color lineColor = rsi > rsiMA ? color.green : color.red
26     // Draw a new horizontal line. Uses the default `xloc.bar_index`.
27     newLine = line.new(bar_index, rsiMA, bar_index + 1, rsiMA, extend = extend.right, ↪
28     ↪color = lineColor, width = 2)
29     // Push the `newLine` into the `lines` array.
30     lines.push(newLine)
31     // Delete the oldest line when the size of the array exceeds the specified ↪
32     ↪`numberofLines`.
33     if array.size(lines) > numberofLines
34         line.delete(lines.shift())
35
36 // Plot the `rsi` and `rsiMA`.
37 plot(rsi, "RSI", color.new(color.blue, 40))
38 plot(rsiMA, "EMA of RSI", color.new(color.gray, 30))

```

**Note that:**

- We declared a `MAX_LINES_COUNT` variable with the “`const int`” *qualified type*, which the script uses as the `max_lines_count` in the `indicator()` function and the `maxval` of the `input.int()` assigned to the `numberofLines` variable.
- This example uses the second overload of the `line.new()` function, which specifies `x1`, `y1`, `x2`, and `y2` coordinates independently.

**Filling the space between lines**

Scripts can *fill* the space between two `line` drawings by creating a `linefill` object that references them with the `linefill.new()` function. Linefills automatically determine their fill boundaries using the properties from the `line1` and `line2` IDs that they reference.

For example, this script calculates a simple linear regression channel. On the first chart bar, the script declares the `basisLine`, `upperLine`, and `lowerLine` variables to reference the channel’s `line` IDs, then it makes two `linefill.new()` calls to create `linefill` objects that fill the upper and lower portions of the channel. The first `linefill` fills the space between the `basisLine` and the `upperLine`, and the second fills the space between the `basisLine` and `lowerLine`.

The script updates the coordinates of the lines across subsequent bars. However, notice that the script never needs to

update the linefills declared on the first bar. They automatically update their fill regions based on the coordinates of their assigned lines:



```

1 // @version=5
2 indicator("Filling the space between lines demo", "Simple linreg channel", true)
3
4 // @variable The number of bars in the linear regression calculation.
5 int lengthInput = input.int(100)
6
7 // @variable The basis line of the regression channel.
8 var line basisLine = line.new(na, na, na, na, extend = extend.right, color = chart.fg_
  ↵color, width = 2)
9 // @variable The channel's upper line.
10 var line upperLine = line.new(na, na, na, na, extend = extend.right, color = color.
   ↵teal, width = 2)
11 // @variable The channel's lower line.
12 var line lowerLine = line.new(na, na, na, na, extend = extend.right, color = color.
   ↵maroon, width = 2)
13
14 // @variable A linefill instance that fills the space between the `basisLine` and_
  ↵`upperLine`.
15 var linefill upperFill = linefill.new(basisLine, upperLine, color.new(color.teal, 80))
16 // @variable A linefill instance that fills the space between the `basisLine` and_
  ↵`lowerLine`.
17 var linefill lowerFill = linefill.new(basisLine, lowerLine, color.new(color.maroon,_
  ↵80))
18
19 // Update the `basisLine` coordinates with current linear regression values.
20 basisLine.set_xy1(bar_index + 1 - lengthInput, ta.linreg(close, lengthInput,_
  ↵lengthInput - 1))
21 basisLine.set_xy2(bar_index, ta.linreg(close, lengthInput, 0))
22
23 // @variable The channel's standard deviation.
24 float stDev = 0.0
25 for i = 0 to lengthInput - 1
26     stDev += math.pow(close[i] - line.get_price(basisLine, bar_index - i), 2)
27 stDev := math.sqrt(stDev / lengthInput) * 2.0
28
29 // Update the `upperLine` and `lowerLine` using the values from the `basisLine` and_
  ↵

```

(continues on next page)

(continued from previous page)

```

30  ↪the `stDev` .
31  upperLine.set_xy1(basisLine.get_x1(), basisLine.get_y1() + stDev)
32  upperLine.set_xy2(basisLine.get_x2(), basisLine.get_y2() + stDev)
33  lowerLine.set_xy1(basisLine.get_x1(), basisLine.get_y1() - stDev)
33  lowerLine.set_xy2(basisLine.get_x2(), basisLine.get_y2() - stDev)

```

To learn more about the `linefill` type, see [this](#) section of the [Fills](#) page.

### 4.12.3 Boxes

The built-ins in the `box.*` namespace create and manage `box` objects:

- The `box.new()` function creates a new box.
- The `box.set_*` () functions modify box properties.
- The `box.get_*` () functions retrieve values from a box instance.
- The `box.copy()` function clones a box instance.
- The `box.delete()` function deletes a box instance.
- The `box.all` variable references a read-only `array` containing the IDs of all boxes displayed by the script. The array's `size` depends on the `max_boxes_count` of the `indicator()` or `strategy()` declaration statement and the number of boxes the script has drawn.

As with `lines`, users can call `box.set_*` (), `box.get_*` (), `box.copy()`, and `box.delete()` built-ins as functions or *methods*.

#### Creating boxes

The `box.new()` function creates a new `box` object to display on the chart. It has the following signatures:

```

box.new(top_left, bottom_right, border_color, border_width, border_style, extend,
↪xloc, bgcolor, text, text_size, text_color, text_halign, text_valign, text_wrap,
↪text_font_family) → series box

box.new(left, top, right, bottom, border_color, border_width, border_style, extend,
↪xloc, bgcolor, text, text_size, text_color, text_halign, text_valign, text_wrap,
↪text_font_family) → series box

```

This function's first overload includes the `top_left` and `bottom_right` parameters, which accept `chart.point` objects representing the top-left and bottom-right corners of the box, respectively. The function copies the information from these *chart points* to set the coordinates of the box's corners. Whether it uses the `index` or `time` fields of the `top_left` and `bottom_right` points as x-coordinates depends on the function's `xloc` value.

The second overload specifies `left`, `top`, `right`, and `bottom` edges of the box. The `left` and `right` parameters accept `int` values specifying the box's left and right x-coordinates, which can be bar index or time values depending on the `xloc` value in the function call. The `top` and `bottom` parameters accept `float` values representing the box's top and bottom y-coordinates.

The function's additional parameters are identical in both overloads:

##### `border_color`

Specifies the color of all four of the box's borders. The default is `color.blue`.

##### `border_width`

Specifies the width of the borders, in pixels. Its default value is 1.

### **border\_style**

Specifies the style of the borders, which can be any of the options in the *Box styles* section of this page.

### **extend**

Determines whether the box's borders extend infinitely beyond the left or right x-coordinates. It accepts one of the following values: `extend.left`, `extend.right`, `extend.both`, or `extend.none` (default).

### **xloc**

Determines whether the left and right edges of the box use bar index or time values as x-coordinates. The default is `xloc.bar_index`.

In the first overload, an `xloc` value of `xloc.bar_index` means that the function will use the `index` fields of the `top_left` and `bottom_right` chart points, and an `xloc` value of `xloc.bar_time` means that it will use their `time` fields.

In the second overload, using an `xloc` value of `xloc.bar_index` means the function treats the `left` and `right` values as bar indices, and `xloc.bar_time` means it will treat them as timestamps.

When the specified x-coordinates represent *bar index* values, it's important to note that the minimum x-coordinate allowed is `bar_index - 9999`. For larger offsets, one can use `xloc.bar_time`.

### **bgcolor**

Specifies the background color of the space inside the box. The default value is `color.blue`.

### **text**

The text to display inside the box. By default, its value is an empty string.

### **text\_size**

Specifies the size of the text within the box. It accepts one of the following values: `size.tiny`, `size.small`, `size.normal`, `size.large`, `size.huge`, or `size.auto` (default).

### **text\_color**

Controls the color of the text. Its default is `color.black`.

### **text\_halign**

Specifies the horizontal alignment of the text within the box's boundaries. It accepts one of the following: `text.align_left`, `text.align_right`, or `text.align_center` (default).

### **text\_valign**

Specifies the vertical alignment of the text within the box's boundaries. It accepts one of the following: `text.align_top`, `text.align_bottom`, or `text.align_center` (default).

### **text\_wrap**

Determines whether the box will wrap the text within it. If its value is `text.wrap_auto`, the box wraps the text to ensure it does not span past its vertical borders. It also clips the wrapped text when it extends past the bottom.

If the value is `text.wrap_none`, the box displays the text on a single line that can extend beyond its borders. The default is `text.wrap_none`.

### **text\_font\_family**

Defines the font family of the box's text. Using `font.family_default` displays the box's text with the system's default font. The `font.family_monospace` displays the text in a monospace format. The default value is `font.family_default`.

Let's write a simple script to display boxes on a chart. The example below draws a box projecting each bar's `high` and `low` values from the horizontal center of the current bar to the center of the next available bar.

On each bar, the script creates `topLeft` and `bottomRight` points via `chart.point.now()` and `chart.point_from_index()`, then calls `box.new()` to construct a new box and display it on the chart. It also highlights the background on the unconfirmed chart bar using `bgcolor()` to indicate that it redraws that box until the bar's last update:



```

1 // @version=5
2 indicator("Creating boxes demo", overlay = true)
3
4 // @variable The `chart.point` for the top-left corner of the box. Contains `index`_
5 // and `time` information.
6 topLeft = chart.point.now(high)
7 // @variable The `chart.point` for the bottom-right corner of the box. Does not_
8 // contain `time` information.
9 bottomRight = chart.point.from_index(bar_index + 1, low)
10
11 // Draw a box using the `topLeft` and `bottomRight` corner points. Uses the `index`_
12 // fields as x-coordinates.
13 box.new(topLeft, bottomRight, color.purple, 2, bgcolor = color.new(color.gray, 70))
14
15 // Color the background on the unconfirmed bar.
16 bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed_
17 // bar highlight")

```

#### Note that:

- The `bottomRight` point's `index` field is one bar greater than the `index` in the `topLeft`. If the x-coordinates of the corners were equal, the script would draw a vertical line at the horizontal center of each bar, resembling the example in this page's [Creating lines](#) section.
- Similar to [lines](#), if the `topLeft` and `bottomRight` contained identical coordinates, the box wouldn't display on the chart since there would be no space between them to draw. However, its ID would still exist.
- This script only displays approximately the last 50 boxes on the chart, as we have not specified a `max_boxes_count` in the `indicator()` function call.

## Modifying boxes

Multiple *setter* functions exist in the `box.*` namespace, allowing scripts to modify the properties of `box` objects:

- `box.set_top_left_point()` and `box.set_bottom_right_point()` respectively update the top-left and bottom-right coordinates of the `id` box using information from the specified point.
- `box.set_left()` and `box.set_right()` set the left or right x-coordinate of the `id` box to a new `left/right` value, which can be a bar index or time value depending on the box's `xloc` property.
- `box.set_top()` and `box.set_bottom()` set the top or bottom y-coordinate of the `id` box to a new `top/bottom` value.
- `box.set_lefttop()` sets the `left` and `top` coordinates of the `id` box, and `box.set_rightbottom()` sets its `right` and `bottom` coordinates.
- `box.set_border_color()`, `box.set_border_width()` and `box.set_border_style()` respectively update the `color`, `width`, and `style` of the `id` box's border.
- `box.set_extend()` sets the horizontal `extend` property of the `id` box.
- `box.set_bgcolor()` sets the color of the space inside the `id` box to a new `color`.
- `box.set_text()`, `box.set_text_size()`, `box.set_text_color()`, `box.set_text_halign()`, `box.set_text_valign()`, `box.set_text_wrap()`, and `box.set_text_font_family()` update the `id` box's text-related properties.

As with setter functions in the `line.*` namespace, all box setters modify the `id` box directly without returning a value, and each setter function accepts “series” arguments.

Note that, unlike `lines`, the `box.*` namespace does not contain a setter function to modify a box's `xloc`. Users must *create* a new box with the desired `xloc` setting for such cases.

This example uses boxes to visualize the ranges of upward and downward bars with the highest `volume` over a user-defined `timeframe`. When the script detects a `change` in the `timeframe`, it assigns `new boxes` to its `upBox` and `downBox` variables, resets its `upVolume` and `downVolume` values, and highlights the chart background.

When an upward or downward bar's `volume` exceeds the `upVolume` or `downVolume`, the script updates the volume-tracking variables and calls `box.set_top_left_point()` and `box.set_bottom_right_point()` to update the `upBox` or `downBox` coordinates. The setters use the information from the `chart.points.now()` and `chart.point.from_time()` to project that bar's `high` and `low` values from the current time to the `closing time` of the `timeframe`:



```

1 //@version=5
2 indicator("Modifying boxes demo", "High volume boxes", true, max_boxes_count = 100)
3
4 //@variable The timeframe of the calculation.
5 string timeframe = input.timeframe("D", "Timeframe")
6
7 //@variable A box projecting the range of the upward bar with the highest `volume`_
8 //over the `timeframe`.
9 var box upBox = na
10 //@variable A box projecting the range of the downward bar with the lowest `volume`_
11 //over the `timeframe`.
12 var box downBox = na
13 //@variable The highest volume of upward bars over the `timeframe`.
14 var float upVolume = na
15 //@variable The highest volume of downward bars over the `timeframe`.
16 var float downVolume = na
17
18 // Color variables.
19 var color upBorder = color.teal
20 var color upFill = color.new(color.teal, 90)
21 var color downBorder = color.maroon
22 var color downFill = color.new(color.maroon, 90)
23
24 //@variable The closing time of the `timeframe`.
25 int closeTime = time_close(timeframe)
26 //@variable Is `true` when a new bar starts on the `timeframe`.
27 bool changeTF = timeframe.change(timeframe)
28
29 //@variable The `chart.point` for the top-left corner of the boxes. Contains `index`_
30 //and `time` information.
31 topLeft = chart.point.now(high)
32 //@variable The `chart.point` for the bottom-right corner of the boxes. Does not_
33 //contain `index` information.
34 bottomRight = chart.point.from_time(closeTime, low)
35
36 if changeTF and not na(volume)
37     if close > open
38         // Update `upVolume` and `downVolume` values.
39         upVolume := volume
40         downVolume := 0.0
41         // Draw a new `upBox` using `time` and `price` info from the `topLeft` and_
42         //`bottomRight` points.
43         upBox := box.new(topLeft, bottomRight, upBorder, 3, xloc = xloc.bar_time,_
44         bgcolor = upFill)
45         // Draw a new `downBox` with `na` coordinates.
46         downBox := box.new(na, na, na, na, downBorder, 3, xloc = xloc.bar_time,_
47         bgcolor = downFill)
48     else
49         // Update `upVolume` and `downVolume` values.
50         upVolume := 0.0
51         downVolume := volume
52         // Draw a new `upBox` with `na` coordinates.
53         upBox := box.new(na, na, na, na, upBorder, 3, xloc = xloc.bar_time, bgcolor =_
54         upFill)
55         // Draw a new `downBox` using `time` and `price` info from the `topLeft` and_
56         //`bottomRight` points.
57         downBox := box.new(topLeft, bottomRight, downBorder, 3, xloc = xloc.bar_time,_
58         bgcolor = downFill)

```

(continues on next page)

(continued from previous page)

```

49    ↪bgcolor = downFill)
// Update the ``upVolume`` and change the ``upBox`` coordinates when volume increases
50    ↪on an upward bar.
51 else if close > open and volume > upVolume
52     upVolume := volume
53     box.set_top_left_point(upBox, topLeft)
54     box.set_bottom_right_point(upBox, bottomRight)
55 // Update the ``downVolume`` and change the ``downBox`` coordinates when volume
56    ↪increases on a downward bar.
57 else if close <= open and volume > downVolume
58     downVolume := volume
59     box.set_top_left_point(downBox, topLeft)
60     box.set_bottom_right_point(downBox, bottomRight)
61
62 // Highlight the background when a new `timeframe` bar starts.
63 bgcolor(changeTF ? color.new(color.orange, 70) : na, title = "Timeframe change
64    ↪highlight")

```

**Note that:**

- The `indicator()` function call contains `max_boxes_count = 100`, meaning the script will preserve the last 100 boxes on the chart.
- We utilized *both overloads* of `box.new()` in this example. On the first bar of the timeframe, the script calls the first overload for the `upBox` when the bar is rising, and it calls that overload for the `downBox` when the bar is falling. It uses the second overload to assign a new box with `na` values to the other box variable on that bar.

**Box styles**

Users can include one of the following `line.style_*` variables in their `box.new()` or `box.set_border_style()` function calls to set the border styles of boxes drawn by their scripts:

Argument	Box
<code>line.style_solid</code>	A solid blue rectangular box.
<code>line.style_dotted</code>	A dotted blue rectangular box.
<code>line.style_dashed</code>	A dashed blue rectangular box.

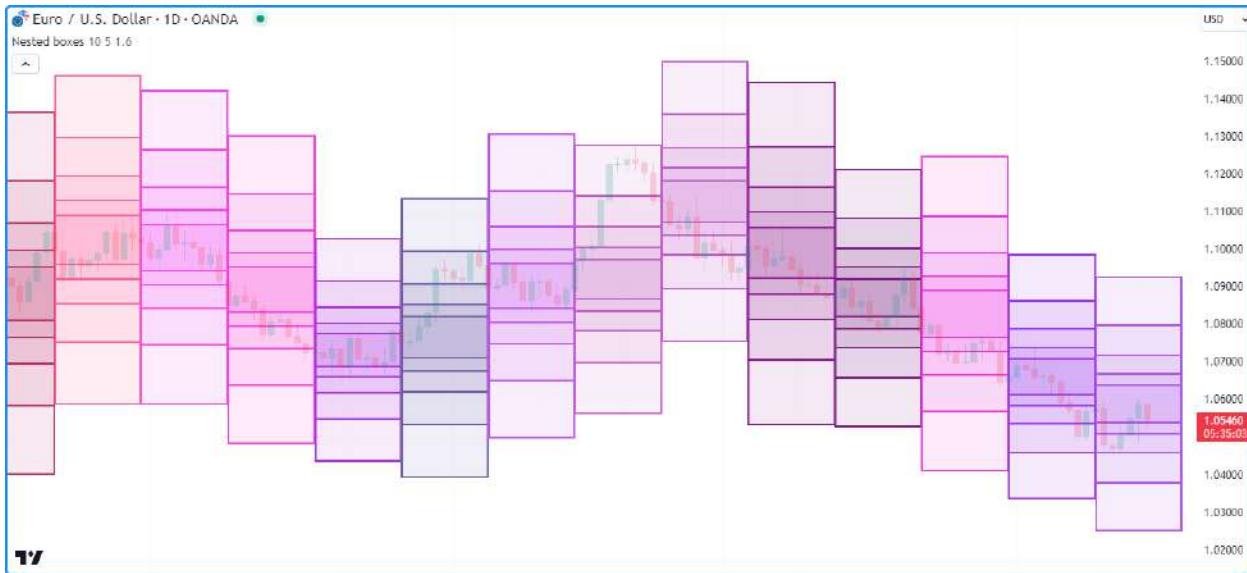
## Reading box values

The `box.*` namespace features *getter* functions that allow scripts to retrieve coordinate values from a box instance:

- `box.get_left()` and `box.get_right()` respectively get the x-coordinates of the left and right edges of the `id` box. Whether the value returned represents a bar index or time value depends on the box's `xloc` property.
- `box.get_top()` and `box.get_bottom()` respectively get the top and bottom y-coordinates of the `id` box.

The example below draws boxes to visualize hypothetical price ranges over a period of `length` bars. At the start of each new period, it uses the average candle range multiplied by the `scaleFactor` input to calculate the corner points of a box centered at the `hl2` price with an `initialRange` height. After drawing the first box, it creates `numberOfBoxes - 1` new boxes inside a `for` loop.

Within each loop iteration, the script gets the `lastBoxDrawn` by retrieving the `last` element from the read-only `box.all` array, then calls `box.get_top()` and `box.get_bottom()` to get its y-coordinates. It uses these values to calculate the coordinates for a new box that's `scaleFactor` times taller than the previous:



```

1 // @version=5
2 indicator("Reading box values demo", "Nested boxes", overlay = true, max_boxes_count_
3   ↵ = 500)
4
5 // @variable The number of bars in the range calculation.
6 int length = input.int(10, "Length", 2, 500)
7 // @variable The number of nested boxes drawn on each period.
8 int numberOfBoxes = input.int(5, "Nested box count", 1)
9 // @variable The scale factor applied to each box.
10 float scaleFactor = input.float(1.6, "Scale factor", 1)
11
12 // @variable The initial box range.
13 float initialRange = scaleFactor * ta.sma(high - low, length)
14
15 if bar_index % length == 0
16   // @variable The top-left `chart.point` for the initial box. Does not contain_
17   ↵ `time` information.
18   topLeft = chart.point.from_index(bar_index, h12 + initialRange / 2)
19   // @variable The bottom-right `chart.point` for the initial box. Does not contain_
20   ↵ `time` information.
21   bottomRight = chart.point.from_index(bar_index + length, h12 - initialRange / 2)
22
23   // Calculate border and fill colors of the boxes.
24   borderColor = color.rgb(math.random(100, 255), math.random(0, 100), math.
25   ↵ random(100, 255))
26   bgColor = color.new(borderColor, math.max(100 * (1 - 1/numberOfBoxes), 90))
27
28   // Draw a new box using the `topLeft` and `bottomRight` points. Uses their_
29   ↵ `index` fields as x-coordinates.
30   box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor)
31
32   if numberOfBoxes > 1
33     // Loop to create additional boxes.
34     for i = 1 to numberOfBoxes - 1
35       // @variable The last box drawn by the script.
36       box lastBoxDrawn = box.all.last()

```

(continues on next page)

(continued from previous page)

```

33 // @variable The top price of the last box.
34 float top = box.get_top(lastBoxDrawn)
35 // @variable The bottom price of the last box.
36 float bottom = box.get_bottom(lastBoxDrawn)
37
38 // @variable The scaled range of the new box.
39 float newRange = scaleFactor * (top - bottom) * 0.5
40
41 // Update the `price` fields of the `topLeft` and `bottomRight` points.
42 // This does not affect the coordinates of previous boxes.
43 topLeft.price := hl2 + newRange
44 bottomRight.price := hl2 - newRange
45
46 // Draw a new box using the updated `topLeft` and `bottomRight` points.
47 box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor)

```

**Note that:**

- The `indicator()` function call uses `max_boxes_count = 500`, meaning the script can display up to 500 boxes on the chart.
- Each drawing has a `right` index length bars beyond the `left` index. Since the x-coordinates of these drawings can be up to 500 bars into the future, we've set the `maxval` of the `length` input to 500.
- On each new period, the script uses randomized `color.rgb()` values for the `border_color` and `bgcolor` of the boxes.
- Each `box.new()` call copies the coordinates from the `chart.point` objects assigned to the `topLeft` and `bottomRight` variables, which is why the script can modify their `price` fields on each loop iteration without affecting the other boxes.

**Cloning boxes**

To clone a specific box `i.d`, use `box.copy()`. This function copies the box and its properties. Any changes to the copied box do not affect the original.

For example, this script declares an `originalBox` variable on the first bar and assigns a `new box` to it once every `length` bars. On other bars, it uses `box.copy()` to create a `copiedBox` and calls `box.set_*()` functions to `modify` its properties. As shown on the chart below, these changes do not modify the `originalBox`:



```

1 // @version=5
2 indicator("Cloning boxes demo", overlay = true, max_boxes_count = 500)
3
4 // @variable The number of bars between each new mainLine assignment.
5 int length = input.int(20, "Length", 2)
6
7 // @variable The `chart.point` for the top-left of the `originalBox`. Contains `time` ↴
8 // and `index` information.
9 topLeft = chart.point.now(high)
10 // @variable The `chart.point` for the bottom-right of the `originalBox`. Does not ↴
11 // contain `time` information.
12 bottomRight = chart.point.from_index(bar_index + 1, low)
13
14 // @variable A new box with `topLeft` and `bottomRight` corners on every `length` bars.
15 var box originalBox = na
16
17 // @variable Is teal when the bar is rising, maroon when it's falling.
18 color originalColor = close > open ? color.teal : color.maroon
19
20 if bar_index % length == 0
21     // Assign a new box using the `topLeft` and `bottomRight` info to the ↴
22     // `originalBox`.
23     // This box uses the `index` fields from the points as x-coordinates.
24     originalBox := box.new(topLeft, bottomRight, originalColor, 2, bgcolor = color.
25     ↴new(originalColor, 60))
26 else
27     // @variable A clone of the `originalBox`.
28     box copiedBox = box.copy(originalBox)
29     // Modify the `copiedBox`. These changes do not affect the `originalBox`.
30     box.set_top(copiedBox, high)
31     box.set_bottom_right_point(copiedBox, bottomRight)
32     box.set_border_color(copiedBox, color.gray)
33     box.set_border_width(copiedBox, 1)
34     box.set_bgcolor(copiedBox, na)

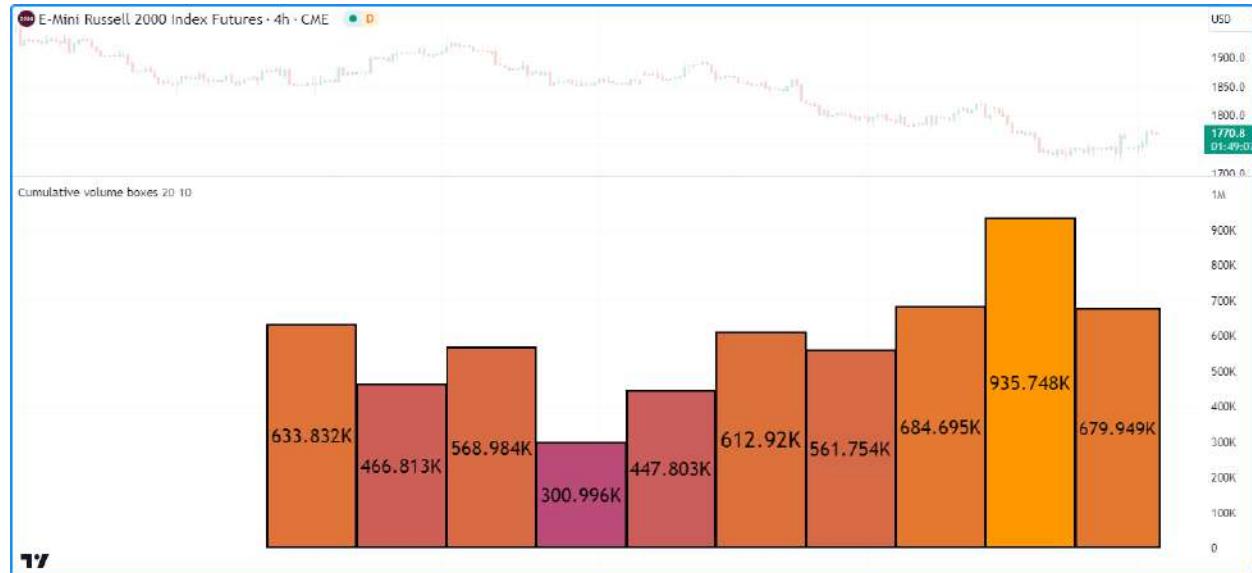
```

## Deleting boxes

To delete boxes drawn by a script, use `box.delete()`. As with `*.delete()` functions in other drawing namespaces, this function is handy for conditionally removing boxes or maintaining a specific number of boxes on the chart.

This example displays boxes representing periodic cumulative volume values. The script *creates* a new box ID and stores it in a `boxes` array once every `length` bars. If the `array's size` exceeds the specified `numberOfBoxes`, the script removes the oldest box from the array using `array.shift()` and deletes it using `box.delete()`.

On other bars, it accumulates `volume` over each period by *modifying* the `top` of the `last` box in the `boxes` array. The script then uses *for loops* to find the `highestTop` of all the array's boxes and set the `bgcolor` of each box with a gradient color based on its `box.get_top()` value relative to the `highestTop`:



```

1 //@version=5
2
3 //@variable The maximum number of boxes to show on the chart.
4 const int MAX_BOXES_COUNT = 500
5
6 indicator("Deleting boxes demo", "Cumulative volume boxes", format = format.volume,
7   max_boxes_count = MAX_BOXES_COUNT)
8
9 //@variable The number of bars in each period.
10 int length = input.int(20, "Length", 1)
11 //@variable The maximum number of volume boxes in the calculation.
12 int numberOfBoxes = input.int(10, "Number of boxes", 1, MAX_BOXES_COUNT)
13
14 //@variable An array containing the ID of each box displayed by the script.
15 var boxes = array.new<box>()
16
17 if bar_index % length == 0
18   // Push a new box into the `boxes` array. The box has the default `xloc.bar_
19   // index` property.
20   boxes.push(box.new(bar_index, 0, bar_index + 1, 0, #000000, 2, text_color =
21   // #000000))
22   // Shift the oldest box out of the array and delete it when the array's size_
23   // exceeds the `numberOfBoxes`.
24   if boxes.size() > numberOfBoxes
25     box.delete(boxes.shift())

```

(continues on next page)

(continued from previous page)

```

22 // @variable The last box drawn by the script as of the current chart bar.
23 box lastBox = boxes.last()
24 // Add the current bar's volume to the top of the `lastBox` and update the `right` ↵
25 // index.
26 lastBox.set_top(lastBox.get_top() + volume)
27 lastBox.set_right(bar_index + 1)
28 // Display the top of the `lastBox` as volume-formatted text.
29 lastBox.set_text(str.tostring(lastBox.get_top(), format.volume))
30
31 // @variable The highest `top` of all boxes in the `boxes` array.
32 float highestTop = 0.0
33 for id in boxes
34     highestTop := math.max(id.get_top(), highestTop)
35
36 // Set the `bgcolor` of each `id` in `boxes` with a gradient based on the ratio of ↵
37 // its `top` to the `highestTop`.
38 for id in boxes
39     id.set_bgcolor(color.from_gradient(id.get_top() / highestTop, 0, 1, color.purple, ↵
40     color.orange))

```

**Note that:**

- At the top of the code, we've declared a `MAX_BOXES_COUNT` variable with the “`const int`” *qualified type*. We use this value as the `max_boxes_count` in the `indicator()` function and the maximum possible value of the `numberOfBoxes` input.
- This script uses the second overload of the `box.new()` function, which specifies the box's `left`, `top`, `right`, and `bottom` coordinates separately.
- We've included `format.volume` as the `format` argument in the `indicator()` call, which tells the script that the y-axis of the chart pane represents *volume* values. Each box also displays its `top` value as `volume`-formatted text.

#### 4.12.4 Polylines

Pine Script™ polylines are **advanced** drawings that sequentially connect the coordinates from an array of `chart.point` instances using straight or *curved* line segments.

These powerful drawings can connect up to 10,000 points at any available location on the chart, allowing scripts to draw custom series, polygons, and other complex geometric formations that are otherwise difficult or impossible to draw using `line` or `box` objects.

The `polyline.*` namespace features the following built-ins for creating and managing `polyline` objects:

- The `polyline.new()` function creates a new polyline instance.
- The `polyline.delete()` function deletes an existing polyline instance.
- The `polyline.all` variable references a read-only `array` containing the IDs of all polylines displayed by the script. The array's `size` depends on the `max_polylines_count` of the `indicator()` or `strategy()` declaration statement and the number of polylines drawn by the script.

Unlike `lines` or `boxes`, polylines do not have functions for modification or reading their properties. To redraw a polyline on the chart, one can *delete* the existing instance and *create* a new polyline with the desired changes.

## Creating polylines

The `polyline.new()` function creates a new `polyline` instance to display on the chart. It has the following signature:

```
polyline.new(points, curved, closed, xloc, line_color, fill_color, line_style, line_
→width) → series polyline
```

The following eight parameters affect the behavior of a polyline drawing:

### `points`

Accepts an `array` of `chart.point` objects that determine the coordinates of each point in the polyline. The drawing connects the coordinates from each element in the `array` sequentially, starting from the *first*. Whether the polyline uses the `index` or `time` field from each `chart point` for its x-coordinates depends on the `xloc` value in the function call.

### `curved`

Specifies whether the drawing uses curved line segments to connect each `chart.point` in the `points` array. The default value is `false`, meaning it uses straight line segments.

### `closed`

Controls whether the polyline will connect the last `chart.point` in the `points` array to the first, forming a *closed polyline*. The default value is `false`.

### `xloc`

Specifies which field from each `chart.point` in the `points` array the polyline uses for its x-coordinates. When its value is `xloc.bar_index`, the function uses the `index` fields to create the polyline. When its value is `xloc.bar_time`, the function uses the `time` fields. The default value is `xloc.bar_index`.

### `line_color`

Specifies the color of all line segments in the polyline drawing. The default is `color.blue`.

### `fill_color`

Controls the color of the closed space filled by the polyline drawing. Its default value is `na`.

### `line_style`

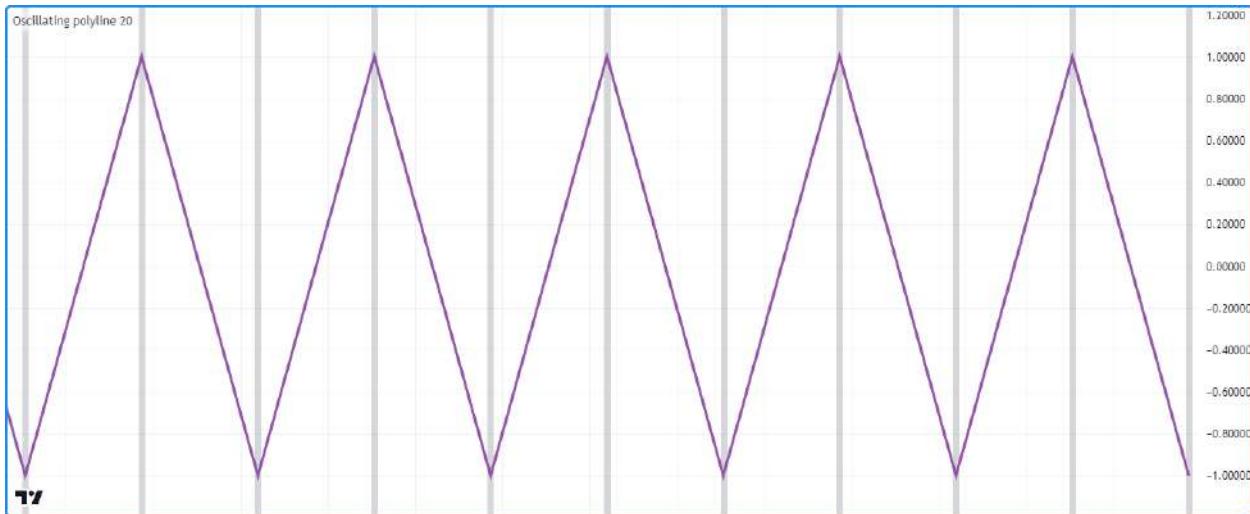
Specifies the style of the polyline, which can be any of the available options in the *Line styles* section of this page. The default is `line.style_solid`.

### `line_width`

Specifies the width of the polyline, in pixels. The default value is 1.

This script demonstrates a simple example of drawing a polyline on the chart. It pushes a new `chart.point` with an alternating price value into a `points` array and colors the background with `bgcolor()` once every length bars.

On the `last confirmed historical bar`, the script draws a new polyline on the chart, connecting the coordinates from each `chart point` in the `array`, starting from the first:



```

1 // @version=5
2 indicator("Creating polylines demo", "Oscillating polyline")
3
4 //@variable The number of bars between each point in the drawing.
5 int length = input.int(20, "Length between points", 2)
6
7 //@variable An array of `chart.point` objects to sequentially connect with a polyline.
8 var points = array.new<chart.point>()
9
10 //@variable The y-coordinate of each point in the `points`. Alternates between 1 and -1 on each `newPoint`.
11 var int yValue = 1
12
13 //@variable Is `true` once every `length` bars, `false` otherwise.
14 bool newPoint = bar_index % length == 0
15
16 if newPoint
17     // Push a new `chart.point` into the `points`. The new point contains `time` and `index` info.
18     points.push(chart.point.now(yValue))
19     // Change the sign of the `yValue`.
20     yValue *= -1
21
22 // Draw a new `polyline` on the last confirmed historical chart bar.
23 // The polyline uses the `time` field from each `chart.point` in the `points` array as x-coordinates.
24 if barstate.islastconfirmedhistory
25     polyline.new(points, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3)
26
27 // Highlight the chart background on every `newPoint` condition.
28 bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

```

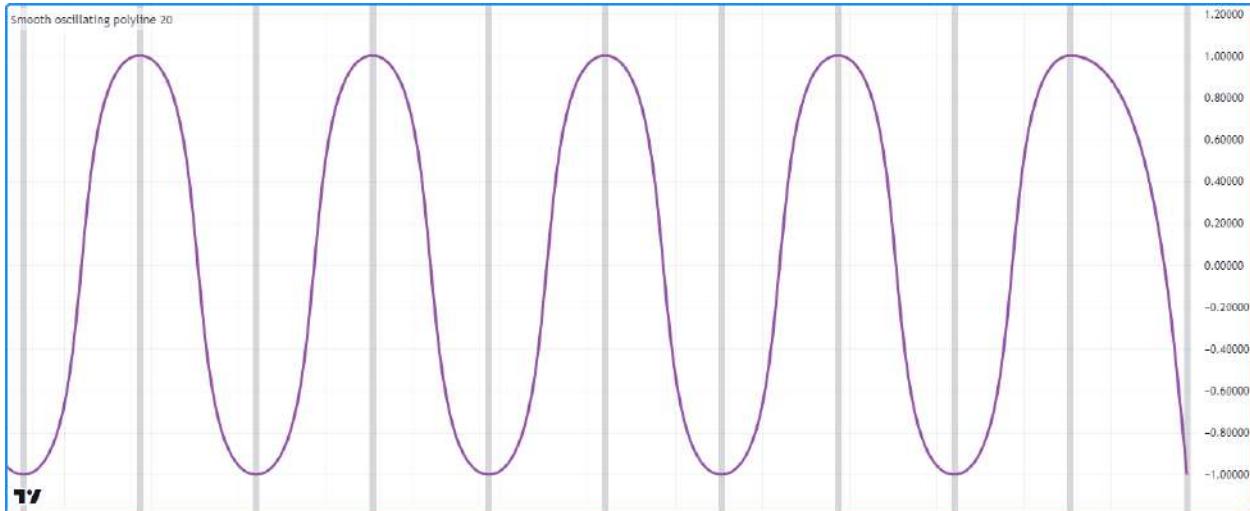
**Note that:**

- This script uses only *one* polyline to connect each *chart point* from the *array* with straight line segments, and this drawing spans throughout the available chart data, starting from the first bar.
- While one can achieve a similar effect using *lines*, doing so would require a new *line* instance on each occurrence of the *newPoint* condition, and such a drawing would be limited to a maximum of 500 line segments. This single unclosed polyline drawing, on the other hand, can contain up to 9,999 line segments.

## Curved drawings

Polyline can draw *curves* that are otherwise impossible to produce with *lines* or *boxes*. When enabling the `curved` parameter of the `polyline.new()` function, the resulting polyline interpolates *nonlinear* values between the coordinates from each `chart.point` in its `array` of points to generate a curvy effect.

For instance, the “Oscillating polyline” script in our previous example uses *straight* line segments to produce a drawing resembling a triangle wave, meaning a waveform that zig-zags between its peaks and valleys. If we set the `curved` parameter in the `polyline.new()` call from that example to `true`, the resulting drawing would connect the points using *curved* segments, producing a smooth, nonlinear shape similar to a sine wave:



```

1 // @version=5
2 indicator("Curved drawings demo", "Smooth oscillating polyline")
3
4 // @variable The number of bars between each point in the drawing.
5 int length = input.int(20, "Length between points", 2)
6
7 // @variable An array of `chart.point` objects to sequentially connect with a polyline.
8 var points = array.new<chart.point>()
9
10 // @variable The y-coordinate of each point in the `points`. Alternates between 1 and -1 on each `newPoint`.
11 var int yValue = 1
12
13 // @variable Is `true` once every `length` bars, `false` otherwise.
14 bool newPoint = bar_index % length == 0
15
16 if newPoint
17     // Push a new `chart.point` into the `points`. The new point contains `time` and `index` info.
18     points.push(chart.point.now(yValue))
19     // Change the sign of the `yValue`.
20     yValue *= -1
21
22 // Draw a new curved `polyline` on the last confirmed historical chart bar.
23 // The polyline uses the `time` field from each `chart.point` in the `points` array as x-coordinates.
24 if barstate.islastconfirmedhistory
25     polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6,

```

(continues on next page)

(continued from previous page)

```

26   ↵line_width = 3)
27 // Highlight the chart background on every `newPoint` condition.
28 bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

```

Notice that in this example, the smooth curves have relatively consistent behavior, and no portion of the drawing extends past its defined coordinates, which is not always the case when drawing curved polylines. The data used to construct a polyline heavily impacts the smooth, piecewise function it interpolates between its points. In some cases, the interpolated curve *can* reach beyond its actual coordinates.

Let's add some variation to the `chart.points` in our example's `points` array to demonstrate this behavior. In the version below, the script multiplies the `yValue` by a random value in the `chart.point.now()` calls.

To visualize the behavior, this script also creates a horizontal `line` at the `price` value from each `chart.point` in the `points` array, and it displays another polyline connecting the same points with straight line segments. As we see on the chart, both polylines pass through all coordinates from the `points` array. However, the curvy polyline occasionally reaches *beyond* the vertical boundaries indicated by the horizontal `lines`, whereas the polyline drawn using straight segments does not:



```

1 // @version=5
2 indicator("Curved drawings demo", "Random oscillating polylines")
3
4 // @variable The number of bars between each point in the drawing.
5 int length = input.int(20, "Length between points", 2)
6
7 // @variable An array of `chart.point` objects to sequentially connect with a polyline.
8 var points = array.new<chart.point>()
9
10 // @variable The sign of each `price` in the `points`. Alternates between 1 and -1 on
11 // each `newPoint`.
12 var int yValue = 1
13
14 // @variable Is `true` once every `length` bars.
15 bool newPoint = bar_index % length == 0
16
17 if newPoint
18     // Push a new `chart.point` with a randomized `price` into the `points`.
        // The new point contains `time` and `index` info.

```

(continues on next page)

(continued from previous page)

```

19 points.push(chart.point.now(yValue * math.random()))
20 // Change the sign of the `yValue`.
21 yValue *= -1
22
23 //@variable The newest `chart.point`.
24 lastPoint = points.last()
25 // Draw a horizontal line at the `lastPoint.price`. This line uses the default
26 // `xloc.bar_index`.
27 line.new(lastPoint.index - length, lastPoint.price, lastPoint.index + length,
28 // lastPoint.price, color = color.red)
29
30 // Draw two `polyline` instances on the last confirmed chart bar.
31 // Both polylines use the `time` field from each `chart.point` in the `points` array
32 // as x-coordinates.
33 if barstate.islastconfirmedhistory
34     polyline.new(points, curved = false, xloc = xloc.bar_time, line_color = #EB8A3B,
35 // line_width = 2)
36     polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6,
37 // line_width = 3)
38
39 // Highlight the chart background on every `newPoint` condition.
40 bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

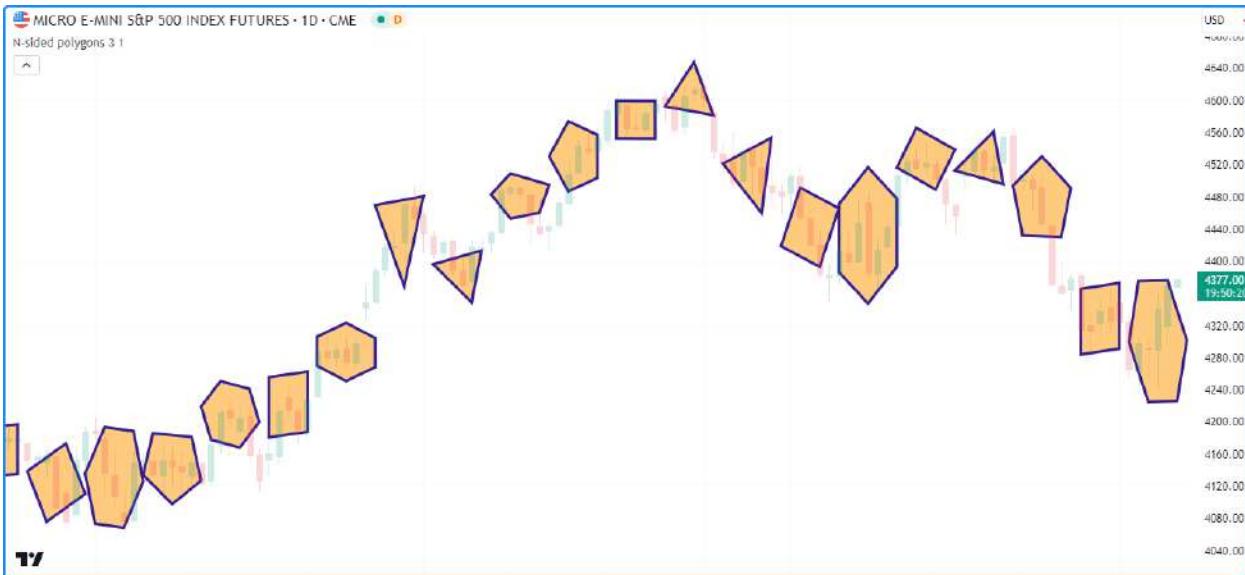
```

## Closed shapes

Since a single polyline can contain numerous straight or curved line segments, and the `closed` parameter allows the drawing to connect the coordinates from the first and last `chart.point` in its `array` of `points`, we can use polylines to draw many different types of closed polygonal shapes.

Let's draw some polygons in Pine. The following script periodically draws randomized polygons centered at `hl2` price values.

On each occurrence of the `newPolygon` condition, it `clears` the `points` array, calculates the `numberOfSides` and `rotationOffset` of the new polygon drawing based on `math.random()` values, then uses a `for loop` to push `numberOfSides` new `chart points` into the `array` that contain stepped coordinates from an elliptical path with `xScale` and `yScale` semi-axes. The script draws the polygon by connecting each `chart.point` from the `points` array using a `closed polyline` with straight line segments:



```

1 // @version=5
2 indicator("Closed shapes demo", "N-sided polygons", true)
3
4 // @variable The size of the horizontal semi-axis.
5 float xScale = input.float(3.0, "X scale", 1.0)
6 // @variable The size of the vertical semi-axis.
7 float yScale = input.float(1.0, "Y scale") * ta.atr(2)
8
9 // @variable An array of `chart.point` objects containing vertex coordinates.
10 var points = array.new<chart.point>()
11
12 // @variable The condition that triggers a new polygon drawing. Based on the
13 // horizontal axis to prevent overlaps.
14 bool newPolygon = bar_index % int(math.round(2 * xScale)) == 0 and barstate.
15 //isconfirmed
16
17 if newPolygon
18     // Clear the `points` array.
19     points.clear()
20
21     // @variable The number of sides and vertices in the new polygon.
22     int numberOfSides = int(math.random(3, 7))
23     // @variable A random rotation offset applied to the new polygon, in radians.
24     float rotationOffset = math.random(0.0, 2.0) * math.pi
25     // @variable The size of the angle between each vertex, in radians.
26     float step = 2 * math.pi / numberOfSides
27
28     // @variable The counter-clockwise rotation angle of each vertex.
29     float angle = rotationOffset
30
31     for i = 1 to numberOfSides
32         // @variable The approximate x-coordinate from an ellipse at the `angle`,_
33         // rounded to the nearest integer.
34         int xValue = int(math.round(xScale * math.cos(angle))) + bar_index
35         // @variable The y-coordinate from an ellipse at the `angle`.
36         float yValue = yScale * math.sin(angle) + hl2

```

(continues on next page)

(continued from previous page)

```

35     // Push a new `chart.point` containing the `xValue` and `yValue` into the
36     // `points` array.
37     // The new point does not contain `time` information.
38     points.push(chart.point.from_index(xValue, yValue))
39     // Add the `step` to the `angle`.
40     angle += step
41
42     // Draw a closed polyline connecting the `points`.
43     // The polyline uses the `index` field from each `chart.point` in the `points`_
44     // array.
45     polyline.new(
46         points, closed = true, line_color = color.navy, fill_color = color.new(color.
47         orange, 50), line_width = 3
48     )

```

**Note that:**

- This example shows the last ~50 polylines on the chart, as we have not specified a `max_polyline_count` value in the `indicator()` function call.
- The `yScale` calculation multiplies an `input.float()` by `ta.atr(2)` to adapt the vertical scale of the drawings to recent price ranges.
- The resulting polygons have a maximum width of twice the horizontal semi-axis (`2 * xScale`), rounded to the nearest integer. The `newPolygon` condition uses this value to prevent the polygon drawings from overlapping.
- The script rounds the `xValue` calculation to the nearest integer because the `index` field of a `chart.point` only accepts an `int` value, as the x-axis of the chart does not include fractional bar indices.

## Deleting polylines

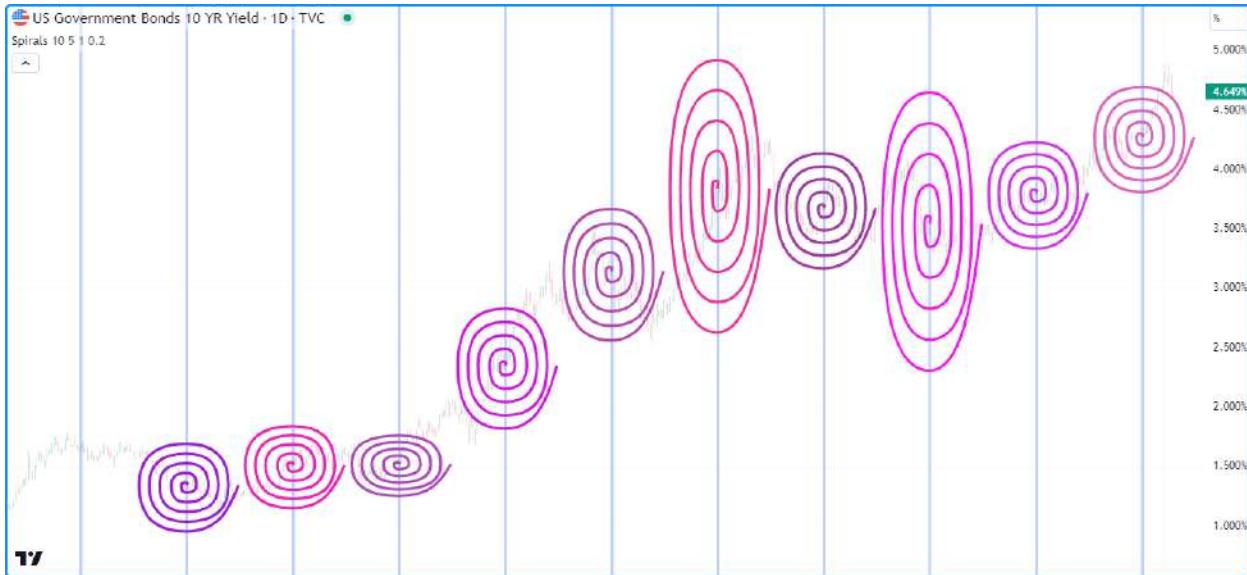
To delete a specific polyline `id`, use `polyline.delete()`. This function removes the `polyline` object from the script and its drawing on the chart.

As with other drawing objects, we can use `polyline.delete()` to maintain a specific number of polyline drawings or conditionally remove drawings from a chart.

For example, the script below periodically draws approximate arithmetic spirals and stores their polyline IDs in an `array`, which it *uses as a queue* to manage the number of drawings it displays.

When the `newSpiral` condition occurs, the script creates a `points` array and adds *chart points* within a `for loop`. On each loop iteration, it calls the `spiralPoint()` *user-defined function* to create a new `chart.point` containing stepped values from an elliptical path that grows with respect to the angle. The script then creates a randomly colored *curved polyline* connecting the coordinates from the `points` and pushes its ID into the `polylines` array.

When the array's `size` exceeds the specified `numberOfSpirals`, the script removes the oldest polyline using `array.shift()` and deletes the object using `polyline.delete()`:



```

1 // @version=5
2
3 //@variable The maximum number of polylines allowed on the chart.
4 const int MAX_POLYLINES_COUNT = 100
5
6 indicator("Deleting polylines example", "Spirals", true, max_polyline_count = MAX_
7 →POLYLINES_COUNT)
8
9 //@variable The number of spiral drawings on the chart.
10 int numberOfSpirals = input.int(10, "Spirals shown", 1, MAX_POLYLINES_COUNT)
11 //@variable The number of full spiral rotations to draw.
12 int rotations = input.int(5, "Rotations", 1)
13 //@variable The scale of the horizontal semi-axis.
14 float xScale = input.float(1.0, "X scale")
15 //@variable The scale of the vertical semi-axis.
16 float yScale = input.float(0.2, "Y scale") * ta.atr(2)
17
18 //@function Calculates an approximate point from an elliptically-scaled arithmetic_
19 →spiral.
20 //@returns A `chart.point` with `index` and `price` information.
21 spiralPoint(float angle, int xOffset, float yOffset) =>
22     result = chart.point.from_index(
23         int(math.round(angle * xScale * math.cos(angle))) + xOffset,
24         angle * yScale * math.sin(angle) + yOffset
25     )
26
27 //@variable An array of polylines.
28 var polylines = array.new<polyline>()
29
30 //@variable The condition to create a new spiral.
31 bool newSpiral = bar_index % int(math.round(4 * math.pi * rotations * xScale)) == 0
32
33 if newSpiral
34     //@variable An array of `chart.point` objects for the `spiral` drawing.
35     points = array.new<chart.point>()
36     //@variable The counter-clockwise angle between calculated points, in radians.
37     float step = math.pi / 2

```

(continues on next page)

(continued from previous page)

```

36 // @variable The rotation angle of each calculated point on the spiral, in radians.
37 float theta = 0.0
38 // Loop to create the spiral's points. Creates 4 points per full rotation.
39 for i = 0 to rotations * 4
40     // @variable A new point on the calculated spiral.
41     chart.point newPoint = spiralPoint(theta, bar_index, ohlc4)
42     // Add the `newPoint` to the `points` array.
43     points.push(newPoint)
44     // Add the `step` to the `theta` angle.
45     theta += step
46
47     // @variable A random color for the new `spiral` drawing.
48     color spiralColor = color.rgb(math.random(150, 255), math.random(0, 100), math.
49     ↪random(150, 255))
49     // @variable A new polyline connecting the spiral points. Uses the `index` field
50     ↪from each point as x-coordinates.
50     polyline spiral = polyline.new(points, true, line_color = spiralColor, line_width
51     ↪= 3)
51
52     // Push the new `spiral` into the `polylines` array.
53     polylines.push(spiral)
54     // Shift the first polyline out of the array and delete it when the array's size
54     ↪exceeds the `numberOfSpirals`.
55     if polylines.size() > numberOfSpirals
56         polyline.delete(polylines.shift())
57
58 // Highlight the background when `newSpiral` is `true`.
59 bgcolor(newSpiral ? color.new(color.blue, 70) : na, title = "New drawing highlight")

```

**Note that:**

- We declared a MAX\_POLYLINES\_COUNT global variable with a constant value of 100. The script uses this constant as the max\_polyline\_count value in the `indicator()` function and the maxval of the `numberOfSpirals` input.
- As with our “N-sided polygons” example in the [previous section](#), we round the calculation of x-coordinates to the nearest integer since the `index` field of a `chart.point` can only accept an `int` value.
- Despite the smooth appearance of the drawings, each polyline’s `points` array only contains *four* `chart.point` objects per spiral rotation. Since the `polyline.new()` call includes `curved = true`, each polyline uses *smooth curves* to connect their `points`, producing a visual approximation of the spiral’s actual curvature.
- The width of each spiral is approximately `4 * math.pi * rotations * xScale`, rounded to the nearest integer. We use this value in the `newSpiral` condition to space each drawing and prevent overlaps.

**Redrawing polylines**

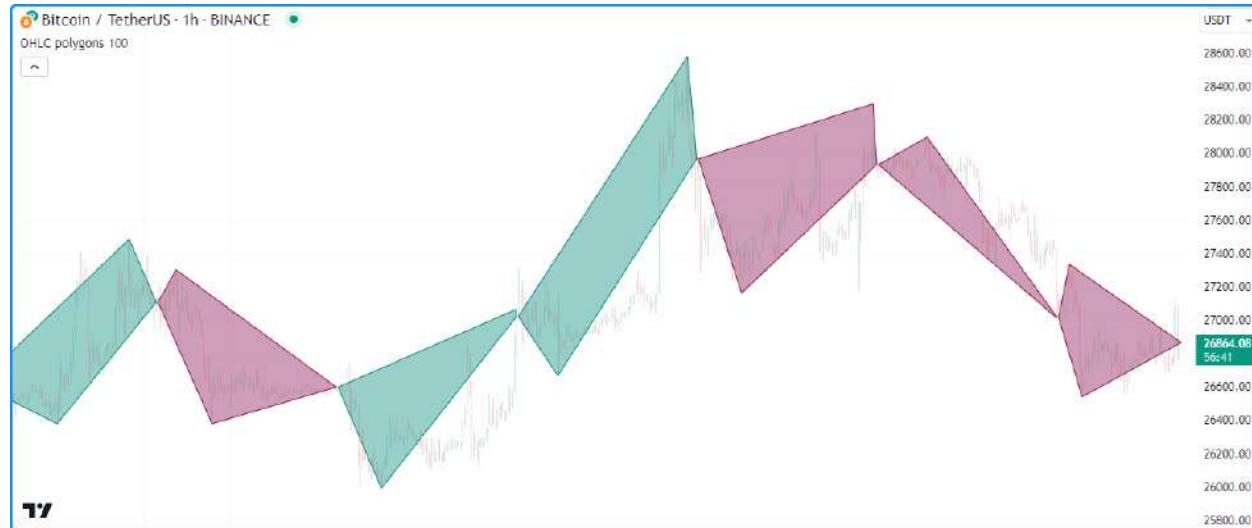
It may be desirable in some cases to change a polyline drawing throughout a script’s execution. While the `polyline.*` namespace does not contain built-in setter functions, we can *redraw* polylines referenced by variables or *collections* by *deleting* the existing polylines and assigning *new instances* with the desired changes.

The following example uses `polyline.delete()` and `polyline.new()` calls to update the value of a polyline variable.

This script draws closed polylines that connect the open, high, low, and close points of periods containing length bars. It creates a `currentDrawing` variable on the first bar and assigns a polyline ID to it on every chart bar. It uses the `openPoint`, `highPoint`, `lowPoint`, and `closePoint` variables to reference `chart points` that track the period’s developing OHLC values. As new values emerge, the script assigns new `chart.point` objects to the variables, collects them

in an `array` using `array.from`, then creates a `new polyline` connecting the coordinates from the array's points and assigns it to the `currentDrawing`.

When the `newPeriod` condition is `false` (i.e., the current period is not complete), the script *deletes* the polyline referenced by the `currentDrawing` before *creating a new one*, resulting in a dynamic drawing that changes over the developing period:



```

1 //@version=5
2 indicator("Redrawing polylines demo", "OHLC polygons", true, max_polylines_count =_
3             ↴100)
4
5 //@variable The length of the period.
6 int length = input.int(100, "Length", 1)
7
8 var chart.point openPoint = na
9 //@variable A `chart.point` representing the highest point of each period.
10 var chart.point highPoint = na
11 //@variable A `chart.point` representing the lowest point of each period.
12 var chart.point lowPoint = na
13 //@variable A `chart.point` representing the current bar's closing point.
14 closePoint = chart.point.now(close)
15
16 //@variable The current period's polyline drawing.
17 var polyline currentDrawing = na
18
19 //@variable Is `true` once every `length` bars.
20 bool newPeriod = bar_index % length == 0
21
22 if newPeriod
23     // Assign new chart points to the `openPoint`, `highPoint`, and `closePoint`.
24     openPoint := chart.point.now(open)
25     highPoint := chart.point.now(high)
26     lowPoint := chart.point.now(low)
27 else
28     // Assign a new `chart.point` to the `highPoint` when the `high` is greater than_
29     // its `price`.
30     if high > highPoint.price
            highPoint := chart.point.now(high)

```

(continues on next page)

(continued from previous page)

```

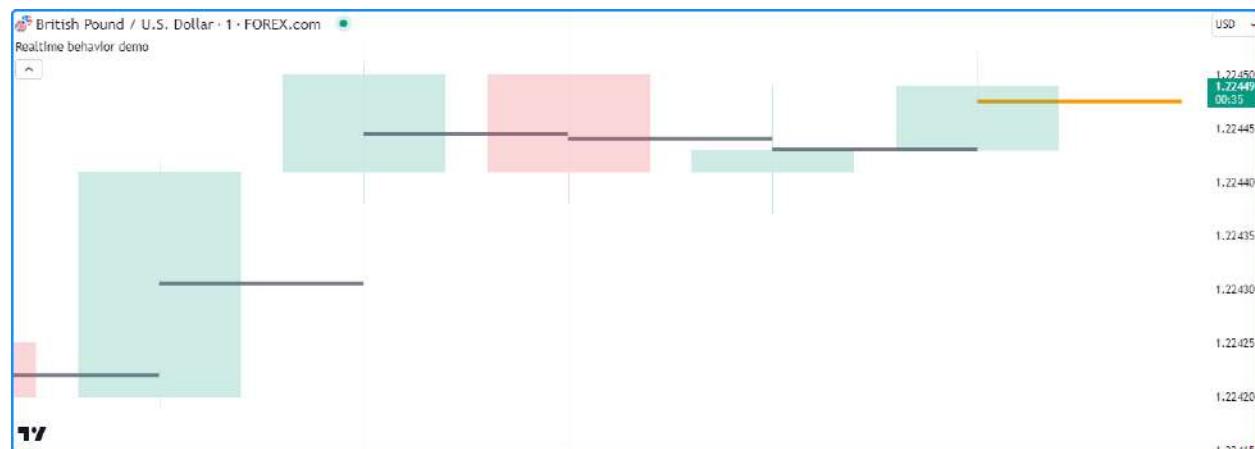
31 // Assign a new `chart.point` to the `lowPoint` when the `low` is less than its
32 // `price`.
33 if low < lowPoint.price
34     lowPoint := chart.point.now(low)
35
36 // @variable Is teal when the `closePoint.price` is greater than the `openPoint.price`,
37 // maroon otherwise.
38 color drawingColor = closePoint.price > openPoint.price ? color.teal : color.maroon
39
40 // Delete the polyline assigned to the `currentDrawing` if it's not a `newPeriod`.
41 if not newPeriod
42     polyline.delete(currentDrawing)
43 // Assign a new polyline to the `currentDrawing`.
44 // Uses the `index` field from each `chart.point` in its array as x-coordinates.
45 currentDrawing := polyline.new(
46     array.from(openPoint, highPoint, closePoint, lowPoint), closed = true,
47     line_color = drawingColor, fill_color = color.new(drawingColor, 60)
48 )

```

#### 4.12.5 Realtime behavior

*Lines*, *boxes*, and *polylines* are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes on a realtime bar. See the page on Pine Script™'s *Execution model*.

This script demonstrates the effect of rollback when it executes on the realtime, *unconfirmed* chart bar:



The `line.new()` call in this example creates a new `line` ID on each iteration when values change on the unconfirmed bar. The script automatically deletes the objects created on each change in that bar because of the *rollback* before each iteration. It only *commits* the last line created before the bar closes, and that `line` instance is the one that persists on the confirmed bar.

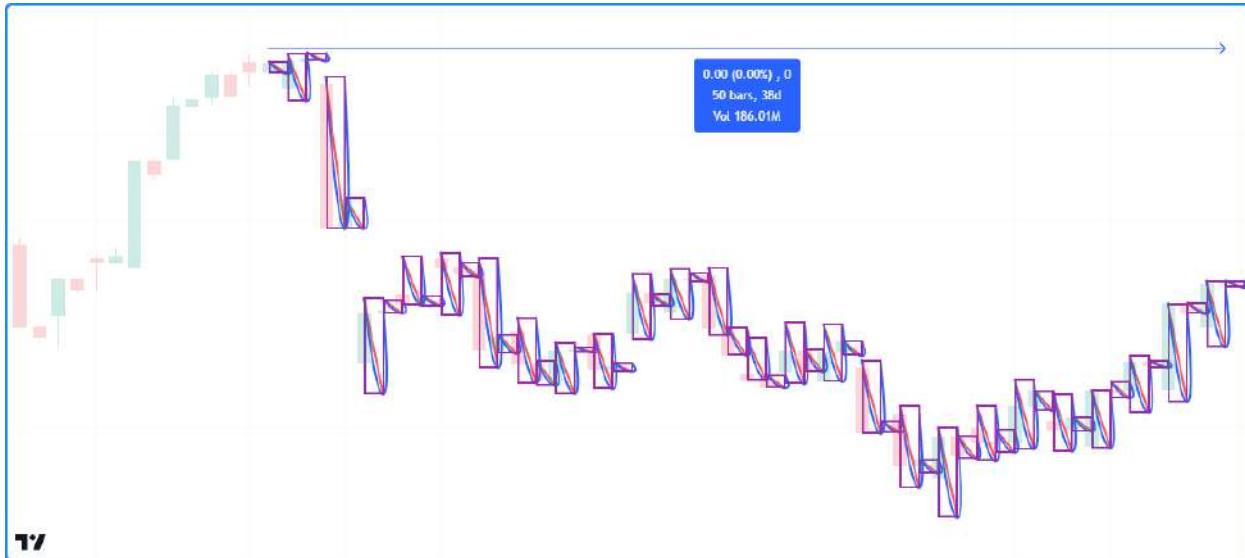
## 4.12.6 Limitations

### Total number of objects

*Lines*, *boxes*, and *polylines* consume server resources, which is why there are limits on the total number of drawings per script. When a script creates more drawing objects than the allowed limit, the Pine Script™ runtime automatically deletes the oldest ones in a process referred to as *garbage collection*.

A single script can contain up to 500 lines, 500 boxes, and 100 polylines. Users can control the garbage collection limits by specifying the `max_lines_count`, `max_boxes_count`, and `max_polyline_count` values in their script's `indicator()` or `strategy()` declaration statement.

This script demonstrates how garbage collection works in Pine. It creates a new line, box, and polyline on each chart bar. We haven't specified values for the `max_lines_count`, `max_boxes_count`, or `max_polyline_count` parameters in the `indicator()` function call, so the script will maintain the most recent ~50 lines, boxes, and polylines on the chart, as this is the default setting for each parameter:



```

1 // @version=5
2 indicator("Garbage collection demo", overlay = true)
3
4 // @variable A new `chart.point` at the current `bar_index` and `high`.
5 firstPoint = chart.point.now(high)
6 // @variable A new `chart.point` one bar into the future at the current `low`.
7 secondPoint = chart.point.from_index(bar_index + 1, low)
8 // @variable A new `chart.point` one bar into the future at the current `high`.
9 thirdPoint = chart.point.from_index(bar_index + 1, high)
10
11 // Draw a new `line` connecting the `firstPoint` to the `secondPoint`.
12 line.new(firstPoint, secondPoint, color = color.red, width = 2)
13 // Draw a new `box` with the `firstPoint` top-left corner and `secondPoint` bottom-
14 // right corner.
15 box.new(firstPoint, secondPoint, color.purple, 2, bgcolor = na)
16 // Draw a new `polyline` connecting the `firstPoint`, `secondPoint`, and `thirdPoint` -
17 // sequentially.
18 polyline.new(array.from(firstPoint, secondPoint, thirdPoint), true, line_width = 2)

```

**Note that:**

- We've used TradingView's "Measure" drawing tool to measure the number of bars covered by the script's drawing objects.

### Future references with `xloc.bar\_index`

Objects positioned using `xloc.bar_index` can contain x-coordinates no further than 500 bars into the future.

### Other contexts

Scripts cannot use `lines`, `boxes`, or `polylines` in `request.*()` functions. Instances of these types can use the values from `request.*()` calls, but scripts can only create and draw them in the chart's context.

This limitation is also why drawing objects will not work when using the `timeframe` parameter in the `indicator()` declaration statement.

### Historical buffer and `max\_bars\_back`

Using `barstate.isrealtime` in combination with drawings may sometimes produce unexpected results. For example, the intention of this script is to ignore all historical bars and draw horizontal lines spanning 300 bars back on *realtime* bars:

```

1 // @version=5
2 indicator("Historical buffer demo", overlay = true)
3
4 //@variable A `chart.point` at the `bar_index` from 300 bars ago and current `close`.
5 firstPoint = chart.point.from_index(bar_index[300], close)
6 //@variable The current bar's `chart.point` containing the current `close`.
7 secondPoint = chart.point.now(close)
8
9 // Draw a new line on realtime bars.
10 if barstate.isrealtime
11     line.new(firstPoint, secondPoint)

```

However, it will fail at runtime and raise an error. The script fails because it cannot determine the buffer size for historical values of the underlying `time` series. Although the code doesn't contain the built-in `time` variable, the built-in `bar_index` uses the `time` series in its inner workings. Therefore, accessing the value of the `bar_index` from 300 bars back requires the history buffer of the `time` series to be at least 300 bars.

Pine Script™ includes a mechanism that detects the required historical buffer size automatically in most cases. It works by letting the script access historical values any number of bars back for a limited duration. In this script's case, using `barstate.isrealtime` to control the drawing of lines prevents it from accessing the historical series, so it cannot infer the required historical buffer size, and the script fails.

The simple solution to this issue is to use the `max_bars_back()` function to *explicitly define* the historical buffer of the `time` series before evaluating the `conditional structure`:

Such issues can be confusing, but they're quite rare. The Pine Script™ team hopes to eliminate them over time.





## 4.13 Non-standard charts data

- *Introduction*
- `'ticker.heikinashi()'`
- `'ticker.renko()'`
- `'ticker.linebreak()'`
- `'ticker.kagi()'`
- `'ticker.pointfigure()'`

### 4.13.1 Introduction

These functions allow scripts to fetch information from non-standard bars or chart types, regardless of the type of chart the script is running on. They are: `ticker.heikinashi()`, `ticker.renko()`, `ticker.linebreak()`, `ticker.kagi()` and `ticker.pointfigure()`. All of them work in the same manner; they create a special ticker identifier to be used as the first argument in a `request.security()` function call.

### 4.13.2 `'ticker.heikinashi()'`

*Heikin-Ashi* means *average bar* in Japanese. The open/high/low/close values of Heikin-Ashi candlesticks are synthetic; they are not actual market prices. They are calculated by averaging combinations of real OHLC values from the current and previous bar. The calculations used make Heikin-Ashi bars less noisy than normal candlesticks. They can be useful to make visual assessments, but are unsuited to backtesting or automated trading, as orders execute on market prices — not Heikin-Ashi prices.

The `ticker.heikinashi()` function creates a special ticker identifier for requesting Heikin-Ashi data with the `request.security()` function.

This script requests the close value of Heikin-Ashi bars and plots them on top of the normal candlesticks:



```

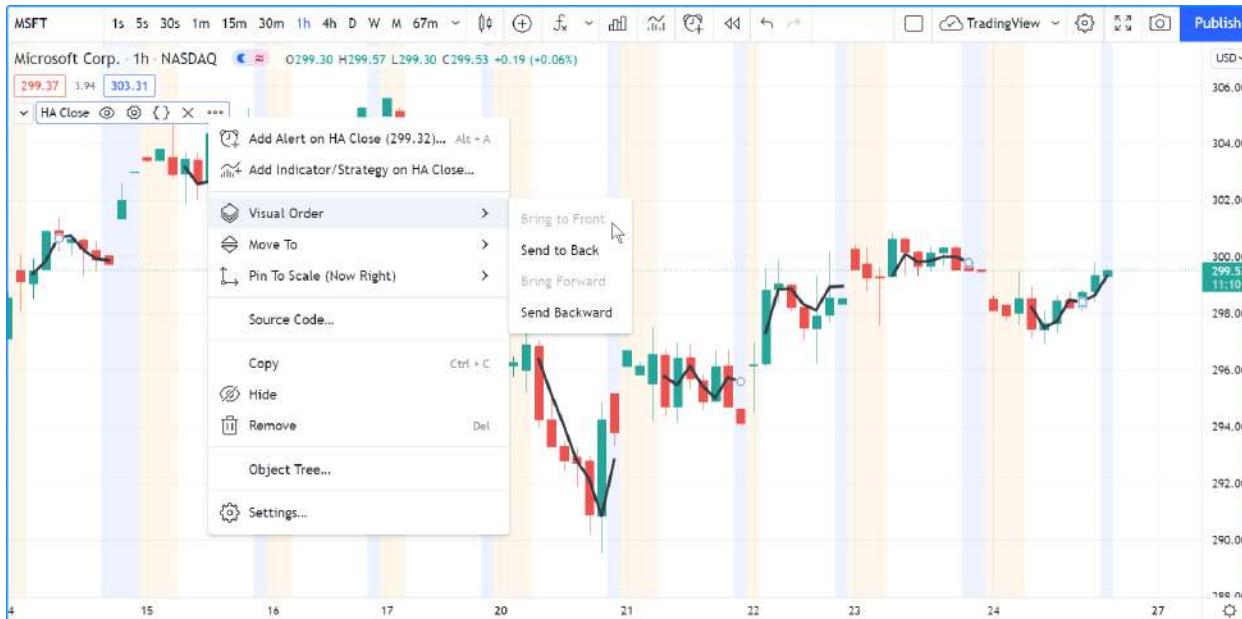
1 //@version=5
2 indicator("HA Close", "", true)
3 haTicker = ticker.heikinashi(syminfo.tickerid)
4 haClose = request.security(haTicker, timeframe.period, close)
5 plot(haClose, "HA Close", color.black, 3)

```

Note that:

- The close values for Heikin-Ashi bars plotted as the black line are very different from those of real candles using market prices. They act more like a moving average.
- The black line appears over the chart bars because we have selected “Visual Order/Bring to Front” from the script’s “More” menu.

If you wanted to omit values for extended hours in the last example, an intermediary ticker without extended session information would need to be created first:



```

1 //@version=5
2 indicator("HA Close", "", true)
3 regularSessionTicker = ticker.new(syminfo.prefix, syminfo.ticker, session.regular)
4 haTicker = ticker.heikinashi(regularSessionTicker)
5 haClose = request.security(haTicker, timeframe.period, close, gaps = barmerge.gaps_on)
6 plot(haClose, "HA Close", color.black, 3, plot.style_linebr)

```

Note that:

- We use the `ticker.new()` function first, to create a ticker without extended session information.
- We use that ticker instead of `syminfo.tickerid` in our `ticker.heikinashi()` call.
- In our `request.security()` call, we set the `gaps` parameter's value to `barmerge.gaps_on`. This instructs the function not to use previous values to fill slots where data is absent. This makes it possible for it to return `na` values outside of regular sessions.
- To be able to see this on the chart, we also need to use a special `plot.style_linebr` style, which breaks the plots on `na` values.

This script plots Heikin-Ashi candles under the chart:



```

1 //@version=5
2 indicator("Heikin-Ashi candles")
3 CANDLE_GREEN = #26A69A
4 CANDLE_RED   = #EF5350
5
6 haTicker = ticker.heikinashi(syminfo.tickerid)
7 [haO, haH, haL, haC] = request.security(haTicker, timeframe.period, [open, high, low, close])
8 candleColor = haC >= haO ? CANDLE_GREEN : CANDLE_RED
9 plotcandle(haO, haH, haL, haC, color = candleColor)

```

Note that:

- We use a `tuple` with `request.security()` to fetch four values with the same call.
- We use `plotcandle()` to plot our candles. See the [Bar plotting](#) page for more information.

### 4.13.3 `ticker.renko()`

*Renko* bars only plot price movements, without taking time or volume into consideration. They look like bricks stacked in adjacent columns<sup>1</sup>. A new brick is only drawn after the price passes the top or bottom by a predetermined amount. The `ticker.renko()` function creates a ticker id which can be used with `request.security()` to fetch Renko values, but there is no Pine Script™ function to draw Renko bars on the chart:

```

1 // @version=5
2 indicator("", "", true)
3 renkoTicker = ticker.renko(syminfo.tickerid, "ATR", 10)
4 renkoLow = request.security(renkoTicker, timeframe.period, low)
5 plot(renkoLow)

```

### 4.13.4 `ticker.linebreak()`

The *Line Break* chart type displays a series of vertical boxes that are based on price changes<sup>1</sup>. The `ticker.linebreak()` function creates a ticker id which can be used with `request.security()` to fetch “Line Break” values, but there is no Pine Script™ function to draw such bars on the chart:

```

1 // @version=5
2 indicator("", "", true)
3 lineBreakTicker = ticker.linebreak(syminfo.tickerid, 3)
4 lineBreakClose = request.security(lineBreakTicker, timeframe.period, close)
5 plot(lineBreakClose)

```

### 4.13.5 `ticker.kagi()`

*Kagi* charts are made of a continuous line that changes directions. The direction changes when the price changes<sup>1</sup> beyond a predetermined amount. The `ticker.kagi()` function creates a ticker id which can be used with `request.security()` to fetch “Kagi” values, but there is no Pine Script™ function to draw such bars on the chart:

```

1 // @version=5
2 indicator("", "", true)
3 kagiBreakTicker = ticker.linebreak(syminfo.tickerid, 3)
4 kagiBreakClose = request.security(kagiBreakTicker, timeframe.period, close)
5 plot(kagiBreakClose)

```

### 4.13.6 `ticker.pointfigure()`

*Point and Figure* (PnF) charts only plot price movements<sup>1</sup>, without taking time into consideration. A column of X’s is plotted as the price rises, and O’s are plotted when price drops. The `ticker.pointfigure()` function creates a ticker id which can be used with `request.security()` to fetch “PnF” values, but there is no Pine Script™ function to draw such bars on the chart. Every column of X’s or O’s is represented with four numbers. You may think of them as synthetic OHLC PnF values:

```

1 // @version=5
2 indicator("", "", true)
3 pnfTicker = ticker.pointfigure(syminfo.tickerid, "hl", "ATR", 14, 3)

```

(continues on next page)

<sup>1</sup> On TradingView, Renko, Line Break, Kagi and PnF chart types are generated from OHLC values from a lower timeframe. These chart types thus represent only an approximation of what they would be like if they were generated from tick data.

(continued from previous page)

```

4 [pnfO, pnFC] = request.security(pnfTicker, timeframe.period, [open, close], barmerge.
5   ↵gaps_on)
6 plot(pnfO, "PnF Open", color.green, 4, plot.style_linebr)
6 plot(pnfC, "PnF Close", color.red, 4, plot.style_linebr)

```



## 4.14 Other timeframes and data

- *Introduction*
- *Common characteristics*
- *Data feeds*
- `request.security()`
- `request.security\_lower\_tf()`
- *Custom contexts*
- *Historical and realtime behavior*
- `request.currency\_rate()`
- `request.dividends()`, `request.splits()`, and `request.earnings()`
- `request.quandl()`
- `request.financial()`
- `request.economic()`
- `request.seed()`

### 4.14.1 Introduction

Pine Script™ allows users to request data from sources and contexts other than those their charts use. The functions we present on this page can fetch data from a variety of alternative sources:

- *request.security()* retrieves data from another symbol, timeframe, or other context.
- *request.security\_lower\_tf()* retrieves *intrabar* data, i.e., data from a timeframe lower than the chart timeframe.
- *request.currency\_rate()* requests a *daily rate* to convert a value expressed in one currency to another.
- *request.dividends()*, *request.splits()*, and *request.earnings()* respectively retrieve information about an issuing company's dividends, splits, and earnings.

- `request.quandl()` retrieves information from NASDAQ Data Link (formerly Quandl).
- `request.financial()` retrieves financial data from FactSet.
- `request.economic()` retrieves economic and industry data.
- `request.seed()` retrieves data from a *user-maintained* GitHub repository.

---

**Note:** Throughout this page, and in other parts of our documentation that discuss `request.*()` functions, we often use the term “context” to describe the ticker ID, timeframe, and any modifications (price adjustments, session settings, non-standard chart types, etc.) that apply to a chart or the data retrieved by a script.

---

These are the signatures of the functions in the `request.*` namespace:

```
request.security(symbol, timeframe, expression, gaps, lookahead, ignore_invalid_
↳symbol, currency) → series <type>

request.security_lower_tf(symbol, timeframe, expression, ignore_invalid_symbol,_
↳currency, ignore_invalid_timeframe) → array<type>

request.currency_rate(from, to, ignore_invalid_currency) → series float

request.dividends(ticker, field, gaps, lookahead, ignore_invalid_symbol, currency) →_
↳series float

request.splits(ticker, field, gaps, lookahead, ignore_invalid_symbol) → series float

request.earnings(ticker, field, gaps, lookahead, ignore_invalid_symbol, currency) →_
↳series float

request.quandl(ticker, gaps, index, ignore_invalid_symbol) → series float

request.financial(symbol, financial_id, period, gaps, ignore_invalid_symbol,_
↳currency) → series float

request.economic(country_code, field, gaps, ignore_invalid_symbol) → series float

request.seed(source, symbol, expression, ignore_invalid_symbol) → series <type>
```

The `request.*()` family of functions has numerous potential applications. Throughout this page, we will discuss in detail these functions and some of their typical use cases.

---

**Note:** Users can also allow compatible scripts to evaluate their scopes in other contexts without requiring `request.*()` functions by using the `timeframe` parameter of the `indicator()` declaration statement.

---

## 4.14.2 Common characteristics

Many functions in the `request.*()` namespace share some common properties and parameters. Before we explore each function in depth, let's familiarize ourselves with these characteristics.

### Usage

All `request.*()` functions return “series” results, which means they can produce different values on every bar. However, most `request.*()` function parameters require “const”, “input”, or “simple” arguments.

In essence, Pine Script™ must determine the values of most arguments passed into a `request.*()` function upon compilation of the script or on the first chart bar, depending on the *qualified type* that each parameter accepts, and these values cannot change throughout the execution of the script. The only exception is the `expression` parameter in `request.security()`, `request.security_lower_tf()`, and `request.seed()`, which accepts “series” arguments.

Calls to `request.*()` functions execute on every chart bar, and scripts cannot selectively deactivate them throughout their execution. Scripts cannot call `request.*()` functions within the local scopes of *conditional structures*, *loops*, or functions and methods exported by *Libraries*, but they can use such function calls within the bodies of non-exported *user-defined functions* and *methods*.

When using any `request.*()` functions within a script, runtime performance is an important consideration. These functions can have a sizable impact on script performance. While scripts can contain a maximum of 40 calls to the `request.*()` namespace, users should strive to minimize the number of calls in their scripts to keep resource consumption as low as possible. For more information on the limitations of these functions, see *this* section of our User Manual’s page on Pine’s *limitations*.

### ‘gaps’

When using a `request.*()` function to retrieve data from another context, the data may not come in on each new bar as it would with the current chart. The `gaps` parameter of a `request.*()` function allows users to control how the function responds to nonexistent values in the requested series.

---

**Note:** When using the `indicator()` function to evaluate a script in another context, the `timeframe_gaps` parameter specifies how it handles nonexistent values. The parameter is similar to the `gaps` parameter for `request.*()` functions.

---

Suppose we have a script that requests hourly data for the chart’s symbol with `request.security()` executing on a 1-minute chart. In this case, the function call will only return new values on the 1-minute bars that cover the opening/closing times of the symbol’s hourly bars. On other chart bars, we can decide whether the function will return `na` values or the last available values via the `gaps` parameter.

When the `gaps` parameter uses `barmerge.gaps_on`, the function will return `na` results on all chart bars where new data isn’t yet confirmed from the requested context. Otherwise, when the parameter uses `barmerge.gaps_off`, the function will fill the gaps in the requested data with the last confirmed values on historical bars and the most recent developing values on realtime bars.

The script below demonstrates the difference in behavior by *plotting* the results from two `request.security()` calls that fetch the `close` price of the current symbol from the hourly timeframe on a 1-minute chart. The first call uses `gaps = barmerge.gaps_off` and the second uses `gaps = barmerge.gaps_on`:



```

1 // @version=5
2 indicator("gaps demo", overlay = true)
3
4 // @variable The `close` requested from the hourly timeframe without gaps.
5 float dataWithoutGaps = request.security(syminfo.tickerid, "60", close, gaps =
6   ↪ barmerge.gaps_off)
7 // @variable The `close` requested from the hourly timeframe with gaps.
8 float dataWithGaps = request.security(syminfo.tickerid, "60", close, gaps = barmerge.
9   ↪ gaps_on)
10
11 // Plot the requested data.
12 plot(dataWithoutGaps, "Data without gaps", color.blue, 3, plot.style_linebr)
13 plot(dataWithGaps, "Data with gaps", color.purple, 15, plot.style_linebr)
14
15 // Highlight the background for realtime bars.
16 bgcolor(barstate.isrealtime ? color.new(color.aqua, 70) : na, title = "Realtime bar
17   ↪ highlight")

```

#### Note that:

- `barmerge.gaps_off` is the default value for the `gaps` parameter in all applicable `request.*()` functions.
- The script plots the requested series as lines with breaks (`plot.style_linebr`), which don't bridge over `na` values as the default style (`plot.style_line`) does.
- When using `barmerge.gaps_off`, the `request.security()` function returns the last confirmed `close` from the hourly timeframe on all historical bars. When running on *realtime bars* (the bars with the `color.aqua` background in this example), it returns the symbol's current `close` value, regardless of confirmation. For more information, see the *Historical and realtime behavior* section of this page.

### `'ignore_invalid_symbol'`

The `ignore_invalid_symbol` parameter of `request.*()` functions determines how a function will handle invalid data requests, e.g.:

- Using a `request.*()` function with a nonexistent ticker ID as the `symbol/ticker` parameter.
- Using `request.financial()` to retrieve information that does not exist for the specified `symbol` or `period`.
- Using `request.economic()` to request a `field` that doesn't exist for a `country_code`.

A `request.*()` function call will produce a *runtime error* and halt the execution of the script when making an erroneous request if its `ignore_invalid_symbol` parameter is `false`. When this parameter's value is `true`, the function will return `na` values in such a case instead of raising an error.

This example uses `request.*()` calls within a *user-defined function* to retrieve data for estimating an instrument's market capitalization (market cap). The user-defined `calcMarketCap()` function calls `request.financial()` to retrieve the total shares outstanding for a symbol and `request.security()` to retrieve a tuple containing the symbol's `close` price and `currency`. We've included `ignore_invalid_symbol = true` in both of these `request.*()` calls to prevent runtime errors for invalid requests.

The script displays a `formatted string` representing the symbol's estimated market cap value and currency in a `table` on the chart and uses a `plot` to visualize the `marketCap` history:



```

1 // @version=5
2 indicator("ignore_invalid_symbol demo", "Market cap estimate", format = format.volume)
3
4 // @variable The symbol to request data from.
5 string symbol = input.symbol("TSX:SHOP", "Symbol")
6
7 // @function Estimates the market capitalization of the specified `tickerID` if the
8 // → data exists.
9 calcMarketCap(simple string tickerID) =>
10    // @variable The quarterly total shares outstanding for the `tickerID`. Returns
11    // → `na` when the data isn't available.
12    float tso = request.financial(tickerID, "TOTAL_SHARES_OUTSTANDING", "FQ", ignore_
13    // → invalid_symbol = true)
14    // @variable The `close` price and currency for the `tickerID`. Returns `[na, na]` →
15    // when the `tickerID` is invalid.

```

(continues on next page)

(continued from previous page)

```

12 [price, currency] = request.security(
13     tickerID, timeframe.period, [close, syminfo.currency], ignore_invalid_symbol_
14     ↵= true
15 )
16 // Return a tuple containing the market cap estimate and the quote currency.
17 [tso * price, currency]
18
19 // @variable A `table` object with a single cell that displays the `marketCap` and_
20 // `quoteCurrency`.
21 var table infoTable = table.new(position.top_right, 1, 1)
22 // Initialize the table's cell on the first bar.
23 if barstate.isfirst
24     table.cell(infoTable, 0, 0, "", text_color = color.white, text_size = size.huge,_
25     ↵bgcolor = color.teal)
26
27 // Get the market cap estimate and quote currency for the `symbol`.
28 [marketCap, quoteCurrency] = calcMarketCap(symbol)
29
30 // @variable The formatted text displayed inside the `infoTable`.
31 string tableText = str.format("Market cap:\n{0} {1}", str.tostring(marketCap, format.-
32     ↵volume), quoteCurrency)
33 // Update the `infoTable`.
34 table.cell_set_text(infoTable, 0, 0, tableText)
35
36 // Plot the `marketCap` value.
37 plot(marketCap, "Market cap", color.new(color.purple, 60), style = plot.style_area)

```

**Note that:**

- The `calcMarketCap()` function will only return values on valid instruments with total shares outstanding data, such as the one we've selected for this example. It will return a market cap value of `na` on others that don't have financial data, including forex, crypto, and derivatives.
- Not all issuing companies publish quarterly financial reports. If the `symbol`'s issuing company doesn't report on a quarterly basis, change the "FQ" value in this script to the company's minimum reporting period. See the [request.financial\(\)](#) section for more information.
- We've used `format.volume` in the `indicator()` and `str.tostring()` calls, which specify that the y-axis of the chart pane represents volume-formatted values and the "string" representation of the `marketCap` value shows as volume-formatted text.
- This script creates a `table` and initializes its cell on the `first` chart bar, then `updates the cell's text` on subsequent bars. To learn more about working with tables, see the [Tables](#) page of our User Manual.

**`currency`**

The `currency` parameter of a `request.*()` function allows users to specify the currency of the requested data. When this parameter's value differs from the `syminfo.currency` of the requested context, the function will convert the requested values to express them in the specified `currency`. This parameter can accept a built-in variable from the `currency.*` namespace, such as `currency.JPY`, or a "string" representing the ISO 4217 currency code (e.g., "JPY").

The conversion rate between the `syminfo.currency` of the requested data and the specified `currency` depends on the corresponding "FX\_IDC" daily rate from the previous day. If no available instrument provides the conversion rate directly, the function will use the value from a `spread symbol` to derive the rate.

---

**Note:** Not all `request.*()` function calls return values expressed as a currency amount. Therefore, currency con-

version is *not* always necessary. For example, some series returned by `request.financial()` are expressed in units other than currency, such as the “PIOTROSKI\_F\_SCORE” and “NUMBER\_OF\_EMPLOYEES” metrics. It is up to programmers to determine when currency conversion is appropriate in their data requests.

---

### ‘lookahead’

The `lookahead` parameter in `request.security()`, `request.dividends()`, `request.splits()`, and `request.earnings()` specifies the lookahead behavior of the function call. Its default value is `barmerge.lookahead_off`.

When requesting data from a higher-timeframe (HTF) context, the `lookahead` value determines whether the function can request values from times *beyond* those of the historical bars it executes on. In other words, the `lookahead` value determines whether the requested data may contain *lookahead bias* on historical bars.

When requesting data from a lower-timeframe (LTF) context, the `lookahead` parameter determines whether the function requests values from the first or last *intrabar* (LTF bar) on each chart bar.

**Programmers should exercise extreme caution when using lookahead in their scripts, namely when requesting data from higher timeframes.** When using `barmerge.lookahead_on` as the `lookahead` value, ensure that it does not compromise the integrity of the script’s logic by leaking *future* data into historical chart bars.

The following scenarios are cases where enabling lookahead is acceptable in a `request.*()` call:

- The expression in `request.security()` references a series with a *historical offset* (e.g., `close[1]`), which prevents the function from requesting future values that it would *not* have access to on a realtime basis.
- The specified `timeframe` in the call is the same as the chart the script executes on, i.e., `timeframe.period`.
- The function call requests data from an intrabar timeframe, i.e., a timeframe smaller than the `timeframe.period`. See [this section](#) for more information.

---

**Note:** Using `request.security()` to leak future data into the past is **misleading** and **not allowed** in script publications. While your script’s results on historical bars may look great due to its seemingly “magical” acquisition of prescience (which it will not be able to reproduce on realtime bars), you will be misleading yourself and the users of your script. If you [publish your script](#) to share it with others, ensure you **do not mislead users** by accessing future information on historical bars.

---

This example demonstrates how the `lookahead` parameter affects the behavior of higher-timeframe data requests and why enabling lookahead in `request.security()` without offsetting the expression is misleading. The script calls `request.security()` to get the HTF `high` price for the current chart’s symbol in three different ways and `plots` the resulting series on the chart for comparison.

The first call uses `barmerge.lookahead_off` (default), and the others use `barmerge.lookahead_on`. However, the third `request.security()` call also *offsets* its expression using the history-referencing operator `[]` to avoid leaking future data into the past.

As we see on the chart, the `plot` of the series requested using `barmerge.lookahead_on` without an offset (`fuchsia` line) shows final HTF `high` prices *before* they’re actually available on historical bars, whereas the other two calls do not:



```

1 // @version=5
2 indicator("lookahead demo", overlay = true)
3
4 //@variable The timeframe to request the data from.
5 string timeframe = input.timeframe("30", "Timeframe")
6
7 //@variable The requested `high` price from the current symbol on the `timeframe` ↴
8 // without lookahead bias.
9 // On realtime bars, it returns the current `high` of the `timeframe`.
10 float lookaheadOff = request.security(syminfo.tickerid, timeframe, high, lookahead = ↴
11 barmerge.lookahead_off)
12
13 //@variable The requested `high` price from the current symbol on the `timeframe` ↴
14 // with lookahead bias.
15 // Returns values that should NOT be accessible yet on historical bars.
16 float lookaheadOn = request.security(syminfo.tickerid, timeframe, high, lookahead = ↴
17 barmerge.lookahead_on)
18
19 //@variable The requested `high` price from the current symbol on the `timeframe` ↴
20 // without lookahead bias or repainting.
21 // Behaves the same on historical and realtime bars.
22 float lookaheadOnOffset = request.security(syminfo.tickerid, timeframe, high[1], ↴
23 //lookahead = barmerge.lookahead_on)
24
25 // Plot the values.
26 plot(lookaheadOff, "High, no lookahead bias", color.new(color.blue, 40), 5)
27 plot(lookaheadOn, "High with lookahead bias", color.fuchsia, 3)
28 plot(lookaheadOnOffset, "High, no lookahead bias or repaint", color.aqua, 3)
29 // Highlight the background on realtime bars.
30 bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar ↴
31 highlight")

```

#### Note that:

- The series requested using `barmerge.lookahead_off` has a new historical value at the *end* of each HTF period, and both series requested using `barmerge.lookahead_on` have new historical data at the *start* of each period.
- On realtime bars, the plot of the series without lookahead (blue) and the series with lookahead and no historical offset (fuchsia) show the *same value* (i.e., the HTF period's unconfirmed `high` price), as no data exists beyond

those points to leak into the past. Both of these plots will *repaint* their results after restarting the script's execution, as `realtime` bars will become `historical` bars.

- The series that uses lookahead and a historical offset (`aqua`) does not repaint its values, as it always references the last *confirmed* value from the higher timeframe. See the [Avoiding repainting](#) section of this page for more information.

---

**Note:** In Pine Script™ v1 and v2, the `security()` function did not include a `lookahead` parameter, but it behaved as it does in later versions of Pine with `lookahead = barmerge.lookahead_on`, meaning that it systematically used data from the future HTF context on historical bars. Therefore, users should *exercise caution* with Pine v1 or v2 scripts that use HTF `security()` calls unless the function calls contain historical offsets.

---

### 4.14.3 Data feeds

TradingView's data providers supply different data feeds that scripts can access to retrieve information about an instrument, including:

- Intraday historical data (for timeframes < 1D)
- End-of-day (EOD) historical data (for timeframes  $\geq$  1D)
- Realtime data (which may be delayed, depending on your account type and extra data services)
- Extended hours data

Not all of these data feed types exist for every instrument. For example, the symbol "BNC:BLX" only has EOD data available.

For some instruments with intraday and EOD historical feeds, volume data may not be the same since some trades (block trades, OTC trades, etc.) may only be available at the *end* of the trading day. Consequently, the EOD feed will include this volume data, but the intraday feed will not. Differences between EOD and intraday volume feeds are almost nonexistent for instruments such as cryptocurrencies, but they are commonplace in stocks.

Slight price discrepancies may also occur between EOD and intraday feeds. For example, the high value on one EOD bar may not match any intraday high values supplied by the data provider for that day.

Another distinction between EOD and intraday data feeds is that EOD feeds do not contain information from *extended hours*.

When retrieving information on realtime bars with `request.*()` functions, it's important to note that historical and realtime data reported for an instrument often rely on *different* data feeds. A broker/exchange may retroactively modify values reported on realtime bars, which the data will only reflect after refreshing the chart or restarting the execution of the script.

Another important consideration is that the chart's data feeds and feeds requested from providers by the script are managed by *independent*, concurrent processes. Consequently, in some *rare* cases, it's possible for races to occur where requested results temporarily fall out of synch with the chart on a realtime bar, which a script retroactively adjusts after restarting its execution.

These points may account for variations in the values retrieved by `request.*()` functions when requesting data from other contexts. They may also result in discrepancies between data received on realtime bars and historical bars. There are no steadfast rules about the variations one may encounter in their requested data feeds.

---

**Note:** As a rule, TradingView *does not* generate data; it relies on its data providers for the information displayed on charts and accessed by scripts.

---

When using data feeds requested from other contexts, it's also crucial to consider the *time axis* differences between the chart the script executes on and the requested feeds since `request.*()` functions adapt the returned series to the chart's time axis. For example, requesting "BTCUSD" data on the "SPY" chart with `request.security()` will only show new values when the "SPY" chart has new data as well. Since "SPY" is not a 24-hour symbol, the "BTCUSD" data returned will contain gaps that are otherwise not present when viewing its chart directly.

#### 4.14.4 `request.security()`

The `request.security()` function allows scripts to request data from other contexts than the chart the script executes on, such as:

- Other symbols, including `spread symbols`
- Other timeframes (see our User Manual's page on *Timeframes* to learn about timeframe specifications in Pine Script™)
- *Custom contexts*, including alternative sessions, price adjustments, chart types, etc. using `ticker.*()` functions

This is the function's signature:

```
request.security(symbol, timeframe, expression, gaps, lookahead, ignore_invalid_
    ↵symbol, currency) → series <type>
```

The `symbol` value is the ticker identifier representing the symbol to fetch data from. This parameter accepts values in any of the following formats:

- A "string" representing a symbol (e.g., "IBM" or "EURUSD") or an "*Exchange:Symbol*" pair (e.g., "NYSE:IBM" or "OANDA:EURUSD"). When the value does not contain an exchange prefix, the function selects the exchange automatically. We recommend specifying the exchange prefix when possible for consistent results. Users can also pass an empty string to this parameter, which prompts the function to use the current chart's symbol.
- A "string" representing a `spread symbol` (e.g., "AMD/INTC"). Note that "Bar Replay" mode does not work with these symbols.
- The `syminfo.ticker` or `syminfo.tickerid` built-in variables, which return the symbol or the "Exchange:Symbol" pair that the current chart references. We recommend using `syminfo.tickerid` to avoid ambiguity unless the exchange information does not matter in the data request. For more information on `syminfo.*` variables, see [this](#) section of our *Chart information* page.
- A custom ticker identifier created using `ticker.*()` functions. Ticker IDs constructed from these functions may contain additional settings for requesting data using `non-standard chart` calculations, alternative sessions, and other contexts. See the *Custom contexts* section for more information.

The `timeframe` value specifies the timeframe of the requested data. This parameter accepts "string" values in our *timeframe specification* format (e.g., a value of "1D" represents the daily timeframe). To request data from the same timeframe as the chart the script executes on, use the `timeframe.period` variable or an empty string.

The `expression` parameter of the `request.security()` function determines the data it retrieves from the specified context. This versatile parameter accepts "series" values of `int`, `float`, `bool`, `color`, `string`, and `chart.point` types. It can also accept *tuples*, *collections*, *user-defined types*, and the outputs of function and `method` calls. For more details on the data one can retrieve, see the *Requestable data* section below.

---

**Note:** When using the value from an `input.source()` call in the `expression` argument and the input references a series from another indicator, `request.*()` functions calculate that value's results using the **chart's symbol**, regardless of the `symbol` argument supplied, since they cannot evaluate the scopes required by an external series. We therefore do not recommend attempting to request external source input data from other contexts.

---

### Timeframes

The `request.security()` function can request data from any available timeframe, regardless of the chart the script executes on. The timeframe of the data retrieved depends on the `timeframe` argument in the function call, which may represent a higher timeframe (e.g., using “1D” as the `timeframe` value while running the script on an intraday chart) or the chart’s timeframe (i.e., using `timeframe.period` or an empty string as the `timeframe` argument).

Scripts can also request *limited* data from lower timeframes with `request.security()` (e.g., using “1” as the `timeframe` argument while running the script on a 60-minute chart). However, we don’t typically recommend using this function for LTF data requests. The `request.security_lower_tf()` function is more optimal for such cases.

### Higher timeframes

Most use cases of `request.security()` involve requesting data from a timeframe higher than or the same as the chart timeframe. For example, this script retrieves the `hl2` price from a requested `higherTimeframe`. It *plots* the resulting series on the chart alongside the current chart’s `hl2` for comparison:



```

1 //@version=5
2 indicator("Higher timeframe security demo", overlay = true)
3
4 //@variable The higher timeframe to request data from.
5 string higherTimeframe = input.timeframe("240", "Higher timeframe")
6
7 //@variable The `hl2` value from the `higherTimeframe`. Combines lookahead with an
8 //offset to avoid repainting.
9 float htfPrice = request.security(syminfo.tickerid, higherTimeframe, hl2[1],_
10 //lookahead = barmerge.lookahead_on)
11
12 // Plot the `hl2` from the chart timeframe and the `higherTimeframe`.
13 plot(hl2, "Current timeframe HL2", color.teal, 2)
14 plot(htfPrice, "Higher timeframe HL2", color.purple, 3)

```

#### Note that:

- We’ve included an offset to the `expression` argument and used `barmerge.lookahead_on` in `request.security()` to ensure the series returned behaves the same on historical and realtime bars. See the [Avoiding repainting](#) section for more information.

Notice that in the above example, it is possible to select a `higherTimeframe` value that actually represents a *lower timeframe* than the one the chart uses, as the code does not prevent it. When designing a script to work specifically with higher timeframes, we recommend including conditions to prevent it from accessing lower timeframes, especially if you intend to *publish* it.

Below, we've added an `if` structure to our previous example that raises a `runtime error` when the `higherTimeframe` input represents a timeframe smaller than the chart timeframe, effectively preventing the script from requesting LTF data:



```

1 // @version=5
2 indicator("Higher timeframe security demo", overlay = true)
3
4 // @variable The higher timeframe to request data from.
5 string higherTimeframe = input.timeframe("240", "Higher timeframe")
6
7 // Raise a runtime error when the `higherTimeframe` is smaller than the chart's
8 // timeframe.
9 if timeframe.in_seconds() > timeframe.in_seconds(higherTimeframe)
10    runtime.error("The requested timeframe is smaller than the chart's timeframe.
11    Select a higher timeframe.")
12
13 // @variable The `hl2` value from the `higherTimeframe`. Combines lookahead with an
14 // offset to avoid repainting.
15 float htfPrice = request.security(syminfo.tickerid, higherTimeframe, hl2[1],
16    lookahead = barmerge.lookahead_on)
17
18 // Plot the `hl2` from the chart timeframe and the `higherTimeframe`.
19 plot(hl2, "Current timeframe HL2", color.teal, 2)
20 plot(htfPrice, "Higher timeframe HL2", color.purple, 3)

```

## Lower timeframes

Although the `request.security()` function is intended to operate on timeframes greater than or equal to the chart timeframe, it *can* request data from lower timeframes as well, with limitations. When calling this function to access a lower timeframe, it will evaluate the expression from the LTF context. However, it can only return the results from a *single* intrabar (LTF bar) on each chart bar.

The intrabar that the function returns data from on each historical chart bar depends on the `lookahead` value in the function call. When using `barmerge.lookahead_on`, it will return the *first* available intrabar from the chart period. When using `barmerge.lookahead_off`, it will return the *last* intrabar from the chart period. On realtime bars, it returns the last

available value of the `expression` from the timeframe, regardless of the `lookahead` value, as the realtime intrabar information retrieved by the function is not yet sorted.

This script retrieves `close` data from the valid timeframe closest to a fourth of the size of the chart timeframe. It makes two calls to `request.security()` with different `lookahead` values. The first call uses `barmerge.lookahead_on` to access the first intrabar value in each chart bar. The second uses the default `lookahead` value (`barmerge.lookahead_off`), which requests the last intrabar value assigned to each chart bar. The script `plots` the outputs of both calls on the chart to compare the difference:



```

1 //@version=5
2 indicator("Lower timeframe security demo", overlay = true)
3
4 //@variable The valid timeframe closest to 1/4 the size of the chart timeframe.
5 string lowerTimeframe = timeframe.from_seconds(int(timeframe.in_seconds() / 4))
6
7 //@variable The `close` value on the `lowerTimeframe`. Represents the first intrabar_
8 ↴value on each chart bar.
9 float firstLTFClose = request.security(syminfo.tickerid, lowerTimeframe, close,
10 ↴lookahead = barmerge.lookahead_on)
11 //@variable The `close` value on the `lowerTimeframe`. Represents the last intrabar_
12 ↴value on each chart bar.
13 float lastLTFClose = request.security(syminfo.tickerid, lowerTimeframe, close)
14
15 // Plot the values.
16 plot(firstLTFClose, "First intrabar close", color.teal, 3)
17 plot(lastLTFClose, "Last intrabar close", color.purple, 3)
18 // Highlight the background on realtime bars.
19 bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime_
20 ↴background highlight")

```

### Note that:

- The script determines the value of the `lowerTimeframe` by calculating the number of seconds in the chart timeframe with `timeframe.in_seconds()`, then dividing by four and converting the result to a `valid timeframe string` via `timeframe.from_seconds()`.
- The plot of the series without lookahead (purple) aligns with the `close` value on the chart timeframe, as this is the last intrabar value in the chart bar.
- Both `request.security()` calls return the *same* value (the current `close`) on each `realtime` bar, as shown on the

bars with the `orange` background.

- Scripts can retrieve up to 100,000 intrabars from a lower-timeframe context. See [this](#) section of the [Limitations](#) page.

---

**Note:** While scripts can use `request.security()` to retrieve the values from a *single* intrabar on each chart bar, which might provide utility in some unique cases, we recommend using the `request.security_lower_tf()` function for intrabar analysis when possible, as it returns an `array` containing data from *all* available intrabars within a chart bar. See [this section](#) to learn more.

---

## Requestable data

The `request.security()` function is quite versatile, as it can retrieve values of any fundamental type (`int`, `float`, `bool`, `color`, or `string`). It can also request the IDs of data structures and built-in or *user-defined types* that reference fundamental types. The data this function requests depends on its `expression` parameter, which accepts any of the following arguments:

- *Built-in variables and function calls*
- *Variables calculated by the script*
- *Tuples*
- *Calls to user-defined functions*
- *Chart points*
- *Collections*
- *User-defined types*

---

**Note:** The `request.security()` function duplicates the scopes and operations required by the `expression` to calculate its requested values in another context, which elevates runtime memory consumption. Additionally, the extra scopes produced by each call to `request.security()` count toward the script's *compilation limits*. See the [Scope count](#) section of the [Limitations](#) page for more information.

---

## Built-in variables and functions

A frequent use case of `request.security()` is requesting the output of a built-in variable or function/*method* call from another symbol or timeframe.

For example, suppose we want to calculate the 20-bar `SMA` of a symbol's `ohlc4` price from the daily timeframe while on an intraday chart. We can accomplish this with a single line of code:

```
float ma = request.security(syminfo.tickerid, "1D", ta.sma(ohlc4, 20))
```

The above line calculates the value of `ta.sma(ohlc4, 20)` on the current symbol from the daily timeframe.

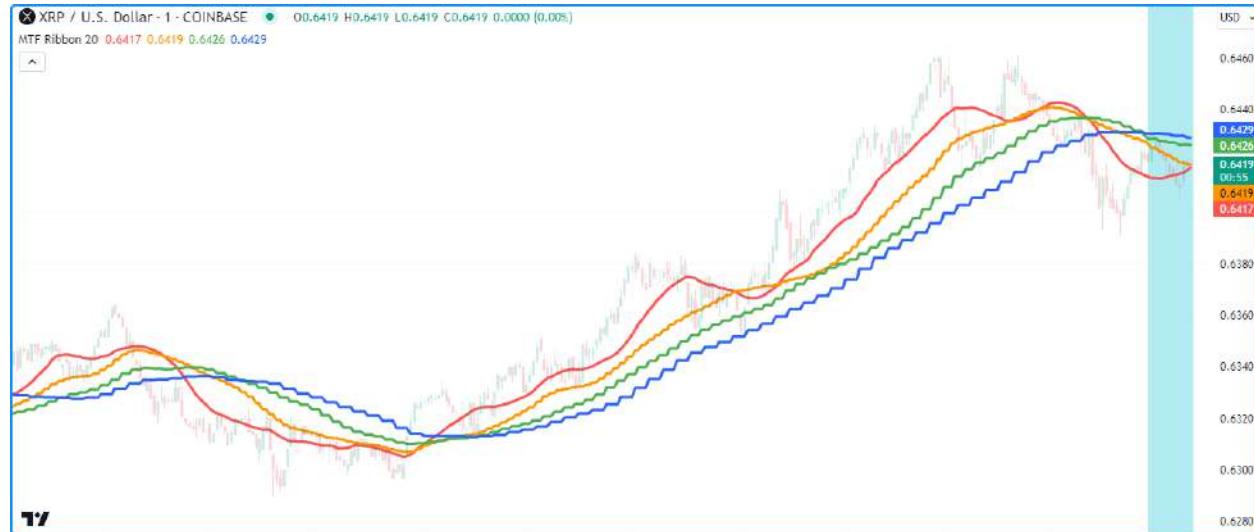
It's important to note that newcomers to Pine may sometimes confuse the above line of code as being equivalent to the following:

```
float ma = ta.sma(request.security(syminfo.tickerid, "1D", ohlc4), 20)
```

However, this line will return an entirely *different* result. Rather than requesting a 20-bar SMA from the daily timeframe, it requests the `ohlc4` price from the daily timeframe and calculates the `ta.sma()` of the results over 20 **chart bars**.

In essence, when the intention is to request the results of an expression from other contexts, pass the expression *directly* to the `expression` parameter in the `request.security()` call, as demonstrated in the initial example.

Let's expand on this concept. The script below calculates a multi-timeframe (MTF) ribbon of moving averages, where each moving average in the ribbon calculates over the same number of bars on its respective timeframe. Each `request.security()` call uses `ta.sma(close, length)` as its `expression` argument to return a `length`-bar SMA from the specified timeframe:



```

1 // @version=5
2 indicator("Requesting built-ins demo", "MTF Ribbon", true)
3
4 // @variable The length of each moving average.
5 int length = input.int(20, "Length", 1)
6
7 // @variable The number of seconds in the chart timeframe.
8 int chartSeconds = timeframe.in_seconds()
9
10 // Calculate the higher timeframes closest to 2, 3, and 4 times the size of the chart
11 // <--timeframe.
12 string htf1 = timeframe.from_seconds(chartSeconds * 2)
13 string htf2 = timeframe.from_seconds(chartSeconds * 3)
14 string htf3 = timeframe.from_seconds(chartSeconds * 4)
15
16 // Calculate the `length`-bar moving averages from each timeframe.
17 float chartAvg = ta.sma(ohlc4, length)
18 float htfAvg1 = request.security(syminfo.tickerid, htf1, ta.sma(ohlc4, length))
19 float htfAvg2 = request.security(syminfo.tickerid, htf2, ta.sma(ohlc4, length))
20 float htfAvg3 = request.security(syminfo.tickerid, htf3, ta.sma(ohlc4, length))
21
22 // Plot the results.
23 plot(chartAvg, "Chart timeframe SMA", color.red, 3)
24 plot(htfAvg1, "Double timeframe SMA", color.orange, 3)
25 plot(htfAvg2, "Triple timeframe SMA", color.green, 3)
26 plot(htfAvg3, "Quadruple timeframe SMA", color.blue, 3)
27
28 // Highlight the background on realtime bars.
29 bgcolor(barstate.isrealtime ? color.new(color.aqua, 70) : na, title = "Realtime
   <--highlight")

```

**Note that:**

- The script calculates the ribbon's higher timeframes by multiplying the chart's `timeframe.in_seconds()` value by 2, 3, and 4, then converting each result into a *valid timeframe string* using `timeframe.from_seconds()`.
- Instead of calling `ta.sma()` within each `request.security()` call, one could use the `chartAvg` variable as the expression in each call to achieve the same result. See the *next section* for more information.
- On realtime bars, this script also tracks *unconfirmed* SMA values from each higher timeframe. See the *Historical and realtime behavior* section to learn more.

## Calculated variables

The `expression` parameter of `request.security()` accepts variables declared in the global scope, allowing scripts to evaluate their variables' calculations from other contexts without redundantly listing the operations in each function call.

For example, one can declare the following variable:

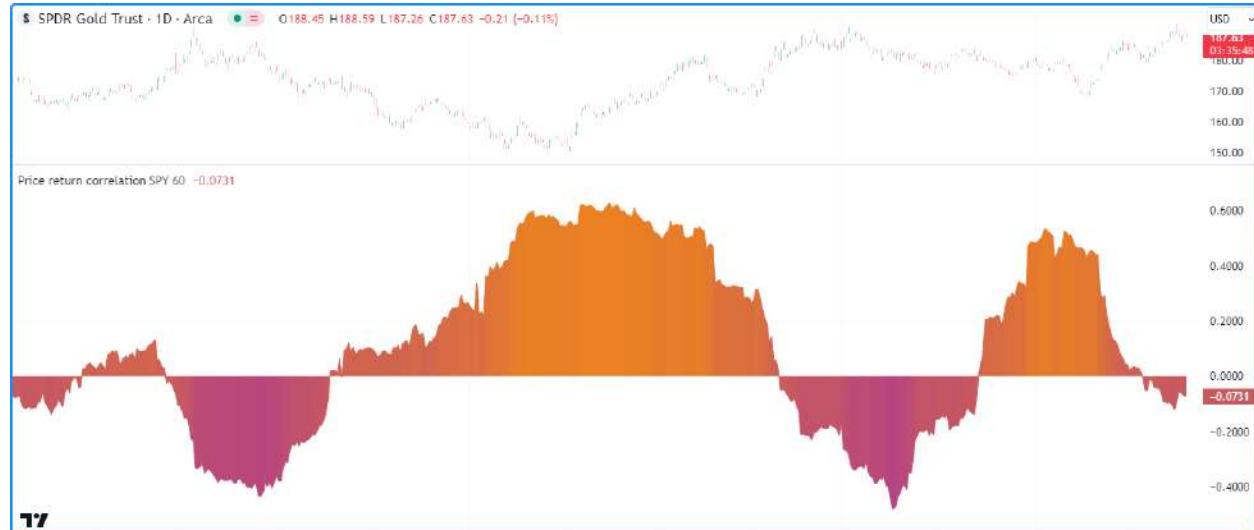
```
priceReturn = (close - close[1]) / close[1]
```

and execute the variable's calculation from another context with `request.security()`:

```
requestedReturn = request.security(symbol, timeframe.period, priceReturn)
```

The function call in the line above will return the result of the `priceReturn` calculation on another `symbol`'s data as a series adapted to the current chart, which the script can display directly on the chart or utilize in additional operations.

The following example compares the price returns of the current chart's symbol and another specified `symbol`. The script declares the `priceReturn` variable from the chart's context, then uses that variable in `request.security()` to evaluate its calculation on another `symbol`. It then calculates the `correlation` between the `priceReturn` and `requestedReturn` and *plots* the result on the chart:



```
1 // @version=5
2 indicator("Requesting calculated variables demo", "Price return correlation")
3
4 // @variable The symbol to compare to the chart symbol.
5 string symbol = input.symbol("SPY", "Symbol to compare")
6 // @variable The number of bars in the calculation window.
7 int length = input.int(60, "Length", 1)
```

(continues on next page)

(continued from previous page)

```

9 // @variable The close-to-close price return.
10 float priceReturn = (close - close[1]) / close[1]
11 // @variable The close-to-close price return calculated on another `symbol`.
12 float requestedReturn = request.security(symbol, timeframe.period, priceReturn)
13
14 // @variable The correlation between the `priceReturn` and `requestedReturn` over
15 // `length` bars.
15 float correlation = ta.correlation(priceReturn, requestedReturn, length)
16 // @variable The color of the correlation plot.
17 color plotColor = color.from_gradient(correlation, -1, 1, color.purple, color.orange)
18
19 // Plot the correlation value.
20 plot(correlation, "Correlation", plotColor, style = plot.style_area)

```

**Note that:**

- The `request.security()` call executes the same calculation used in the `priceReturn` declaration, except it uses the `close` values fetched from the input `symbol`.
- The script colors the plot with a `gradient` based on the `correlation` value. To learn more about color gradients in Pine, see [this](#) section of our User Manual's page on `colors`.

**Tuples**

*Tuples* in Pine Script™ are comma-separated sets of expressions enclosed in brackets that can hold multiple values of any available type. We use tuples when creating functions or other local blocks that return more than one value.

The `request.security()` function can accept a tuple as its `expression` argument, allowing scripts to request multiple series of different types using a single function call. The expressions within requested tuples can be of any type outlined throughout the [Requestable data](#) section of this page, excluding other tuples.

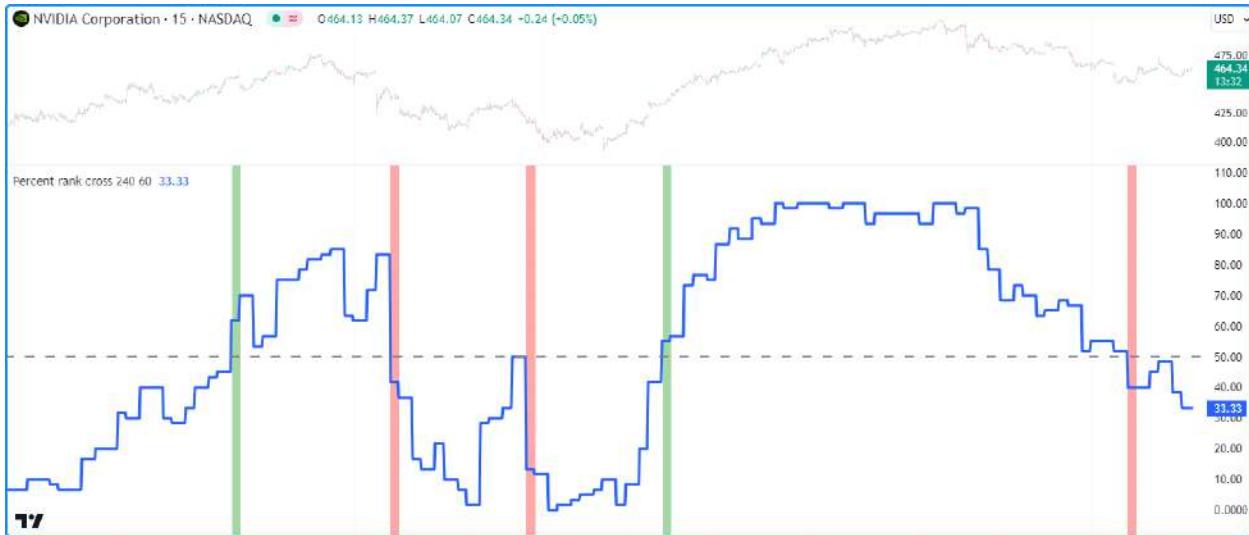
---

**Note:** The combined size of all tuples returned by `request.*()` calls in a script cannot exceed 127 elements. See [this](#) section of the [Limitations](#) page for more information.

---

Tuples are particularly handy when a script needs to retrieve more than one value from a specific context.

For example, this script calculates the `percent rank` of the `close` price over `length` bars and assigns the expression to the `rank` variable. It then calls `request.security()` to request a tuple containing the `rank`, `ta.crossover(rank, 50)`, and `ta.crossunder(rank, 50)` values from the specified `timeframe`. The script `plots` the `requestedRank` and uses the `crossOver` and `crossUnder` “bool” values within `bgcolor()` to conditionally highlight the chart pane’s background:



```

1 // @version=5
2 indicator("Requesting tuples demo", "Percent rank cross")
3
4 // @variable The timeframe of the request.
5 string timeframe = input.timeframe("240", "Timeframe")
6 // @variable The number of bars in the calculation.
7 int length = input.int(20, "Length")
8
9 // @variable The previous bar's percent rank of the `close` price over `length` bars.
10 float rank = ta.percentrank(close, length)[1]
11
12 // Request the `rank` value from another `timeframe`, and two "bool" values
13 // indicating the `rank` from the `timeframe`
14 // crossed over or under 50.
15 [requestedRank, crossOver, crossUnder] = request.security(
16     syminfo.tickerid, timeframe, [rank, ta.crossover(rank, 50), ta.crossunder(rank,
17     -50)],
18     lookahead = barmerge.lookahead_on
19 )
20
21 // Plot the `requestedRank` and create a horizontal line at 50.
22 plot(requestedRank, "Percent Rank", linewidth = 3)
23 hline(50, "Cross line", linewidth = 2)
24 // Highlight the background of all bars where the `timeframe`'s `crossOver` or
25 // `crossUnder` value is `true`.
26 bgcolor(crossOver ? color.new(color.green, 50) : crossUnder ? color.new(color.red,
27     -50) : na)

```

#### Note that:

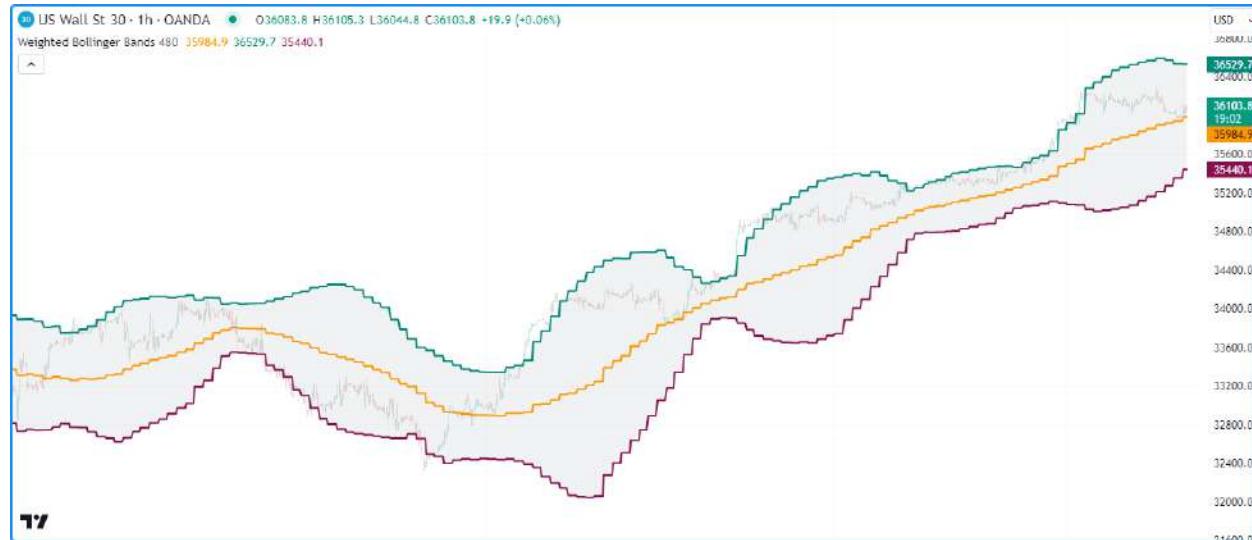
- We've offset the `rank` variable's expression by one bar using the history-referencing operator `[]` and included `barmerge.lookahead_on` in the `request.security()` call to ensure the values on realtime bars do not repaint after becoming historical bars. See the [Avoiding repainting](#) section for more information.
- The `request.security()` call returns a tuple, so we use a *tuple declaration* to declare the `requestedRank`, `crossOver`, and `crossUnder` variables. To learn more about using tuples, see [this section](#) of our User Manual's [Type system](#) page.

## User-defined functions

*User-defined functions* and *methods* are custom functions written by users. They allow users to define sequences of operations associated with an identifier that scripts can conveniently call throughout their execution (e.g., `myUDF()`).

The `request.security()` function can request the results of *user-defined functions* and *methods* whose scopes consist of any types outlined throughout this page's *Requestable data* section.

For example, this script contains a user-defined `weightedBB()` function that calculates Bollinger Bands with the basis average weighted by a specified weight series. The function returns a *tuple* of custom band values. The script calls the `weightedBB()` as the expression argument in `request.security()` to retrieve a *tuple* of band values calculated on the specified timeframe and *plots* the results on the chart:



```

1 // @version=5
2 indicator("Requesting user-defined functions demo", "Weighted Bollinger Bands", true)
3
4 // @variable The timeframe of the request.
5 string timeframe = input.timeframe("480", "Timeframe")
6
7 // @function Calculates Bollinger Bands with a custom weighted basis.
8 // @param source The series of values to process.
9 // @param length The number of bars in the calculation.
10 // @param mult The standard deviation multiplier.
11 // @param weight The series of weights corresponding to each `source` value.
12 // @returns A tuple containing the basis, upper band, and lower band respectively.
13 weightedBB(float source, int length, float mult = 2.0, float weight = 1.0) =>
14     // @variable The basis of the bands.
15     float ma = math.sum(source * weight, length) / math.sum(weight, length)
16     // @variable The standard deviation from the `ma`.
17     float dev = 0.0
18     // Loop to accumulate squared error.
19     for i = 0 to length - 1
20         difference = source[i] - ma
21         dev += difference * difference
22     // Divide `dev` by the `length`, take the square root, and multiply by the `mult`.
23     dev := math.sqrt(dev / length) * mult
24     // Return the bands.
25     [ma, ma + dev, ma - dev]
26

```

(continues on next page)

(continued from previous page)

```

27 // Request weighted bands calculated on the chart symbol's prices over 20 bars from
28 // the
29 // last confirmed bar on the `timeframe`.
30 [basis, highBand, lowBand] = request.security(
31     syminfo.tickerid, timeframe, weightedBB(close[1], 20, 2.0, (high - low)[1]), -
32     lookahead = barmerge.lookahead_on
33 )
34
35 // Plot the values.
36 basisPlot = plot(basis, "Basis", color.orange, 2)
37 upperPlot = plot(highBand, "Upper", color.teal, 2)
38 lowerPlot = plot(lowBand, "Lower", color.maroon, 2)
39 fill(upperPlot, lowerPlot, color.new(color.gray, 90), "Background")

```

**Note that:**

- We offset the source and weight arguments in the `weightedBB()` call used as the expression in `request.security()` and used `barmerge.lookahead_on` to ensure the requested results reflect the last confirmed values from the `timeframe` on realtime bars. See [this section](#) to learn more.

**Chart points**

*Chart points* are reference types that represent coordinates on the chart. *Lines*, *boxes*, *polylines*, and *labels* use `chart.point` objects to set their display locations.

The `request.security()` function can use the ID of a `chart.point` instance in its `expression` argument, allowing scripts to retrieve chart coordinates from other contexts.

The example below requests a tuple of historical *chart points* from a higher timeframe and uses them to draw *boxes* on the chart. The script declares the `topLeft` and `bottomRight` variables that reference `chart.point` IDs from the last confirmed bar. It then uses `request.security()` to request a *tuple* containing the IDs of *chart points* representing the `topLeft` and `bottomRight` from a `higherTimeframe`.

When a new bar starts on the `higherTimeframe`, the script draws a new box using the `time` and `price` coordinates from the `requestedTopLeft` and `requestedBottomRight` chart points:



```

1 // @version=5
2 indicator("Requesting chart points demo", "HTF Boxes", true, max_boxes_count = 500)
3
4 // @variable The timeframe to request data from.
5 string higherTimeframe = input.timeframe("1D", "Timeframe")
6
7 // Raise a runtime error if the `higherTimeframe` is smaller than the chart's
8 // timeframe.
9 if timeframe.in_seconds(higherTimeframe) < timeframe.in_seconds(timeframe.period)
10    runtime.error("The selected timeframe is too small. Choose a higher timeframe.")
11
12 // @variable A `chart.point` containing top-left coordinates from the last confirmed
13 // bar.
14 topLeft = chart.point.now(high) [1]
15 // @variable A `chart.point` containing bottom-right coordinates from the last
16 // confirmed bar.
17 bottomRight = chart.point.from_time(time_close, low) [1]
18
19 // Request the last confirmed `topLeft` and `bottomRight` chart points from the
20 // `higherTimeframe`.
21 [requestedTopLeft, requestedBottomRight] = request.security(
22     syminfo.tickerid, higherTimeframe, [topLeft, bottomRight], lookahead = barmerge.
23     lookahead_on
24 )
25
26 // Draw a new box when a new `higherTimeframe` bar starts.
27 // The box uses the `time` fields from the `requestedTopLeft` and
28 // `requestedBottomRight` as x-coordinates.
29 if timeframe.change(higherTimeframe)
30     box.new(
31         requestedTopLeft, requestedBottomRight, color.purple, 3,
32         xloc = xloc.bar_time, bgcolor = color.new(color.purple, 90)
33     )

```

### Note that:

- Since this example is designed specifically for higher timeframes, we've included a custom `runtime error` that the script raises when the `timeframe.in_seconds()` of the `higherTimeframe` is smaller than that of the chart's `timeframe`.

### Collections

Pine Script™ *collections* (*arrays*, *matrices*, and *maps*) are data structures that contain an arbitrary number of elements with specified types. The `request.security()` function can retrieve the IDs of *collections* whose elements consist of:

- Fundamental types
- *Chart points*
- *User-defined types* that satisfy the criteria listed in the *section below*

This example calculates the ratio of a confirmed bar's high-low range to the range between the `highest` and `lowest` values over 10 bars from a specified symbol and timeframe. It uses `maps` to hold the values used in the calculations.

The script creates a `data map` with “string” keys and “float” values to hold `high`, `low`, `highest`, and `lowest` price values on each bar, which it uses as the expression in `request.security()` to calculate an `otherData` map representing the `data` from the specified context. It uses the values associated with the “High”, “Low”, “Highest”, and “Lowest” keys of the `otherData` map to calculate the `ratio` that it `plots` in the chart pane:



```

1 // @version=5
2 indicator("Requesting collections demo", "Bar range ratio")
3
4 // @variable The ticker ID to request data from.
5 string symbol = input.symbol("", "Symbol")
6 // @variable The timeframe of the request.
7 string timeframe = input.timeframe("30", "Timeframe")
8
9 // @variable A map with "string" keys and "float" values.
10 var map<string, float> data = map.new<string, float>()
11
12 // Put key-value pairs into the `data` map.
13 map.put(data, "High", high)
14 map.put(data, "Low", low)
15 map.put(data, "Highest", ta.highest(10))
16 map.put(data, "Lowest", ta.lowest(10))
17
18 // @variable A new `map` whose data is calculated from the last confirmed bar of the
19 // requested context.
20 map<string, float> otherData = request.security(symbol, timeframe, data[1], lookahead_
21 // = barmerge.lookahead_on)
22
23 // @variable The ratio of the context's bar range to the max range over 10 bars.
24 // Returns `na` if no data is available.
25 float ratio = na
26 if not na(otherData)
27     ratio := (otherData.get("High") - otherData.get("Low")) / (otherData.get("Highest"
28 // ) - otherData.get("Lowest"))
29
30 // @variable A gradient color for the plot of the `ratio`.
31 color ratioColor = color.from_gradient(ratio, 0, 1, color.purple, color.orange)
32
33 // Plot the `ratio`.
34 plot(ratio, "Range Ratio", ratioColor, 3, plot.style_area)

```

#### Note that:

- The `request.security()` call in this script can return `na` if no data is available from the specified context. Since one cannot call `methods` on a `map` variable when its value is `na`, we've added an `if` structure to only calculate a new `ratio` value when `otherData` references a valid `map` ID.

### User-defined types

*User-defined types (UDTs)* are *composite types* containing an arbitrary number of *fields*, which can be of any available type, including other *user-defined types*.

The `request.security()` function can retrieve the IDs of *objects* produced by *UDTs* from other contexts if their fields consist of:

- Fundamental types
- *Chart points*
- *Collections* that satisfy the criteria listed in the *section above*
- Other *UDTs* whose fields consist of any of these types

The following example requests an *object* ID using a specified *symbol* and displays its field values on a chart pane.

The script contains a `TickerInfo` UDT with “string” fields for `syminfo.*` values, an `array` field to store recent “float” price data, and an “int” field to hold the requested ticker’s `bar_index` value. It assigns a new `TickerInfo` ID to an `info` variable on every bar and uses the variable as the expression in `request.security()` to retrieve the ID of an *object* representing the calculated `info` from the specified *symbol*.

The script displays the `requestedInfo` object’s `description`, `tickerType`, `currency`, and `barIndex` values in a `label` and uses `plotcandle()` to display the values from its `prices` array:



```

1 // @version=5
2 indicator("Requesting user-defined types demo", "Ticker info")
3
4 // @variable The symbol to request information from.
5 string symbol = input.symbol("NASDAQ:AAPL", "Symbol")
6
7 // @type          A custom type containing information about a ticker.
8 // @field description  The symbol's description.
9 // @field tickerType   The type of ticker.
10 // @field currency     The symbol's currency.
11 // @field prices       An array of the symbol's current prices.
12 // @field barIndex      The ticker's `bar_index`.
13 type TickerInfo
14     string      description
15     string      tickerType

```

(continues on next page)

(continued from previous page)

```

16     string      currency
17     array<float> prices
18     int         barIndex
19
20 // @variable A `TickerInfo` object containing current data.
21 info = TickerInfo.new(
22     syminfo.description, syminfo.type, syminfo.currency, array.from(open, high, low, ←
23     close), bar_index
24 )
25 // @variable The `info` requested from the specified `symbol`.
26 TickerInfo requestedInfo = request.security(symbol, timeframe.period, info)
27 // Assign a new `TickerInfo` instance to `requestedInfo` if one wasn't retrieved.
28 if na(requestedInfo)
29     requestedInfo := TickerInfo.new(prices = array.new<float>(4))
30
31 // @variable A label displaying information from the `requestedInfo` object.
32 var infoLabel = label.new(
33     na, na, "", color = color.purple, style = label.style_label_left, textcolor = ←
34     color.white, size = size.large
35 )
36 // @variable The text to display inside the `infoLabel`.
37 string infoText = na(requestedInfo) ? "" : str.format(
38     "{0}\nType: {1}\nCurrency: {2}\nBar Index: {3}",
39     requestedInfo.description, requestedInfo.tickerType, requestedInfo.currency, ←
40     requestedInfo.barIndex
41 )
42
43 // Set the `point` and `text` of the `infoLabel`.
44 label.set_point(infoLabel, chart.point.now(array.last(requestedInfo.prices)))
45 label.set_text(infoLabel, infoText)
46 // Plot candles using the values from the `prices` array of the `requestedInfo`.
47 plotcandle(
48     requestedInfo.prices.get(0), requestedInfo.prices.get(1), requestedInfo.prices.←
49     get(2), requestedInfo.prices.get(3),
50     "Requested Prices"
51 )

```

**Note that:**

- The `syminfo.*` variables used in this script all return “simple string” qualified types. However, `objects` in Pine are *always* qualified as “series”. Consequently, all values assigned to the `info` object’s fields automatically adopt the “series” *qualifier*.
- It is possible for the `request.security()` call to return `na` due to differences between the data requested from the `symbol` and the main chart. This script assigns a new `TickerInfo` object to the `requestedInfo` in that case to prevent runtime errors.

#### 4.14.5 `request.security\_lower\_tf()`

The `request.security_lower_tf()` function is an alternative to `request.security()` designed for reliably requesting information from lower-timeframe (LTF) contexts.

While `request.security()` can retrieve data from a *single* intrabar (LTF bar) in each chart bar, `request.security_lower_tf()` retrieves data from *all* available intrabars in each chart bar, which the script can access and use in additional calculations. Each `request.security_lower_tf()` call can retrieve up to 100,000 intrabars from a lower timeframe. See [this](#) section of our [Limitations](#) page for more information.

---

**Note:** Working with `request.security_lower_tf()` involves frequent usage of `arrays` since it always returns `array` results. We therefore recommend you familiarize yourself with `arrays` to make the most of this function in your scripts.

---

Below is the function's signature, which is similar to `request.security()`:

```
request.security_lower_tf(symbol, timeframe, expression, ignore_invalid_symbol, -  
→currency, ignore_invalid_timeframe) → array<type>
```

This function **only** requests data from timeframes less than or equal to the chart's timeframe. If the `timeframe` of the request represents a higher timeframe than the chart's `timeframe`, the function will either raise a runtime error or return `na` values depending on the `ignore_invalid_timeframe` argument in the call. The default value for this parameter is `false`, meaning it will raise an error and halt the script's execution when attempting to request HTF data.

#### Requesting intrabar data

Intrabar data can provide a script with additional information that may not be obvious or accessible from solely analyzing data sampled on the chart's timerframe. The `request.security_lower_tf()` function can retrieve many data types from an intrabar context.

Before you venture further in this section, we recommend exploring the [Requestable data](#) portion of the `request.security()` section above, which provides foundational information about the types of data one can request. The `expression` parameter in `request.security_lower_tf()` accepts most of the same arguments discussed in that section, excluding direct references to `collections` and mutable variables declared in the script's main scope. Although it accepts many of the same types of arguments, this function returns `array` results, which comes with some differences in interpretation and handling, as explained below.

---

**Note:** As with `request.security()`, `request.security_lower_tf()` duplicates the scopes and operations required to calculate the `expression` from another context. The scopes from `request.security_lower_tf()` increase runtime memory consumption and count toward the script's compilation limits. See the [Scope count](#) section of the [Limitations](#) page to learn more.

---

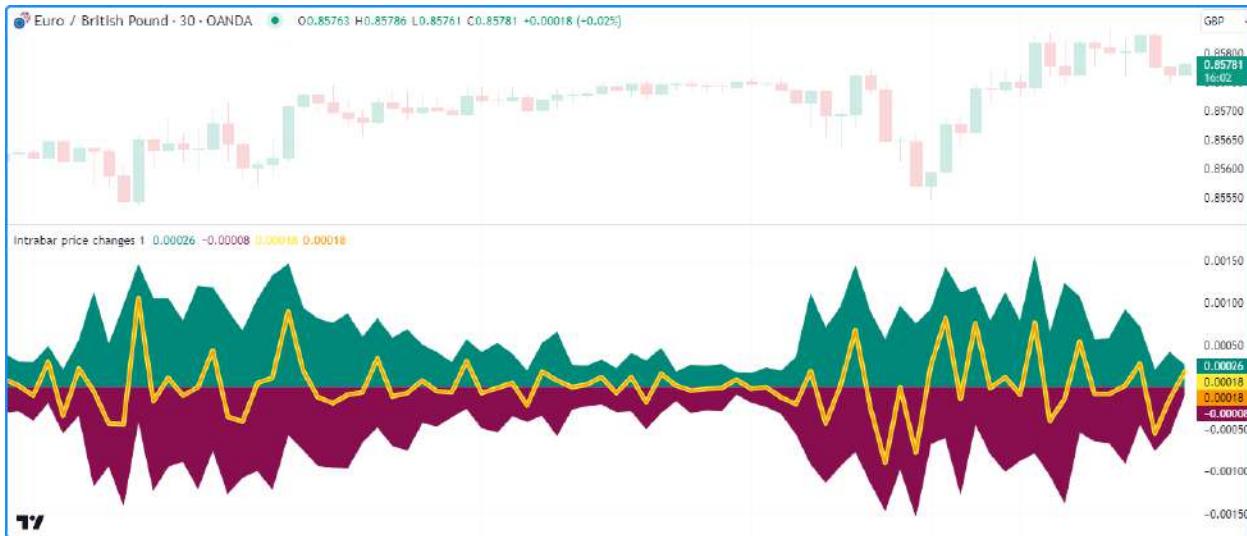
#### Intrabar data arrays

Lower timeframes contain more data points than higher timeframes, as new values come in at a *higher frequency*. For example, when comparing a 1-minute chart to an hourly chart, the 1-minute chart will have up to 60 times the number of bars per hour, depending on the available data.

To address the fact that multiple intrabars exist within a chart bar, `request.security_lower_tf()` always returns its results as `arrays`. The elements in the returned `arrays` represent the `expression` values retrieved from the lower timeframe sorted in ascending order based on each intrabar's timestamp.

The `type template` assigned to the returned `arrays` corresponds to the value types passed in the `request.security_lower_tf()` call. For example, using an “int” as the expression will produce an `array<int>` instance, a “bool” as the expression will produce an `array<bool>` instance, etc.

The following script uses intrabar information to decompose the chart’s close-to-close price changes into positive and negative parts. It calls `request.security_lower_tf()` to fetch a “float” array of `ta.change(close)` values from the lowerTimeframe on each chart bar, then accesses all the array’s elements using a `for...in` loop to accumulate `positiveChange` and `negativeChange` sums. The script adds the accumulated values to calculate the `netChange`, then `plots` the results on the chart alongside the `priceChange` for comparison:



```

1 // @version=5
2 indicator("Intrabar arrays demo", "Intrabar price changes")
3
4 // @variable The lower timeframe of the requested data.
5 string lowerTimeframe = input.timeframe("1", "Timeframe")
6
7 // @variable The close-to-close price change.
8 float priceChange = ta.change(close)
9
10 // @variable An array of `close` values from available intrabars on the
11 // `lowerTimeframe`.
12 array<float> intrabarChanges = request.security_lower_tf(syminfo.tickerid,
13 // lowerTimeframe, priceChange)
14
15 // @variable The total positive intrabar `close` movement on the chart bar.
16 float positiveChange = 0.0
17 // @variable The total negative intrabar `close` movement on the chart bar.
18 float negativeChange = 0.0
19
20 // Loop to calculate totals, starting from the chart bar's first available intrabar.
21 for change in intrabarChanges
22     // Add the `change` to `positiveChange` if its sign is 1, and add to
23     // `negativeChange` if its sign is -1.
24     switch math.sign(change)
25         1 => positiveChange += change
26         -1 => negativeChange += change
27
28 // @variable The sum of `positiveChange` and `negativeChange`. Equals the
29 // `priceChange` on bars with available intrabars.

```

(continues on next page)

(continued from previous page)

```

26 float netChange = positiveChange + negativeChange
27
28 // Plot the `positiveChange`, `negativeChange`, and `netChange`.
29 plot(positiveChange, "Positive intrabar change", color.teal, style = plot.style_area)
30 plot(negativeChange, "Negative intrabar change", color.maroon, style = plot.style_
31 ↪area)
31 plot(netChange, "Net intrabar change", color.yellow, 5)
32 // Plot the `priceChange` to compare.
33 plot(priceChange, "Chart price change", color.orange, 2)

```

**Note that:**

- The *plots* based on intrabar data may not appear on all available chart bars, as `request.security_lower_tf()` can only access up to the most recent 100,000 intrabars available from the requested context. When executing this function on a chart bar that doesn't have accessible intrabar data, it will return an *empty array*.
- The number of intrabars per chart bar may vary depending on the data available from the context and the chart the script executes on. For example, a provider's 1-minute data feed may not include data for every minute within the 60-minute timeframe due to a lack of trading activity over some 1-minute intervals. To check the number of intrabars retrieved for a chart bar, one can use `array.size()` on the resulting *array*.
- If the `lowerTimeframe` value is greater than the chart's timeframe, the script will raise a *runtime error*, as we have not supplied an `ignore_invalid_timeframe` argument in the `request.security_lower_tf()` call.

**Tuples of intrabar data**

When passing a tuple or a function call that returns a tuple as the `expression` argument in `request.security_lower_tf()`, the result is a tuple of *arrays* with *type templates* corresponding to the types within the argument. For example, using a `[float, string, color]` tuple as the `expression` will result in `[array<float>, array<string>, array<color>]` data returned by the function. Using a tuple `expression` allows a script to fetch several *arrays* of intrabar data with a single `request.security_lower_tf()` function call.

---

**Note:** The combined size of all tuples returned by `request.*()` calls in a script is limited to 127 elements. See [this](#) section of the [Limitations](#) page for more information.

---

The following example requests OHLC data from a lower timeframe and visualizes the current bar's intrabars on the chart using *lines and boxes*. The script calls `request.security_lower_tf()` with the `[open, high, low, close]` tuple as its `expression` to retrieve a tuple of *arrays* representing OHLC information from a calculated `lowerTimeframe`. It then uses a `for` loop to set line coordinates with the retrieved data and current bar indices to display the results next to the current chart bar, providing a “magnified view” of the price movement within the latest candle. It also draws a `box` around the *lines* to indicate the chart region occupied by intrabar drawings:



```

1 // @version=5
2 indicator("Tuples of intrabar data demo", "Candle magnifier", max_lines_count = 500)
3
4 // @variable The maximum number of intrabars to display.
5 int maxIntrabars = input.int(20, "Max intrabars", 1, 250)
6 // @variable The width of the drawn candle bodies.
7 int candleWidth = input.int(20, "Candle width", 2)
8
9 // @variable The largest valid timeframe closest to `maxIntrabars` times smaller than
10 // the chart timeframe.
11 string lowerTimeframe = timeframe.from_seconds(math.ceil(timeframe.in_seconds() /_
12 //maxIntrabars))
13
14 // @variable An array of lines to represent intrabar wicks.
15 var array<line> wicks = array.new<line>()
16 // @variable An array of lines to represent intrabar bodies.
17 var array<line> bodies = array.new<line>()
18 // @variable A box that surrounds the displayed intrabars.
19 var box magnifierBox = box.new(na, na, na, na, bgcolor = na)
20
21 // Fill the `wicks` and `bodies` arrays with blank lines on the first bar.
22 if barstate.isfirst
23     for i = 1 to maxIntrabars
24         array.push(wicks, line.new(na, na, na, na, color = color.gray))
25         array.push(bodies, line.new(na, na, na, na, width = candleWidth))
26
27 // @variable A tuple of "float" arrays containing `open`, `high`, `low`, and `close`_
28 // prices from the `lowerTimeframe`.
29 [oData, hData, lData, cData] = request.security_lower_tf(syminfo.tickerid,_
30 lowerTimeframe, [open, high, low, close])
31 // @variable The number of intrabars retrieved from the `lowerTimeframe` on the chart_
32 // bar.
33 int numIntrabars = array.size(oData)
34
35 if numIntrabars > 0
36     // Define the start and end bar index values for intrabar display.
37     int startIndex = bar_index + 2
38     int endIndex = startIndex + numIntrabars
39     // Loop to update lines.

```

(continues on next page)

(continued from previous page)

```

35  for i = 0 to maxIntrabars - 1
36      line wickLine = array.get(wicks, i)
37      line bodyLine = array.get(bodies, i)
38      if i < numIntrabars
39          // @variable The `bar_index` of the drawing.
40          int candleIndex = startIndex + i
41          // Update the properties of the `wickLine` and `bodyLine`.
42          line.set_xy1(wickLine, startIndex + i, array.get(hData, i))
43          line.set_xy2(wickLine, startIndex + i, array.get(lData, i))
44          line.set_xy1(bodyLine, startIndex + i, array.get(oData, i))
45          line.set_xy2(bodyLine, startIndex + i, array.get(cData, i))
46          line.set_color(bodyLine, bodyLine.get_y2() > bodyLine.get_y1() ? color.
47          ← teal : color.maroon)
48          continue
49          // Set the coordinates of the `wickLine` and `bodyLine` to `na` if no_
50          ← intrabar data is available at the index.
51          line.set_xy1(wickLine, na, na)
52          line.set_xy2(wickLine, na, na)
53          line.set_xy1(bodyLine, na, na)
54          line.set_xy2(bodyLine, na, na)
55          // Set the coordinates of the `magnifierBox`.
56          box.set_lefttop(magnifierBox, startIndex - 1, array.max(hData) )
57          box.set_rightbottom(magnifierBox, endIndex, array.min(lData) )

```

**Note that:**

- The script draws each candle using two *lines*: one to represent wicks and the other to represent the body. Since the script can display up to 500 lines on the chart, we've limited the `maxIntrabars` input to 250.
- The `lowerTimeframe` value is the result of calculating the `math.ceil()` of the `timeframe.in_seconds()` divided by the `maxIntrabars` and converting to a *valid timeframe string* with `timeframe.from_seconds()`.
- The script sets the top of the box drawing using the `array.max()` of the requested `hData` array, and it sets the box's bottom using the `array.min()` of the requested `lData` array. As we see on the chart, these values correspond to the `high` and `low` of the chart bar.

**Requesting collections**

In some cases, a script may need to request the IDs of *collections* from an intrabar context. However, unlike `request.security()`, one cannot pass *collections* or calls to functions that return them as the expression argument in a `request.security_lower_tf()` call, as *arrays* cannot directly reference other *collections*.

Despite these limitations, it is possible to request *collections* from lower timeframes, if needed, with the help of *wrapper* types.

---

**Note:** The use case described below is **advanced** and **not** recommended for beginners. Before exploring this approach, we recommend understanding how *user-defined types* and *collections* work in Pine Script™. When possible, we recommend using *simpler* methods to manage LTF requests, and only using this approach when *nothing else* will suffice.

To make *collections* requestable with `request.security_lower_tf()`, we must create a *UDT* with a field to reference a collection ID. This step is necessary since *arrays* cannot reference other *collections* directly but *can* reference UDTs with collection fields:

```
//@type A "wrapper" type to hold an `array<float>` instance.
type Wrapper
    array<float> collection
```

With our `Wrapper` UDT defined, we can now pass the IDs of *objects* of the UDT to the `expression` parameter in `request.security_lower_tf()`.

A straightforward approach is to call the built-in `*.new()` function as the `expression`. For example, this line of code calls `Wrapper.new()` with `array.from(close)` as its `collection` within `request.security_lower_tf()`:

```
//@variable An array of `Wrapper` IDs requested from the 1-minute timeframe.
array<Wrapper> wrappers = request.security_lower_tf(syminfo.tickerid, "1", Wrapper,
    ↪new(array.from(close)))
```

Alternatively, we can create a *user-defined function* or *method* that returns an *object* of the *UDT* and call that function within `request.security_lower_tf()`. For instance, this code calls a custom `newWrapper()` function that returns a `Wrapper` ID as the `expression` argument:

```
//@function Creates a new `Wrapper` instance to wrap the specified `collection`.
newWrapper(array<float> collection) =>
    Wrapper.new(collection)

//@variable An array of `Wrapper` IDs requested from the 1-minute timeframe.
array<Wrapper> wrappers = request.security_lower_tf(syminfo.tickerid, "1",
    ↪newWrapper(array.from(close)))
```

The result with either of the above is an `array` containing `Wrapper` IDs from all available intrabars in the chart bar, which the script can use to reference `Wrapper` instances from specific intrabars and use their `collection` fields in additional operations.

The script below utilizes this approach to collect *arrays* of intrabar data from a `lowerTimeframe` and uses them to display data from a specific intrabar. Its custom `Prices` type contains a single `data` field to reference `array<float>` instances that hold price data, and the user-defined `newPrices()` function returns the ID of a `Prices` object.

The script calls `request.security_lower_tf()` with a `newPrices()` call as its `expression` argument to retrieve an `array` of `Prices` IDs from each intrabar in the chart bar, then uses `array.get()` to get the ID from a specified available intrabar, if it exists. Lastly, it uses `array.get()` on the `data` array assigned to that instance and calls `plotcandle()` to display its values on the chart:



```

1 // @version=5
2 indicator("Requesting LTF collections demo", "Intrabar viewer", true)
3
4 // @variable The timeframe of the LTF data request.
5 string lowerTimeframe = input.timeframe("1", "Timeframe")
6 // @variable The index of the intrabar to show on each chart bar. 0 is the first
7 // available intrabar.
8 int intrabarIndex = input.int(0, "Intrabar to show", 0)
9
10 // @variable A custom type to hold an array of price `data`.
11 type Prices
12     array<float> data
13
14 // @function Returns a new `Prices` instance containing current `open`, `high`, `low`, and `close` prices.
15 newPrices() =>
16     Prices.new(array.from(open, high, low, close))
17
18 // @variable An array of `Prices` requested from the `lowerTimeframe`.
19 array<Prices> requestedPrices = request.security_lower_tf(syminfo.tickerid,
20     lowerTimeframe, newPrices())
21
22 // @variable The `Prices` ID from the `requestedPrices` array at the `intrabarIndex`, or `na` if not available.
23 Prices intrabarPrices = array.size(requestedPrices) > intrabarIndex ? array.
24     get(requestedPrices, intrabarIndex) : na
25 // @variable The `data` array from the `intrabarPrices`, or an array of `na` values if `intrabarPrices` is `na`.
26 array<float> intrabarData = na(intrabarPrices) ? array.new<float>(4, na) :
27     intrabarPrices.data
28
29 // Plot the `intrabarData` values as candles.
30 plotcandle(intrabarData.get(0), intrabarData.get(1), intrabarData.get(2),
31             intrabarData.get(3))

```

**Note that:**

- The `intrabarPrices` variable only references a `Prices` ID if the `size` of the `requestedPrices` array is greater than the `intrabarIndex`, as attempting to use `array.get()` to get an element that doesn't exist will result in an *out of bounds error*.
- The `intrabarData` variable only references the `data` field from `intrabarPrices` if a valid `Prices` ID exists since a script cannot reference fields of an `na` value.
- The process used in this example is *not* necessary to achieve the intended result. We could instead avoid using *UDTs* and pass an `[open, high, low, close]` tuple to the `expression` parameter to retrieve a tuple of *arrays* for further operations, as explained in the *previous section*.

#### 4.14.6 Custom contexts

Pine Script™ includes multiple `ticker.*()` functions that allow scripts to construct *custom* ticker IDs that specify additional settings for data requests when used as a `symbol` argument in `request.security()` and `request.security_lower_tf()`:

- `ticker.new()` constructs a custom ticker ID from a specified `prefix` and `ticker` with additional `session` and `adjustment` settings.
- `ticker.modify()` constructs a modified form of a specified `tickerid` with additional `session` and `adjustment` settings.
- `ticker.heikinashi()`, `ticker.renko()`, `ticker.pointfigure()`, `ticker.kagi()`, and `ticker.linebreak()` construct a modified form a `symbol` with *non-standard chart* settings.
- `ticker.inherit()` constructs a new ticker ID for a `symbol` with additional parameters inherited from the `from_tickerid` specified in the function call, allowing scripts to request the `symbol` data with the same modifiers as the `from_tickerid`, including session, dividend adjustment, currency conversion, non-standard chart type, back-adjustment, settlement-as-close, etc.
- `ticker.standard()` constructs a standard ticker ID representing the `symbol` *without* additional modifiers.

Let's explore some practical examples of applying `ticker.*()` functions to request data from custom contexts.

Suppose we want to include dividend adjustment in a stock symbol's prices without enabling the "Adjust data for dividends" option in the "Symbol" section of the chart's settings. We can achieve this in a script by constructing a custom ticker ID for the instrument using `ticker.new()` or `ticker.modify()` with an `adjustment` value of `adjustment.dividends`.

This script creates an `adjustedTickerID` using `ticker.modify()`, uses that ticker ID as the `symbol` in `request.security()` to retrieve a *tuple* of adjusted price values, then plots the result as `candles` on the chart. It also highlights the background when the requested prices differ from the prices without dividend adjustment.

As we see on the "NYSE:XOM" chart below, enabling dividend adjustment results in different historical values before the date of the latest dividend:



```

1 // @version=5
2 indicator("Custom contexts demo 1", "Adjusted prices", true)
3
4 // @variable A custom ticker ID representing the chart's symbol with the dividend_
5 // adjustment modifier.
6 string adjustedTickerID = ticker.modify(syminfo.tickerid, adjustment = adjustment.
7 //dividends)
```

(continues on next page)

(continued from previous page)

```

6 // Request the adjusted prices for the chart's symbol.
7 [o, h, l, c] = request.security(adjustedTickerID, timeframe.period, [open, high, low,_
8   →close])
9
10 // @variable The color of the candles on the chart.
11 color candleColor = c > o ? color.teal : color.maroon
12
13 // Plot the adjusted prices.
14 plotcandle(o, h, l, c, "Adjusted Prices", candleColor)
15 // Highlight the background when `c` is different from `close`.
16 bgcolor(c != close ? color.new(color.orange, 80) : na)

```

### Note that:

- If a modifier included in a constructed ticker ID does not apply to the symbol, the script will *ignore* that modifier when requesting data. For instance, this script will display the same values as the main chart on forex symbols such as “EURUSD”.

While the example above demonstrates a simple way to modify the chart’s symbol, a more frequent use case for `ticker.*()` functions is applying custom modifiers to another symbol while requesting data. If a ticker ID referenced in a script already has the modifiers one would like to apply (e.g., adjustment settings, session type, etc.), they can use `ticker.inherit()` to quickly and efficiently add those modifiers to another symbol.

In the example below, we’ve edited the previous script to request data for a `symbolInput` using modifiers inherited from the `adjustedTickerID`. This script calls `ticker.inherit()` to construct an `inheritedTickerID` and uses that ticker ID in a `request.security()` call. It also requests data for the `symbolInput` without additional modifiers and plots `candles` for both ticker IDs in a separate chart pane to compare the difference.

As shown on the chart, the data requested using the `inheritedTickerID` includes dividend adjustment, whereas the data requested using the `symbolInput` directly does not:



```

1 // @version=5
2 indicator("Custom contexts demo 2", "Inherited adjustment")
3
4 // @variable The symbol to request data from.
5 string symbolInput = input.symbol("NYSE:PFE", "Symbol")
6

```

(continues on next page)

(continued from previous page)

```

7 // @variable A custom ticker ID representing the chart's symbol with the dividend_
8 // adjustment modifier.
9 string adjustedTickerID = ticker.modify(syminfo.tickerid, adjustment = adjustment.
10 // dividends)
11 // @variable A custom ticker ID representing the `symbolInput` with modifiers_
12 // inherited from the `adjustedTickerID`.
13 string inheritedTickerID = ticker.inherit(adjustedTickerID, symbolInput)
14
15 // Request prices using the `symbolInput`.
16 [o1, h1, l1, c1] = request.security(symbolInput, timeframe.period, [open, high, low,
17 // close])
18 // Request prices using the `inheritedTickerID`.
19 [o2, h2, l2, c2] = request.security(inheritedTickerID, timeframe.period, [open, high,
20 // low, close])
21
22 // @variable The color of the candles that use the `inheritedTickerID` prices.
23 color candleColor = c2 > o2 ? color.teal : color.maroon
24
25 // Plot the `symbol` prices.
26 plotcandle(o1, h1, l1, c1, "Symbol", color.gray, color.gray, bordercolor = color.gray)
27 // Plot the `inheritedTickerID` prices.
28 plotcandle(o2, h2, l2, c2, "Symbol With Modifiers", candleColor)
29 // Highlight the background when `c1` is different from `c2`.
30 bgcolor(c1 != c2 ? color.new(color.orange, 80) : na)

```

**Note that:**

- Since the `adjustedTickerID` represents a modified form of the `syminfo.tickerid`, if we modify the chart's context in other ways, such as changing the chart type or enabling extended trading hours in the chart's settings, those modifiers will also apply to the `adjustedTickerID` and `inheritedTickerID`. However, they will *not* apply to the `symbolInput` since it represents a *standard* ticker ID.

Another frequent use case for requesting custom contexts is retrieving data that uses *non-standard chart* calculations. For example, suppose we want to use `Renko` price values to calculate trade signals in a `strategy()` script. If we simply change the chart type to “Renko” to get the prices, the `strategy` will also simulate its trades based on those synthetic prices, producing *misleading* results:



```

1 //@version=5
2 strategy(
3     "Custom contexts demo 3", "Renko strategy", true, default_qty_type = strategy.
4     ↵percent_of_equity,
5     default_qty_value = 2, initial_capital = 50000, slippage = 2,
6     commission_type = strategy.commission.cash_per_contract, commission_value = 1, ↵
7     ↵margin_long = 100,
8     margin_short = 100
9 )
10
11 //@variable When `true`, the strategy places a long market order.
12 bool longEntry = ta.crossover(close, open)
13 //@variable When `true`, the strategy places a short market order.
14 bool shortEntry = ta.crossunder(close, open)
15
16 if longEntry
17     strategy.entry("Long Entry", strategy.long)
18 if shortEntry
19     strategy.entry("Short Entry", strategy.short)

```

To ensure our strategy shows results based on *actual* prices, we can create a Renko ticker ID using `ticker.renko()` while keeping the chart on a *standard type*, allowing the script to request and use **Renko** prices to calculate its signals without calculating the strategy results on them:



```

1 //@version=5
2 strategy(
3     "Custom contexts demo 3", "Renko strategy", true, default_qty_type = strategy.
4     ↵percent_of_equity,
5     default_qty_value = 2, initial_capital = 50000, slippage = 1,
6     commission_type = strategy.commission.cash_per_contract, commission_value = 1, ↵
7     ↵margin_long = 100,
8     margin_short = 100
9 )
10
11 //@variable A Renko ticker ID.
12 string renkoTickerID = ticker.renko(syminfo.tickerid, "ATR", 14)

```

(continues on next page)

(continued from previous page)

```

11 // Request the `open` and `close` prices using the `renkoTickerID`.
12 [renkoOpen, renkoClose] = request.security(renkoTickerID, timeframe.period, [open,_
13   ↪close])
14
15 // @variable When `true`, the strategy places a long market order.
16 bool longEntry = ta.crossover(renkoClose, renkoOpen)
17 // @variable When `true`, the strategy places a short market order.
18 bool shortEntry = ta.crossunder(renkoClose, renkoOpen)
19
20 if longEntry
21   strategy.entry("Long Entry", strategy.long)
22 if shortEntry
23   strategy.entry("Short Entry", strategy.short)
24
25 plot(renkoOpen)
26 plot(renkoClose)

```

#### 4.14.7 Historical and realtime behavior

Functions in the `request.*()` namespace can behave differently on historical and realtime bars. This behavior is closely related to Pine's [Execution model](#).

Consider how a script behaves within the main context. Throughout the chart's history, the script calculates its required values once and *commits* them to that bar so their states are accessible later in the execution. On an unconfirmed bar, however, the script recalculates its values on *each update* to the bar's data to align with realtime changes. Before recalculating the values on that bar, it reverts calculated values to their last committed states, otherwise known as *rollback*, and it only commits values to that bar once the bar closes.

Now consider the behavior of data requests from other contexts with `request.security()`. As when evaluating historical bars in the main context, `request.security()` only returns new historical values when it confirms a bar in its specified context. When executing on realtime bars, it returns recalculated values on each chart bar, similar to how a script recalculates values in the main context on the open chart bar.

However, the function only *confirms* the requested values when a bar from its context closes. When the script restarts its execution, what were previously considered *realtime* bars become *historical* bars. Therefore, `request.security()` will only return the values it confirmed on those bars. In essence, this behavior means that requested data may *repaint* when its values fluctuate on realtime bars without confirmation from the context.

---

**Note:** It's often helpful to distinguish historical bars from realtime bars when working with `request.*()` functions. Scripts can determine whether bars have historical or realtime states via the `barstate.ishistory` and `barstate.isrealtime` variables.

---

In most circumstances where a script requests data from a broader context, one will typically require confirmed, stable values that *do not* fluctuate on realtime bars. The [section below](#) explains how to achieve such a result and avoid repainting data requests.

## Avoiding Repainting

### Higher-timeframe data

When requesting values from a higher timeframe, they are subject to repainting since realtime bars can contain *unconfirmed* information from developing HTF bars, and the script may adjust the times that new values come in on historical bars. To avoid repainting HTF data, one must ensure that the function only returns confirmed values with consistent timing on all bars, regardless of bar state.

The most reliable approach to achieve non-repainting results is to use an `expression` argument that only references past bars (e.g., `close[1]`) while using `barmerge.lookahead_on` as the `lookahead` value.

Using `barmerge.lookahead_on` with non-offset HTF data requests is discouraged since it prompts `request.security()` to “look ahead” to the final values of an HTF bar, retrieving confirmed values *before* they’re actually available in the script’s history. However, if the values used in the `expression` are offset by at least one bar, the “future” data the function retrieves is no longer from the future. Instead, the data represents confirmed values from established, *available* HTF bars. In other words, applying an offset to the `expression` effectively prevents the requested data from repainting when the script restarts its execution and eliminates lookahead bias in the historical series.

The following example demonstrates a repainting HTF data request. The script uses `request.security()` without offset modifications or additional arguments to retrieve the results of a `ta.wma()` call from a higher timeframe. It also highlights the background to indicate which bars were in a realtime state during its calculations.

As shown on the chart below, the `plot` of the requested WMA only changes on historical bars when HTF bars close, whereas it fluctuates on all realtime bars since the data includes unconfirmed values from the higher timeframe:



```

1 //@version=5
2 indicator("Avoiding HTF repainting demo", overlay = true)
3
4 //@variable The multiplier applied to the chart's timeframe.
5 int tfMultiplier = input.int(10, "Timeframe multiplier", 1)
6 //@variable The number of bars in the moving average.
7 int length = input.int(5, "WMA smoothing length")
8
9 //@variable The valid timeframe string closest to `tfMultiplier` times larger than
10 //→ the chart timeframe.
11 string timeframe = timeframe.from_seconds(timeframe.in_seconds() * tfMultiplier)
12 //@variable The weighted MA of `close` prices over `length` bars on the `timeframe`.

```

(continues on next page)

(continued from previous page)

```

13 // This request repaints because it includes unconfirmed HTF data on
14 // realtime bars and it may offset the
15 // times of its historical results.
16 float requestedWMA = request.security(syminfo.tickerid, timeframe, ta.wma(close,
17 //length))
18
19 // Plot the requested series.
20 plot(requestedWMA, "HTF WMA", color.purple, 3)
21 // Highlight the background on realtime bars.
22 bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime bar
23 highlight")

```

To avoid repainting in this script, we can add `lookahead = barmerge.lookahead_on` to the `request.security()` call and offset the call history of `ta.wma()` by one bar with the history-referencing operator `[1]`, ensuring the request always retrieves the last confirmed HTF bar's WMA at the start of each new `timeframe`. Unlike the previous script, this version has consistent behavior on historical and realtime bar states, as we see below:



```

1 // @version=5
2 indicator("Avoiding HTF repainting demo", overlay = true)
3
4 // @variable The multiplier applied to the chart's timeframe.
5 int tfMultiplier = input.int(10, "Timeframe multiplier", 1)
6 // @variable The number of bars in the moving average.
7 int length = input.int(5, "WMA smoothing length")
8
9 // @variable The valid timeframe string closest to `tfMultiplier` times larger than
10 // the chart timeframe.
11 string timeframe = timeframe.from_seconds(timeframe.in_seconds() * tfMultiplier)
12
13 // @variable The weighted MA of `close` prices over `length` bars on the `timeframe`.
14 // This request does not repaint, as it always references the last confirmed
15 // WMA value on all bars.
16 float requestedWMA = request.security(
17     syminfo.tickerid, timeframe, ta.wma(close, length)[1], lookahead = barmerge.
18     lookahead_on
19 )
20
21 // Plot the requested value.

```

(continues on next page)

(continued from previous page)

```

19 plot(requestedWMA, "HTF WMA", color.purple, 3)
20 // Highlight the background on realtime bars.
21 bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime bar"
  ↵highlight")

```

## Lower-timeframe data

The `request.security()` and `request.security_lower_tf()` functions can retrieve data from lower-timeframe contexts. The `request.security()` function can only retrieve data from a *single* intrabar in each chart bar, and `request.security_lower_tf()` retrieves data from *all* available intrabars.

When using these functions to retrieve intrabar data, it's important to note that such requests are **not** immune to repainting behavior. Historical and realtime series often rely on *separate* data feeds. Data providers may retroactively modify realtime data, and it's possible for races to occur in realtime data feeds, as explained in the [Data feeds](#) section of this page. Either case may result in intrabar data retrieved on realtime bars repainting after the script restarts its execution.

Additionally, a particular case that *will* cause repainting LTF requests is using `request.security()` with `barmerge.lookahead_on` to retrieve data from the first intrabar in each chart bar. While it will generally work as expected on historical bars, it will track only the most recent intrabar on realtime bars, as `request.security()` does not retain all intrabar information, and the intrabars retrieved by the function on realtime bars are unsorted until restarting the script's execution:



```

1 //@version=5
2 indicator("Avoiding LTF repainting demo", overlay = true)
3
4 //@variable The lower timeframe of the requested data.
5 string lowerTimeframe = input.timeframe("1", "Timeframe")
6
7 //@variable The first intrabar `close` requested from the `lowerTimeframe` on each
  ↵bar.
8 // Only works as intended on historical bars.
9 float requestedClose = request.security(syminfo.tickerid, lowerTimeframe, close,
  ↵lookahead = barmerge.lookahead_on)
10
11 // Plot the `requestedClose`.
12 plot(requestedClose, "First intrabar close", linewidth = 3)

```

(continues on next page)

(continued from previous page)

```

13 // Highlight the background on realtime bars.
14 bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar
  ↵Highlight")

```

One can mitigate this behavior and track the values from the first intrabar, or any available intrabar in the chart bar, by using `request.security_lower_tf()` since it maintains an array of intrabar values ordered by the times they come in. Here, we call `array.first()` on a requested array of intrabar data to retrieve the close price from the first available intrabar in each chart bar:



```

1 //@version=5
2 indicator("Avoiding LTF repainting demo", overlay = true)
3
4 //@variable The lower timeframe of the requested data.
5 string lowerTimeframe = input.timeframe("1", "Timeframe")
6
7 //@variable An array of intrabar `close` values requested from the `lowerTimeframe`
  ↵on each bar.
8 array<float> requestedCloses = request.security_lower_tf(syminfo.tickerid,
  ↵lowerTimeframe, close)
9
10 //@variable The first intrabar `close` on each bar with available data.
11 float firstClose = requestedCloses.size() > 0 ? requestedCloses.first() : na
12
13 // Plot the `firstClose`.
14 plot(firstClose, "First intrabar close", linewidth = 3)
15 // Highlight the background on realtime bars.
16 bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar
  ↵Highlight")

```

#### Note that:

- While `request.security_lower_tf()` is more optimized for handling historical and realtime intrabars, it's still possible in some cases for minor repainting to occur due to data differences from the provider, as outlined above.
- This code may not show intrabar data on all available chart bars, depending on how many intrabars each chart bar contains, as `request.*()` functions can retrieve up to 100,000 intrabars from an LTF context. See [this](#) section of the [Limitations](#) page for more information.

#### 4.14.8 `request.currency\_rate()`

When a script needs to convert values expressed in one currency to another, one can use `request.currency_rate()`. This function requests a *daily rate* for currency conversion calculations based on “FX\_IDC” data, providing a simpler alternative to fetching specific pairs or spreads with `request.security()`.

While one can use `request.security()` to retrieve daily currency rates, its use case is more involved than `request.currency_rate()`, as one needs to supply a valid *ticker ID* for a currency pair or spread to request the rate. Additionally, a historical offset and `barmerge.lookahead_on` are necessary to prevent the results from repainting, as explained in [this section](#).

The `request.currency_rate()` function, on the other hand, only requires *currency codes*. No ticker ID is needed when requesting rates with this function, and it ensures non-repainting results without requiring additional specification.

The function’s signature is as follows:

```
request.currency_rate(from, to, ignore_invalid_currency) → series float
```

The `from` parameter specifies the currency to convert, and the `to` parameter specifies the target currency. Both parameters accept “string” values in the [ISO 4217](#) format (e.g., “USD”) or any built-in `currency.*` variable (e.g., `currency.USD`).

When the function cannot calculate a valid conversion rate between the `from` and `to` currencies supplied in the call, one can decide whether it will raise a runtime error or return `na` via the `ignore_invalid_currency` parameter. The default value is `false`, meaning the function will raise a runtime error and halt the script’s execution.

The following example demonstrates a simple use case for `request.currency_rate()`. Suppose we want to convert values expressed in Turkish lira (`currency.TRY`) to South Korean won (`currency.KRW`) using a daily conversion rate. If we use `request.security()` to retrieve the rate, we must supply a valid ticker ID and request the last confirmed `close` from the previous day.

In this case, no “FX\_IDC” symbol exists that would allow us to retrieve a conversion rate directly with `request.security()`. Therefore, we first need a ticker ID for a `spread` that converts TRY to an intermediate currency, such as USD, then converts the intermediate currency to KRW. We can then use that ticker ID within `request.security()` with `close[1]` as the `expression` and `barmerge.lookahead_on` as the `lookahead` value to request a non-repainting daily rate.

Alternatively, we can achieve the same result more simply by calling `request.currency_rate()`. This function does all the heavy lifting for us, only requiring `from` and `to` currency arguments to perform its calculation.

As we see below, both approaches return the same daily rate:



```

1 // @version=5
2 indicator("Requesting currency rates demo")
3
4 //@variable The currency to convert.
5 simple string fromCurrency = currency.TRY
6 //@variable The resulting currency.
7 simple string toCurrency = currency.KRW
8
9 //@variable The spread symbol to request. Required in `request.security()` since no_
10 //direct "FX_IDC" rate exists.
11 simple string spreadSymbol = str.format("FX_IDC:{0}{2} * FX_IDC:{2}{1}", fromCurrency,
12 // toCurrency, currency.USD)
13
14 //@variable The non-repainting conversion rate from `request.security()` using the_
15 // `spreadSymbol`.
16 float securityRequestedRate = request.security(spreadSymbol, "1D", close[1],_
17 //lookahead = barmerge.lookahead_on)
18 //@variable The non-repainting conversion rate from `request.currency_rate()` .
19 float nonSecurityRequestedRate = request.currency_rate(fromCurrency, toCurrency)
20
21 // Plot the requested rates. We can multiply TRY values by these rates to convert_
22 //them to KRW.
23 plot(securityRequestedRate, "`request.security()` value", color.purple, 5)
24 plot(nonSecurityRequestedRate, "`request.currency_rate()` value", color.yellow, 2)

```

#### 4.14.9 `request.dividends()`, `request.splits()`, and `request.earnings()`

Analyzing a stock's earnings data and corporate actions provides helpful insights into its underlying financial strength. Pine Script™ provides the ability to retrieve essential information about applicable stocks via `request.dividends()`, `request.splits()`, and `request.earnings()`.

These are the functions' signatures:

```

request.dividends(ticker, field, gaps, lookahead, ignore_invalid_symbol, currency) →_
→series float

request.splits(ticker, field, gaps, lookahead, ignore_invalid_symbol) → series float

request.earnings(ticker, field, gaps, lookahead, ignore_invalid_symbol, currency) →_
→series float

```

Each function has the same parameters in its signature, with the exception of `request.splits()`, which doesn't have a `currency` parameter.

Note that unlike the `symbol` parameter in other `request.*()` functions, the `ticker` parameter in these functions only accepts an *"Exchange:Symbol"* pair, such as "NASDAQ:AAPL". The built-in `syminfo.ticker` variable does not work with these functions since it does not contain exchange information. Instead, one must use `syminfo.tickerid` for such cases.

The `field` parameter determines the data the function will retrieve. Each of these functions accepts different built-in variables as the `field` argument since each requests different information about a stock:

- The `request.dividends()` function retrieves current dividend information for a stock, i.e., the amount per share the issuing company paid out to investors who purchased shares before the ex-dividend date. Passing the built-in `dividends.gross` or `dividends.net` variables to the `field` parameter specifies whether the returned value represents dividends before or after factoring in expenses the company deducts from its payouts.

- The `request.splits()` function retrieves current split and reverse split information for a stock. A split occurs when a company increases its outstanding shares to promote liquidity. A reverse split occurs when a company consolidates its shares and offers them at a higher price to attract specific investors or maintain their listing on a market that has a minimum per-share price. Companies express their split information as *ratios*. For example, a 5:1 split means the company issued additional shares to its shareholders so that they have five times the number of shares they had before the split, and the raw price of each share becomes one-fifth of the previous price. Passing `splits.numerator` or `splits.denominator` to the `field` parameter of `request.splits()` determines whether it returns the numerator or denominator of the split ratio.
- The `request.earnings()` function retrieves the earnings per share (EPS) information for a stock `ticker`'s issuing company. The EPS value is the ratio of a company's net income to the number of outstanding stock shares, which investors consider an indicator of the company's profitability. Passing `earnings.actual`, `earnings.estimate`, or `earnings.standardized` as the `field` argument in `request.earnings()` respectively determines whether the function requests the actual, estimated, or standardized EPS value.

For a detailed explanation of the `gaps`, `lookahead`, and `ignore_invalid_symbol` parameters of these functions, see the [Common characteristics](#) section at the top of this page.

It's important to note that the values returned by these functions reflect the data available as it comes in. This behavior differs from financial data originating from a `request.financial()` call in that the underlying data from such calls becomes available according to a company's fiscal reporting period.

---

**Note:** Scripts can also retrieve information about upcoming earnings and dividends for an instrument via the `earnings.future_*` and `dividends.future_*` built-in variables.

---

Here, we've included an example that displays a handy `table` containing the most recent dividend, split, and EPS data. The script calls the `request.*()` functions discussed in this section to retrieve the data, then converts the values to "strings" with `str.*()` functions and displays the results in the `infoTable` with `table.cell()`:



```

1 // @version=5
2 indicator("Dividends, splits, and earnings demo", overlay = true)
3
4 // @variable The size of the table's text.
5 string tableSize = input.string(
6     size.large, "Table size", [size.auto, size.tiny, size.small, size.normal, size.
7     large, size.huge]
8 )

```

(continues on next page)

(continued from previous page)

```

8 // @variable The color of the table's text and frame.
9 var color tableColor = chart.fg_color
10 // @variable A `table` displaying the latest dividend, split, and EPS information.
11 var table infoTable = table.new(position.top_right, 3, 4, frame_color = tableColor,_
12   ↪frame_width = 1)
13
14 // Add header cells on the first bar.
15 if barstate.isfirst
16   table.cell(infoTable, 0, 0, "Field", text_color = tableColor, text_size =_
17   ↪tableSize)
18   table.cell(infoTable, 1, 0, "Value", text_color = tableColor, text_size =_
19   ↪tableSize)
20   table.cell(infoTable, 2, 0, "Date", text_color = tableColor, text_size =_
21   ↪tableSize)
22   table.cell(infoTable, 0, 1, "Dividend", text_color = tableColor, text_size =_
23   ↪tableSize)
24   table.cell(infoTable, 0, 2, "Split", text_color = tableColor, text_size =_
25   ↪tableSize)
26   table.cell(infoTable, 0, 3, "EPS", text_color = tableColor, text_size = tableSize)
27
28 // @variable The amount of the last reported dividend as of the current bar.
29 float latestDividend = request.dividends(syminfo.tickerid, dividends.gross, barmerge.-
30   ↪gaps_on)
31 // @variable The numerator of that last reported split ratio as of the current bar.
32 float latestSplitNum = request.splits(syminfo.tickerid, splits.numerator, barmerge.-
33   ↪gaps_on)
34 // @variable The denominator of the last reported split ratio as of the current bar.
35 float latestSplitDen = request.splits(syminfo.tickerid, splits.denominator, barmerge.-
36   ↪gaps_on)
37 // @variable The last reported earnings per share as of the current bar.
38 float latestEPS = request.earnings(syminfo.tickerid, earnings.actual, barmerge.gaps_-
39   ↪on)
40
41 // Update the "Value" and "Date" columns when new values come in.
42 if not na(latestDividend)
43   table.cell(
44     infoTable, 1, 1, str.tostring(math.round(latestDividend, 3)), text_color =_
45     ↪tableColor, text_size = tableSize
46   )
47   table.cell(infoTable, 2, 1, str.format_time(time, "yyyy-MM-dd"), text_color =_
48     ↪tableColor, text_size = tableSize)
49 if not na(latestSplitNum)
50   table.cell(
51     infoTable, 1, 2, str.format("{0}-for-{1}", latestSplitNum, latestSplitDen),_
52     ↪text_color = tableColor,
53     text_size = tableSize
54   )
55   table.cell(infoTable, 2, 2, str.format_time(time, "yyyy-MM-dd"), text_color =_
56     ↪tableColor, text_size = tableSize)
57 if not na(latestEPS)
58   table.cell(infoTable, 1, 3, str.tostring(latestEPS), text_color = tableColor,_
59     ↪text_size = tableSize)
60   table.cell(infoTable, 2, 3, str.format_time(time, "yyyy-MM-dd"), text_color =_
61     ↪tableColor, text_size = tableSize)

```

**Note that:**

- We've included `barmerge.gaps_on` in the `request.*()` calls, so they only return values when new data is available. Otherwise, they return `na`.
- The script assigns a `table` ID to the `infoTable` variable on the first chart bar. On subsequent bars, it updates necessary cells with new information whenever data is available.
- If no information is available from any of the `request.*()` calls throughout the chart's history (e.g., if the `ticker` has no dividend information), the script does not initialize the corresponding cells since it's unnecessary.

### 4.14.10 `request.quandl()`

TradingView forms partnerships with many fintech companies to provide users access to extensive information on financial instruments, economic data, and more. One of our many partners is [Nasdaq Data Link](#) (formerly Quandl), which provides multiple *external* data feeds that scripts can access via the `request.quandl()` function.

Here is the function's signature:

```
request.quandl(ticker, gaps, index, ignore_invalid_symbol) → series float
```

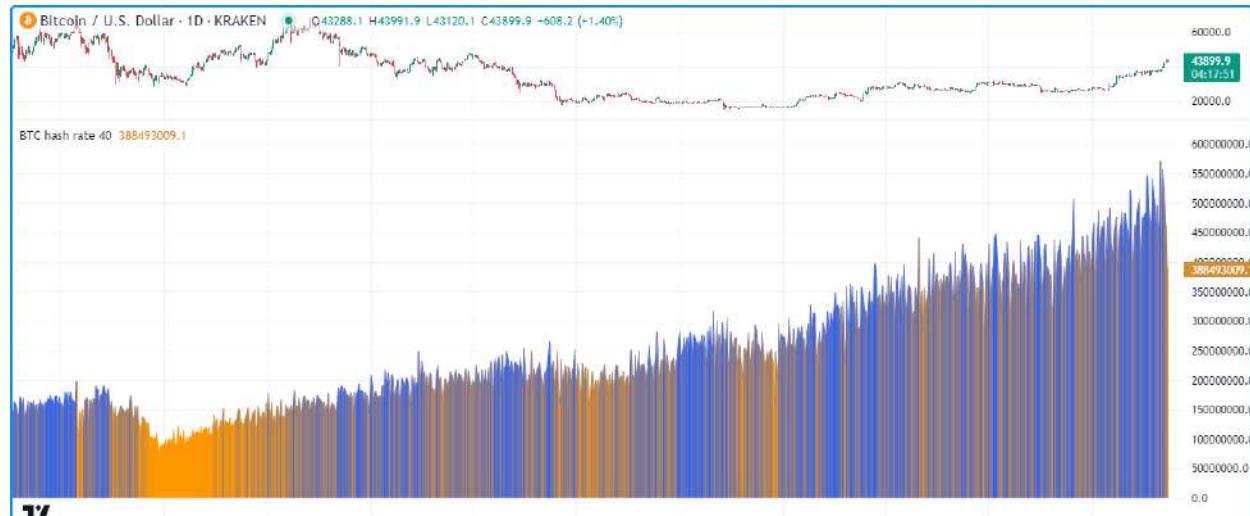
The `ticker` parameter accepts a “simple string” representing the ID of the database published on Nasdaq Data Link and its time series code, separated by the “/” delimiter. For example, the code “FRED/DFF” represents the “Effective Federal Funds Rate” time series from the “Federal Reserve Economic Data” database.

The `index` parameter accepts a “simple int” representing the *column index* of the requested data, where 0 is the first available column. Consult the database's documentation on Nasdaq Data Link's website to see available columns.

For details on the `gaps` and `ignore_invalid_symbol` parameters, see the [Common characteristics](#) section of this page.

**Note:** The `request.quandl()` function can only request **free** data from Nasdaq Data Link. No data that requires a paid subscription to their services is accessible with this function. Nasdaq Data Link may change the data it provides over time, and they may not update available datasets regularly. Therefore, it's up to programmers to research the supported data available for request and review the documentation provided for each dataset. You can search for free data [here](#).

This script requests Bitcoin hash rate (“HRATE”) information from the “Bitcoin Data Insights” (“BCHAIN”) database and `plots` the retrieved time series data on the chart. It uses `color.from_gradient()` to color the `area` plot based on the distance from the current hash rate to its all-time high:



```

1 // @version=5
2 indicator("Quandl demo", "BTC hash rate")
3
4 //@variable The estimated hash rate for the Bitcoin network.
5 float hashRate = request.quandl("BCHAIN/HRATE", barmerge.gaps_off, 0)
6 //@variable The percentage threshold from the all-time highest `hashRate`.
7 float dropThreshold = input.int(40, "Drop threshold", 0, 100)
8
9 //@variable The all-time highest `hashRate`.
10 float maxHashRate = ta.max(hashRate)
11 //@variable The value `dropThreshold` percent below the `maxHashRate`.
12 float minHashRate = maxHashRate * (100 - dropThreshold) / 100
13 //@variable The color of the plot based on the `minHashRate` and `maxHashRate`.
14 color plotColor = color.from_gradient(hashRate, minHashRate, maxHashRate, color.
15   orange, color.blue)
16
17 // Plot the `hashRate`.
18 plot(hashRate, "Hash Rate Estimate", plotColor, style = plot.style_area)

```

#### 4.14.11 `request.financial()`

Financial metrics provide investors with insights about a company's economic and financial health that are not tangible from solely analyzing its stock prices. TradingView offers a wide variety of financial metrics from FactSet that traders can access via the “Financials” tab in the “Indicators” menu of the chart. Scripts can access available metrics for an instrument directly via the `request.financial()` function.

This is the function’s signature:

```
request.financial(symbol, financial_id, period, gaps, ignore_invalid_symbol,_
  ↪ currency) → series float
```

As with the first parameter in `request.dividends()`, `request.splits()`, and `request.earnings()`, the `symbol` parameter in `request.financial()` requires an “*Exchange:Symbol*” pair. To request financial information for the chart’s ticker ID, use `syminfo.tickerid`, as `syminfo.ticker` will not work.

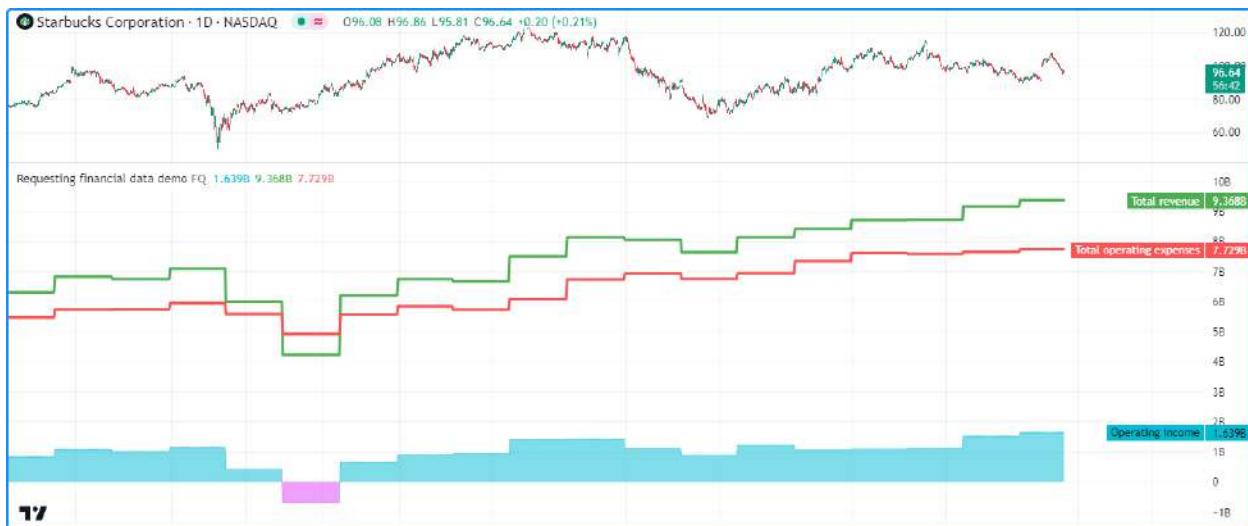
The `financial_id` parameter accepts a “simple string” representing the ID of the requested financial metric. TradingView has numerous financial metrics to choose from. See the *Financial IDs* section below for an overview of all accessible metrics and their “string” identifiers.

The `period` parameter specifies the fiscal period for which new requested data comes in. It accepts one of the following arguments: “**FQ**” (quarterly), “**FH**” (semianual), “**FY**” (annual), or “**TTM**” (trailing twelve months). Not all fiscal periods are available for all metrics or instruments. To confirm which periods are available for specific metrics, see the second column of the tables in the *Financial IDs* section.

See this page’s *Common characteristics* section for a detailed explanation of this function’s `gaps`, `ignore_invalid_symbol`, and `currency` parameters.

It’s important to note that the data retrieved from this function comes in at a *fixed frequency*, independent of the precise date on which the data is made available within a fiscal period. For a company’s dividends, splits, and earnings per share (EPS) information, one can request data reported on exact dates via `request.dividends()`, `request.splits()`, and `request.earnings()`.

This script uses `request.financial()` to retrieve information about the income and expenses of a stock’s issuing company and visualize the profitability of its typical business operations. It requests the “**OPER\_INCOME**”, “**TOTAL\_REVENUE**”, and “**TOTAL\_OPER\_EXPENSE**” *financial IDs* for the `syminfo.tickerid` over the latest `fiscalPeriod`, then *plots* the results on the chart:



```

1 // @version=5
2 indicator("Requesting financial data demo", format = format.volume)
3
4 // @variable The size of the fiscal reporting period. Some options may not be
5 // available, depending on the instrument.
6 string fiscalPeriod = input.string("FQ", "Period", ["FQ", "FH", "FY", "TTM"])
7
8 float operatingIncome = request.financial(syminfo.tickerid, "OPER_INCOME",
9 // fiscalPeriod)
10 float totalRevenue = request.financial(syminfo.tickerid, "TOTAL_REVENUE",
11 // fiscalPeriod)
12 float totalExpenses = request.financial(syminfo.tickerid, "TOTAL_OPER_EXPENSE",
13 // fiscalPeriod)
14
15 // @variable Is aqua when the `totalRevenue` exceeds the `totalExpenses`, fuchsia
16 // otherwise.
17 color incomeColor = operatingIncome > 0 ? color.aqua, 50 : color.new(color.
18 // fuchsia, 50)
19
20 // Display the requested data.
21 plot(operatingIncome, "Operating income", incomeColor, 1, plot.style_area)
22 plot(totalRevenue, "Total revenue", color.green, 3)
23 plot(totalExpenses, "Total operating expenses", color.red, 3)

```

### Note that:

- Not all `fiscalPeriod` options are available for every ticker ID. For example, companies in the US typically publish *quarterly* reports, whereas many European companies publish *semiannual* reports. See [this page](#) in our Help Center for more information.

## Calculating financial metrics

The `request.financial()` function can provide scripts with numerous useful financial metrics that don't require additional calculations. However, some commonly used financial estimates require combining an instrument's current market price with requested financial data. Such is the case for:

- Market Capitalization (market price \* total shares outstanding)
- Earnings Yield (12-month EPS / market price)
- Price-to-Book Ratio (market price / BVPS)
- Price-to-Earnings Ratio (market price / EPS)
- Price-to-Sales Ratio (market cap / 12-month total revenue)

The following script contains *user-defined functions* that calculate the above financial metrics for the `syminfo.tickerid`. We've created these functions so users can easily copy them into their scripts. This example uses them within a `str.format()` call to construct a `tooltipText`, which it displays in tooltips on the chart using `labels`. Hovering over any bar's label will expose the tooltip containing the metrics calculated on that bar:



```

1 //@version=5
2 indicator("Calculating financial metrics demo", overlay = true, max_labels_count =_
3   ↪ 500)
4
5 //@function Calculates the market capitalization (market cap) for the chart's symbol.
6 marketCap() =>
7     //@variable The most recent number of outstanding shares reported for the symbol.
8     float totalSharesOutstanding = request.financial(syminfo.tickerid, "TOTAL_SHARES_"
9   ↪ OUTSTANDING", "FQ")
10    // Return the market cap value.
11    totalSharesOutstanding * close
12
13 //@function Calculates the Earnings Yield for the chart's symbol.
14 earningsYield() =>
15     //@variable The most recent 12-month earnings per share reported for the symbol.
16     float eps = request.financial(syminfo.tickerid, "EARNINGS_PER_SHARE", "TTM")
17     //Return the Earnings Yield percentage.
18     100.0 * eps / close

```

(continues on next page)

(continued from previous page)

```

18 // @function Calculates the Price-to-Book (P/B) ratio for the chart's symbol.
19 priceBookRatio() =>
20     // @variable The most recent Book Value Per Share (BVPS) reported for the symbol.
21     float bookValuePerShare = request.financial(syminfo.tickerid, "BOOK_VALUE_PER_
22     ↪SHARE", "FQ")
23     // Return the P/B ratio.
24     close / bookValuePerShare
25
26 // @function Calculates the Price-to-Earnings (P/E) ratio for the chart's symbol.
27 priceEarningsRatio() =>
28     // @variable The most recent 12-month earnings per share reported for the symbol.
29     float eps = request.financial(syminfo.tickerid, "EARNINGS_PER_SHARE", "TTM")
30     // Return the P/E ratio.
31     close / eps
32
33 // @function Calculates the Price-to-Sales (P/S) ratio for the chart's symbol.
34 priceSalesRatio() =>
35     // @variable The most recent number of outstanding shares reported for the symbol.
36     float totalSharesOutstanding = request.financial(syminfo.tickerid, "TOTAL SHARES_
37     ↪OUTSTANDING", "FQ")
38     // @variable The most recent 12-month total revenue reported for the symbol.
39     float totalRevenue = request.financial(syminfo.tickerid, "TOTAL_REVENUE", "TTM")
40     // Return the P/S ratio.
41     totalSharesOutstanding * close / totalRevenue
42
43 // @variable The text to display in label tooltips.
44 string tooltipText = str.format(
45     "Market Cap: {0} {1}\nEarnings Yield: {2}%\nP/B Ratio: {3}\nP/E Ratio: {4}\nP/S_
46     ↪Ratio: {5}",
47     str.tostring(marketCap(), format.volume), syminfo.currency, earningsYield(),_
48     ↪priceBookRatio(),
49     priceEarningsRatio(), priceSalesRatio()
50 )
51
52 // @variable Displays a blank label with a tooltip containing the `tooltipText`.
53 label info = label.new(chart.point.now(high), tooltip = tooltipText)

```

**Note that:**

- Since not all companies publish quarterly financial reports, one may need to change the “FQ” in these functions to match the minimum reporting period for a specific company, as the `request.financial()` calls will return `na` when “FQ” data isn’t available.

**Financial IDs**

Below is an overview of all financial metrics one can request via `request.financial()`, along with the periods in which reports may be available. We’ve divided this information into four tables corresponding to the categories displayed in the “Financials” section of the “Indicators” menu:

- *Income statements*
- *Balance sheet*
- *Cash flow*
- *Statistics*

Each table has the following three columns:

- The first column contains descriptions of each metric with links to Help Center pages for additional information.
- The second column lists the possible `period` arguments allowed for the metric. Note that all available values may not be compatible with specific ticker IDs, e.g., while “FQ” may be a possible argument, it will not work if the issuing company does not publish quarterly data.
- The third column lists the “string” IDs for the `financial_id` argument in `request.financial()`.

**Note:** The tables in these sections are quite lengthy, as there are many `financial_id` arguments available. Use the “Click to show/hide” option above each table to toggle its visibility.

## Income statements

This table lists the available metrics that provide information about a company’s income, costs, profits and losses.

Financial	period	financial_id
After tax other income/expense	FQ, FH, FY, TTM	AFTER_TAX_OTHER_INCOME
Average basic shares outstanding	FQ, FH, FY	BASIC SHARES OUTSTANDING
Basic earnings per share (Basic EPS)	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_BASIC
Cost of goods sold	FQ, FH, FY, TTM	COST_OF_GOODS
Deprecation and amortization	FQ, FH, FY, TTM	DEP_AMORT_EXP_INCOME_S
Diluted earnings per share (Diluted EPS)	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_DILUTED
Diluted net income available to common stockholders	FQ, FH, FY, TTM	DILUTED_NET_INCOME
Diluted shares outstanding	FQ, FH, FY	DILUTED SHARES OUTSTANDING
Dilution adjustment	FQ, FH, FY, TTM	DILUTION_ADJUSTMENT
Discontinued operations	FQ, FH, FY, TTM	DISCONTINUED_OPERATIONS
EBIT	FQ, FH, FY, TTM	EBIT
EBITDA	FQ, FH, FY, TTM	EBITDA
Equity in earnings	FQ, FH, FY, TTM	EQUITY_IN_EARNINGS
Gross profit	FQ, FH, FY, TTM	GROSS_PROFIT
Interest capitalized	FQ, FH, FY, TTM	INTEREST_CAPITALIZED
Interest expense on debt	FQ, FH, FY, TTM	INTEREST_EXPENSE_ON_DEBT
Interest expense, net of interest capitalized	FQ, FH, FY, TTM	NON_OPER_INTEREST_EXP
Miscellaneous non-operating expense	FQ, FH, FY, TTM	OTHER_INCOME
Net income	FQ, FH, FY, TTM	NET_INCOME
Net income before discontinued operations	FQ, FH, FY, TTM	NET_INCOME_BEF_DISC_OPER
Non-controlling/minority interest	FQ, FH, FY, TTM	MINORITY_INTEREST_EXP
Non-operating income, excl. interest expenses	FQ, FH, FY, TTM	NON_OPER_INCOME
Non-operating income, total	FQ, FH, FY, TTM	TOTAL_NON_OPER_INCOME
Non-operating interest income	FQ, FH, FY, TTM	NON_OPER_INTEREST_INCOME
Operating expenses (excl. COGS)	FQ, FH, FY, TTM	OPERATING_EXPENSES
Operating income	FQ, FH, FY, TTM	OPER_INCOME
Other cost of goods sold	FQ, FH, FY, TTM	COST_OF_GOODS_EXCL_DEP_AMORT
Other operating expenses, total	FQ, FH, FY, TTM	OTHER_OPER_EXPENSE_TOTAL
Preferred dividends	FQ, FH, FY, TTM	PREFERRED_DIVIDENDS
Pretax equity in earnings	FQ, FH, FY, TTM	PRETAX_EQUITY_IN_EARNINGS
Pretax income	FQ, FH, FY, TTM	PRETAX_INCOME
Research & development	FQ, FH, FY, TTM	RESEARCH_AND_DEV
Selling/general/admin expenses, other	FQ, FH, FY, TTM	SELL_GEN_ADMIN_EXP_OTHER
Selling/general/admin expenses, total	FQ, FH, FY, TTM	SELL_GEN_ADMIN_EXP_TOTAL
Taxes	FQ, FH, FY, TTM	INCOME_TAX

continues on next page

Table 1 – continued from previous page

Financial	period	financial_id
Total operating expenses	FQ, FH, FY, TTM	TOTAL_OPER_EXPENSE
Total revenue	FQ, FH, FY, TTM	TOTAL_REVENUE
Unusual income/expense	FQ, FH, FY, TTM	UNUSUAL_EXPENSE_INC

## Balance sheet

This table lists the metrics that provide information about a company's capital structure.

Financial	period	financial_id
Accounts payable	FQ, FH, FY	ACCOUNTS_PAYABLE
Accounts receivable - trade, net	FQ, FH, FY	ACCOUNTS_RECEIVABLES_NET
Accrued payroll	FQ, FH, FY	ACCRUED_PAYROLL
Accumulated depreciation, total	FQ, FH, FY	ACCUM_DEPREC_TOTAL
Additional paid-in capital/Capital surplus	FQ, FH, FY	ADDITIONAL_PAID_IN_CAPITAL
Book value per share	FQ, FH, FY	BOOK_VALUE_PER_SHARE
Capital and operating lease obligations	FQ, FH, FY	CAPITAL_OPERATINGLEASE_OBLIGATIONS
Capitalized lease obligations	FQ, FH, FY	CAPITALLEASE_OBLIGATIONS
Cash & equivalents	FQ, FH, FY	CASH_N_EQUIVALENTS
Cash and short term investments	FQ, FH, FY	CASH_N_SHORT_TERM_INVEST
Common equity, total	FQ, FH, FY	COMMON_EQUITY_TOTAL
Common stock par/Carrying value	FQ, FH, FY	COMMON_STOCK_PAR
Current portion of LT debt and capital leases	FQ, FH, FY	CURRENT_PORT_DEBT_CAPITAL_LEASES
Deferred income, current	FQ, FH, FY	DEFERRED_INCOME_CURRENT
Deferred income, non-current	FQ, FH, FY	DEFERRED_INCOME_NON_CURRENT
Deferred tax assets	FQ, FH, FY	DEFERRED_TAX_ASSESTS
Deferred tax liabilities	FQ, FH, FY	DEFERRED_TAX_LIABILITIES
Dividends payable	FY	DIVIDENDS_PAYABLE
Goodwill, net	FQ, FH, FY	GOODWILL
Gross property/plant/equipment	FQ, FH, FY	PPE_TOTAL_GROSS
Income tax payable	FQ, FH, FY	INCOME_TAX_PAYABLE
Inventories - finished goods	FQ, FH, FY	INVENTORY_FINISHED_GOODS
Inventories - progress payments & other	FQ, FH, FY	INVENTORY_PROGRESS_PAYMENTS
Inventories - raw materials	FQ, FH, FY	INVENTORY_RAW_MATERIALS
Inventories - work in progress	FQ, FH, FY	INVENTORY_WORK_IN_PROGRESS
Investments in unconsolidated subsidiaries	FQ, FH, FY	INVESTMENTS_IN_UNCONCSOLIDATE
Long term debt	FQ, FH, FY	LONG_TERM_DEBT
Long term debt excl. lease liabilities	FQ, FH, FY	LONG_TERM_DEBT_EXCL_CAPITALLEASE
Long term investments	FQ, FH, FY	LONG_TERM_INVESTMENTS
Minority interest	FQ, FH, FY	MINORITY_INTEREST
Net debt	FQ, FH, FY	NET_DEBT
Net intangible assets	FQ, FH, FY	INTANGIBLES_NET
Net property/plant/equipment	FQ, FH, FY	PPE_TOTAL_NET
Note receivable - long term	FQ, FH, FY	LONG_TERM_NOTE_RECEIVABLE
Notes payable	FY	NOTES_PAYABLE_SHORT_TERM_DEBT
Operating lease liabilities	FQ, FH, FY	OPERATINGLEASE_LIABILITIES
Other common equity	FQ, FH, FY	OTHER_COMMON_EQUITY
Other current assets, total	FQ, FH, FY	OTHER_CURRENT_ASSETS_TOTAL
Other current liabilities	FQ, FH, FY	OTHER_CURRENT_LIABILITIES

continues on next page

Table 2 – continued from previous page

Financial	period	financial_id
Other intangibles, net	FQ, FH, FY	OTHER_INTANGIBLES_NET
Other investments	FQ, FH, FY	OTHER_INVESTMENTS
Other long term assets, total	FQ, FH, FY	LONG_TERM_OTHER_ASSETS_TOTAL
Other non-current liabilities, total	FQ, FH, FY	OTHER LIABILITYS_TOTAL
Other receivables	FQ, FH, FY	OTHER_RECEIVABLES
Other short term debt	FY	OTHER_SHORT_TERM_DEBT
Paid in capital	FQ, FH, FY	PAID_IN_CAPITAL
Preferred stock, carrying value	FQ, FH, FY	PREFERRED_STOCK_CARRYING_VALUE
Prepaid expenses	FQ, FH, FY	PREPAID_EXPENSES
Provision for risks & charge	FQ, FH, FY	PROVISION_F_RISKS
Retained earnings	FQ, FH, FY	RETAINED_EARNINGS
Shareholders' equity	FQ, FH, FY	SHRHLDERS_EQUITY
Short term debt	FQ, FH, FY	SHORT_TERM_DEBT
Short term debt excl. current portion of LT debt	FQ, FH, FY	SHORT_TERM_DEBT_EXCL_CURRENT_PORT
Short term investments	FQ, FH, FY	SHORT_TERM_INVEST
Tangible book value per share	FQ, FH, FY	BOOK_TANGIBLE_PER_SHARE
Total assets	FQ, FH, FY	TOTAL_ASSETS
Total current assets	FQ, FH, FY	TOTAL_CURRENT_ASSETS
Total current liabilities	FQ, FH, FY	TOTAL_CURRENT LIABILITYS
Total debt	FQ, FH, FY	TOTAL_DEBT
Total equity	FQ, FH, FY	TOTAL_EQUITY
Total inventory	FQ, FH, FY	TOTAL_INVENTORY
Total liabilities	FQ, FH, FY	TOTAL LIABILITYS
Total liabilities & shareholders' equities	FQ, FH, FY	TOTAL LIABILITYS_SHRHLDERS_EQUITY
Total non-current assets	FQ, FH, FY	TOTAL_NON_CURRENT_ASSETS
Total non-current liabilities	FQ, FH, FY	TOTAL_NON_CURRENT LIABILITYS
Total receivables, net	FQ, FH, FY	TOTAL_RECEIVABLES_NET
Treasury stock - common	FQ, FH, FY	TREASURY_STOCK_COMMON

## Cash flow

This table lists the available metrics that provide information about how cash flows through a company.

Financial	period	financial_id
Amortization	FQ, FH, FY, TTM	AMORTIZATION
Capital expenditures	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES
Capital expenditures - fixed assets	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES_FIXED_ASSETS
Capital expenditures - other assets	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES_OTHER_ASSETS
Cash from financing activities	FQ, FH, FY, TTM	CASH_F_FINANCING_ACTIVITIES
Cash from investing activities	FQ, FH, FY, TTM	CASH_F_INVESTING_ACTIVITIES
Cash from operating activities	FQ, FH, FY, TTM	CASH_F_OPERATING_ACTIVITIES
Change in accounts payable	FQ, FH, FY, TTM	CHANGE_IN_ACCOUNTS_PAYABLE
Change in accounts receivable	FQ, FH, FY, TTM	CHANGE_IN_ACCOUNTS_RECEIVABLE
Change in accrued expenses	FQ, FH, FY, TTM	CHANGE_IN_ACCRUED_EXPENSES
Change in inventories	FQ, FH, FY, TTM	CHANGE_IN_INVENTORIES
Change in other assets/liabilities	FQ, FH, FY, TTM	CHANGE_IN_OTHER_ASSETS
Change in taxes payable	FQ, FH, FY, TTM	CHANGE_IN_TAXES_PAYABLE
Changes in working capital	FQ, FH, FY, TTM	CHANGES_IN_WORKING_CAPITAL

continues on next page

Table 3 – continued from previous page

Financial	period	financial_id
Common dividends paid	FQ, FH, FY, TTM	COMMON_DIVIDENDS_CASH_FLOW
Deferred taxes (cash flow)	FQ, FH, FY, TTM	CASH_FLOW_DEFERRED_TAXES
Depreciation & amortization (cash flow)	FQ, FH, FY, TTM	CASH_FLOW_DEPRECIATION_N_AMORTIZATION
Depreciation/depletion	FQ, FH, FY, TTM	DEPRECIATION_DEPLETION
Financing activities - other sources	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOW_SOURCES
Financing activities - other uses	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOWUSES
Free cash flow	FQ, FH, FY, TTM	FREE_CASH_FLOW
Funds from operations	FQ, FH, FY, TTM	FUNDS_F_OPERATIONS
Investing activities - other sources	FQ, FH, FY, TTM	OTHER_INVESTING_CASH_FLOW_SOURCES
Investing activities - other uses	FQ, FH, FY	OTHER_INVESTING_CASH_FLOWUSES
Issuance of long term debt	FQ, FH, FY, TTM	SUPPLYING_OF_LONG_TERM_DEBT
Issuance/retirement of debt, net	FQ, FH, FY, TTM	ISSUANCE_OF_DEBT_NET
Issuance/retirement of long term debt	FQ, FH, FY, TTM	ISSUANCE_OF_LONG_TERM_DEBT
Issuance/retirement of other debt	FQ, FH, FY, TTM	ISSUANCE_OF_OTHER_DEBT
Issuance/retirement of short term debt	FQ, FH, FY, TTM	ISSUANCE_OF_SHORT_TERM_DEBT
Issuance/retirement of stock, net	FQ, FH, FY, TTM	ISSUANCE_OF_STOCK_NET
Net income (cash flow)	FQ, FH, FY, TTM	NET_INCOME_STARTING_LINE
Non-cash items	FQ, FH, FY, TTM	NON_CASH_ITEMS
Other financing cash flow items, total	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOW_ITEMS_TOTAL
Other investing cash flow items, total	FQ, FH, FY	OTHER_INVESTING_CASH_FLOW_ITEMS_TOTAL
Preferred dividends paid	FQ, FH, FY	PREFERRED_DIVIDENDS_CASH_FLOW
Purchase of investments	FQ, FH, FY, TTM	PURCHASE_OF_INVESTMENTS
Purchase/acquisition of business	FQ, FH, FY, TTM	PURCHASE_OF_BUSINESS
Purchase/sale of business, net	FQ, FH, FY	PURCHASE_SALE_BUSINESS
Purchase/sale of investments, net	FQ, FH, FY, TTM	PURCHASE_SALE_INVESTMENTS
Reduction of long term debt	FQ, FH, FY, TTM	REDUCTION_OF_LONG_TERM_DEBT
Repurchase of common & preferred stock	FQ, FH, FY, TTM	PURCHASE_OF_STOCK
Sale of common & preferred stock	FQ, FH, FY, TTM	SALE_OF_STOCK
Sale of fixed assets & businesses	FQ, FH, FY, TTM	SALES_OF_BUSINESS
Sale/maturity of investments	FQ, FH, FY	SALES_OF_INVESTMENTS
Total cash dividends paid	FQ, FH, FY, TTM	TOTAL_CASH_DIVIDENDS_PAID

## Statistics

This table contains a variety of statistical metrics, including commonly used financial ratios.

Financial	period	financial_id
Accruals	FQ, FH, FY	ACCRUALS_RATIO
Altman Z-score	FQ, FH, FY	ALTMAN_Z_SCORE
Asset turnover	FQ, FH, FY	ASSET_TURNOVER
Beneish M-score	FQ, FH, FY	BENEISH_M_SCORE
Buyback yield %	FQ, FH, FY	BUYBACK_YIELD
COGS to revenue ratio	FQ, FH, FY	COGS_TO_REVENUE
Cash conversion cycle	FQ, FY	CASH_CONVERSION_CYCLE
Cash to debt ratio	FQ, FH, FY	CASH_TO_DEBT
Current ratio	FQ, FH, FY	CURRENT_RATIO
Days inventory	FQ, FY	DAYS_INVENT
Days payable	FQ, FY	DAYS_PAY

continues on next page

Table 4 – continued from previous page

Financial	period	financial_id
Days sales outstanding	FQ, FY	DAY_SALES_OUT
Debt to EBITDA ratio	FQ, FH, FY	DEBT_TO_EBITDA
Debt to assets ratio	FQ, FH, FY	DEBT_TO_ASSET
Debt to equity ratio	FQ, FH, FY	DEBT_TO_EQUITY
Debt to revenue ratio	FQ, FH, FY	DEBT_TO_REVENUE
Dividend payout ratio %	FQ, FH, FY, TTM	DIVIDEND_PAYOUT_RATIO
Dividend yield %	FQ, FH, FY	DIVIDENDS_YIELD
Dividends per share - common stock primary issue	FQ, FH, FY, TTM	DPS_COMMON_STOCK_PRIM_ISSUE
EBITDA margin %	FQ, FH, FY, TTM	EBITDA_MARGIN
EPS basic one year growth	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_BASIC_ONE_YEAR_GROWTH
EPS diluted one year growth	FQ, FH, FY	EARNINGS_PER_SHARE_DILUTED_ONE_YEAR_GROWTH
EPS estimates	FQ, FH, FY	EARNINGS_ESTIMATE
Effective interest rate on debt %	FQ, FH, FY	EFFECTIVE_INTEREST_RATE_ON_DEBT
Enterprise value	FQ, FH, FY	ENTERPRISE_VALUE
Enterprise value to EBIT ratio	FQ, FH, FY	EV_EBIT
Enterprise value to EBITDA ratio	FQ, FH, FY	ENTERPRISE_VALUE_EBITDA
Enterprise value to revenue ratio	FQ, FH, FY	EV_REVENUE
Equity to assets ratio	FQ, FH, FY	EQUITY_TO_ASSET
Float shares outstanding	FY	FLOAT SHARES_OUTSTANDING
Free cash flow margin %	FQ, FH, FY	FREE_CASH_FLOW_MARGIN
Fulmer H factor	FQ, FY	FULMER_H_FACTOR
Goodwill to assets ratio	FQ, FH, FY	GOODWILL_TO_ASSET
Graham's number	FQ, FY	GRAHAM_NUMBERS
Gross margin %	FQ, FH, FY, TTM	GROSS_MARGIN
Gross profit to assets ratio	FQ, FY	GROSS_PROFIT_TO_ASSET
Interest coverage	FQ, FH, FY	INTERST_COVER
Inventory to revenue ratio	FQ, FH, FY	INVENT_TO_REVENUE
Inventory turnover	FQ, FH, FY	INVENT_TURNOVER
KZ index	FY	KZ_INDEX
Long term debt to total assets ratio	FQ, FH, FY	LONG_TERM_DEBT_TO_ASSETS
Net current asset value per share	FQ, FY	NCAVPS_RATIO
Net income per employee	FY	NET_INCOME_PER_EMPLOYEE
Net margin %	FQ, FH, FY, TTM	NET_MARGIN
Number of employees	FY	NUMBER_OF_EMPLOYEES
Operating earnings yield %	FQ, FH, FY	OPERATING_EARNINGS_YIELD
Operating margin %	FQ, FH, FY	OPERATING_MARGIN
PEG ratio	FQ, FY	PEG_RATIO
Piotroski F-score	FQ, FH, FY	PIOTROSKI_F_SCORE
Price earnings ratio forward	FQ, FY	PRICE_EARNINGS_FORWARD
Price sales ratio forward	FQ, FY	PRICE_SALES_FORWARD
Quality ratio	FQ, FH, FY	QUALITY_RATIO
Quick ratio	FQ, FH, FY	QUICK_RATIO
Research & development to revenue ratio	FQ, FH, FY	RESEARCH_AND DEVELOP_TO_REVENUE
Return on assets %	FQ, FH, FY	RETURN_ON_ASSETS
Return on common equity %	FQ, FH, FY	RETURN_ON_COMMON_EQUIITY
Return on equity %	FQ, FH, FY	RETURN_ON_EQUITY
Return on equity adjusted to book value %	FQ, FH, FY	RETURN_ON_EQUITY_ADJUST_TO_BOOK
Return on invested capital %	FQ, FH, FY	RETURN_ON_INVESTED_CAPITAL
Return on tangible assets %	FQ, FH, FY	RETURN_ON_TANG_ASSETS
Return on tangible equity %	FQ, FH, FY	RETURN_ON_TANG_EQUITY

continues on next page

Table 4 – continued from previous page

Financial	period	financial_id
Revenue estimates	FQ, FH, FY	SALES_ESTIMATES
Revenue one year growth	FQ, FH, FY, TTM	REVENUE_ONE_YEAR_GROWTH
Revenue per employee	FY	REVENUE_PER_EMPLOYEE
Shares buyback ratio %	FQ, FH, FY	SHARE_BUYBACK_RATIO
Sloan ratio %	FQ, FH, FY	SLOAN_RATIO
Springate score	FQ, FY	SPRINGATE_SCORE
Sustainable growth rate	FQ, FY	SUSTAINABLE_GROWTH_RATE
Tangible common equity ratio	FQ, FH, FY	TANGIBLE_COMMON_EQUITY_RATIO
Tobin's Q (approximate)	FQ, FH, FY	TOBIN_Q_RATIO
Total common shares outstanding	FQ, FH, FY	TOTAL_SHARES_OUTSTANDING
Zmijewski score	FQ, FY	ZMIJEWSKI_SCORE

#### 4.14.12 `request.economic()`

The `request.economic()` function provides scripts with the ability to retrieve economic data for a specified country or region, including information about the state of the economy (GDP, inflation rate, etc.) or of a particular industry (steel production, ICU beds, etc.).

Below is the signature for this function:

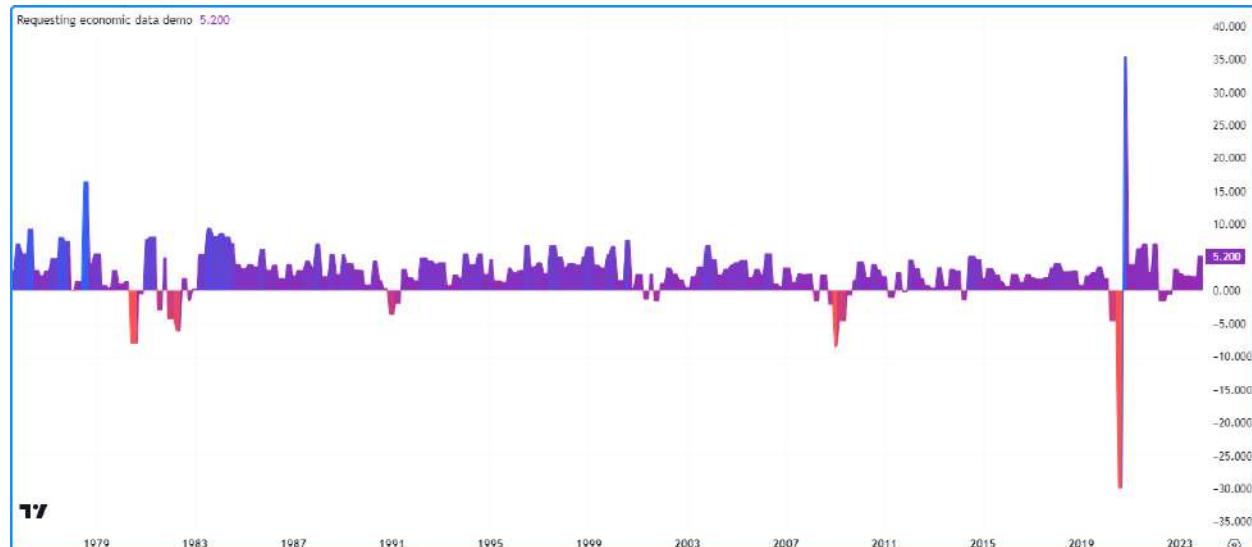
```
request.economic(country_code, field, gaps, ignore_invalid_symbol) → series float
```

The `country_code` parameter accepts a “simple string” representing the identifier of the country or region to request economic data for (e.g., “US”, “EU”, etc.). See the [Country/region codes](#) section for a complete list of codes this function supports. Note that the economic metrics available depend on the country or region specified in the function call.

The `field` parameter specifies the metric the function will request. The [Field codes](#) section covers all accessible metrics and the countries/regions they’re available for.

For a detailed explanation on the last two parameters of this function, see the [Common characteristics](#) section at the top of this page.

This simple example requests the growth rate of the Gross Domestic Product (“GDPQQ”) for the United States (“US”) using `request.economic()`, then *plots* its value on the chart with a gradient color:



```

1 //@version=5
2 indicator("Requesting economic data demo")
3
4 //@variable The GDP growth rate for the US economy.
5 float gdpqq = request.economic("US", "GDPQQ")
6
7 //@variable The all-time maximum growth rate.
8 float maxRate = ta.max(gdpqq)
9 //@variable The all-time minimum growth rate.
10 float minRate = ta.min(gdpqq)
11
12 //@variable The color of the `gdpqq` plot.
13 color rateColor = switch
14     gdpqq >= 0 => color.from_gradient(gdpqq, 0, maxRate, color.purple, color.blue)
15     => color.from_gradient(gdpqq, minRate, 0, color.red, color.purple)
16
17 // Plot the results.
18 plot(gdpqq, "US GDP Growth Rate", rateColor, style = plot.style_area)

```

**Note that:**

- This example does not include a `gaps` argument in the `request.economic()` call, so the function uses the default `barmerge.gaps_off`. In other words, it returns the last retrieved value when new data isn't yet available.

---

**Note:** The tables in the sections below are rather large, as there are numerous `country_code` and `field` arguments available. Use the “**Click to show/hide**” option above each table to toggle its visibility.

---

**Country/region codes**

The table in this section lists all country/region codes available for use with `request.economic()`. The first column of the table contains the “string” values that represent the country or region code, and the second column contains the corresponding country/region names.

It's important to note that the value used as the `country_code` argument determines which `field codes` are accessible to the function.

country_code	Country/region name
AF	Afghanistan
AL	Albania
DZ	Algeria
AD	Andorra
AO	Angola
AG	Antigua and Barbuda
AR	Argentina
AM	Armenia
AW	Aruba
AU	Australia
AT	Austria
AZ	Azerbaijan
BS	Bahamas
BH	Bahrain
BD	Bangladesh

continues on next page

Table 5 – continued from previous page

country_code	Country/region name
BB	Barbados
BY	Belarus
BE	Belgium
BZ	Belize
BJ	Benin
BM	Bermuda
BT	Bhutan
BO	Bolivia
BA	Bosnia and Herzegovina
BW	Botswana
BR	Brazil
BN	Brunei
BG	Bulgaria
BF	Burkina Faso
BI	Burundi
KH	Cambodia
CM	Cameroon
CA	Canada
CV	Cape Verde
KY	Cayman Islands
CF	Central African Republic
TD	Chad
CL	Chile
CN	China
CO	Colombia
KM	Comoros
CG	Congo
CR	Costa Rica
HR	Croatia
CU	Cuba
CY	Cyprus
CZ	Czech Republic
DK	Denmark
DJ	Djibouti
DM	Dominica
DO	Dominican Republic
TL	East Timor
EC	Ecuador
EG	Egypt
SV	El Salvador
GQ	Equatorial Guinea
ER	Eritrea
EE	Estonia
ET	Ethiopia
EU	Euro area
FO	Faroe Islands
FJ	Fiji
FI	Finland
FR	France
GA	Gabon

continues on next page

Table 5 – continued from previous page

country_code	Country/region name
GM	Gambia
GE	Georgia
DE	Germany
GH	Ghana
GR	Greece
GL	Greenland
GD	Grenada
GT	Guatemala
GN	Guinea
GW	Guinea Bissau
GY	Guyana
HT	Haiti
HN	Honduras
HK	Hong Kong
HU	Hungary
IS	Iceland
IN	India
ID	Indonesia
IR	Iran
IQ	Iraq
IE	Ireland
IM	Isle of Man
IL	Israel
IT	Italy
CI	Ivory Coast
JM	Jamaica
JP	Japan
JO	Jordan
KZ	Kazakhstan
KE	Kenya
KI	Kiribati
XK	Kosovo
KW	Kuwait
KG	Kyrgyzstan
LA	Laos
LV	Latvia
LB	Lebanon
LS	Lesotho
LR	Liberia
LY	Libya
LI	Liechtenstein
LT	Lithuania
LU	Luxembourg
MO	Macau
MK	Macedonia
MG	Madagascar
MW	Malawi
MY	Malaysia
MV	Maldives
ML	Mali

continues on next page

Table 5 – continued from previous page

country_code	Country/region name
MT	Malta
MR	Mauritania
MU	Mauritius
MX	Mexico
MD	Moldova
MC	Monaco
MN	Mongolia
ME	Montenegro
MA	Morocco
MZ	Mozambique
MM	Myanmar
NA	Namibia
NP	Nepal
NL	Netherlands
NC	New Caledonia
NZ	New Zealand
NI	Nicaragua
NE	Niger
NG	Nigeria
KP	North Korea
NO	Norway
OM	Oman
PK	Pakistan
PS	Palestine
PA	Panama
PG	Papua New Guinea
PY	Paraguay
PE	Peru
PH	Philippines
PL	Poland
PT	Portugal
PR	Puerto Rico
QA	Qatar
CD	Republic of the Congo
RO	Romania
RU	Russia
RW	Rwanda
WS	Samoa
SM	San Marino
ST	Sao Tome and Principe
SA	Saudi Arabia
SN	Senegal
RS	Serbia
SC	Seychelles
SL	Sierra Leone
SG	Singapore
SK	Slovakia
SI	Slovenia
SB	Solomon Islands
SO	Somalia

continues on next page

Table 5 – continued from previous page

country_code	Country/region name
ZA	South Africa
KR	South Korea
SS	South Sudan
ES	Spain
LK	Sri Lanka
LC	St Lucia
VC	St Vincent and the Grenadines
SD	Sudan
SR	Suriname
SZ	Swaziland
SE	Sweden
CH	Switzerland
SY	Syria
TW	Taiwan
TJ	Tajikistan
TZ	Tanzania
TH	Thailand
TG	Togo
TO	Tonga
TT	Trinidad and Tobago
TN	Tunisia
TR	Turkey
TM	Turkmenistan
UG	Uganda
UA	Ukraine
AE	United Arab Emirates
GB	United Kingdom
US	United States
UY	Uruguay
UZ	Uzbekistan
VU	Vanuatu
VE	Venezuela
VN	Vietnam
YE	Yemen
ZM	Zambia
ZW	Zimbabwe

## Field codes

The table in this section lists the field codes available for use with `request.economic()`. The first column contains the “string” values used as the `field` argument, and the second column contains names of each metric and links to our Help Center with additional information, including the countries/regions they’re available for.

field	Metric
AA	Asylum Applications
ACR	API Crude Runs
AE	Auto Exports
AHE	Average Hourly Earnings

continues on next page

Table 6 – continued from previous page

field	Metric
AHO	API Heating Oil
AWH	Average Weekly Hours
BBS	Banks Balance Sheet
BCLI	Business Climate Indicator
BCOI	Business Confidence Index
BI	Business Inventories
BLR	Bank Lending Rate
BOI	NFIB Business Optimism Index
BOT	Balance Of Trade
BP	Building Permits
BR	Bankruptcies
CA	Current Account
CAG	Current Account To GDP
CAP	Car Production
CAR	Car Registrations
CBBS	Central Bank Balance Sheet
CCC	Claimant Count Change
CCI	Consumer Confidence Index
CCOS	Cushing Crude Oil Stocks
CCP	Core Consumer Prices
CCPI	Core CPI
CCPT	Consumer Confidence Price Trends
CCR	Consumer Credit
CCS	Credit Card Spending
CEP	Cement Production
CF	Capital Flows
CFNAI	Chicago Fed National Activity Index
CI	API Crude Imports
CIND	Coincident Index
CIR	Core Inflation Rate, YoY
CJC	Continuing Jobless Claims
CN	API Cushing Number
COI	Crude Oil Imports
COIR	Crude Oil Imports from Russia
CONSTS	Construction Spending
COP	Crude Oil Production
COR	Crude Oil Rigs
CORD	Construction Orders, YoY
CORPI	Corruption Index
CORR	Corruption Rank
COSC	Crude Oil Stocks Change
COUT	Construction Output, YoY
CP	Copper Production
CPCEPI	Core PCE Price Index
CPI	Consumer Price Index
CPIHU	CPI Housing Utilities
CPIM	CPI Median
CPIT	CPI Transportation
CPITM	CPI Trimmed Mean
CPMI	Chicago PMI

continues on next page

Table 6 – continued from previous page

field	Metric
CPPI	Core Producer Price Index
CPR	Corporate Profits
CRLPI	Cereals Price Index
CRR	Cash Reserve Ratio
CS	Consumer Spending
CSC	API Crude Oil Stock Change
CSHPI	Case Shiller Home Price Index
CSHPIMM	Case Shiller Home Price Index, MoM
CSHPIYY	Case Shiller Home Price Index, YoY
CSS	Chain Store Sales
CTR	Corporate Tax Rate
CU	Capacity Utilization
DFMI	Dallas Fed Manufacturing Index
DFP	Distillate Fuel Production
DFS	Distillate Stocks
DFSI	Dallas Fed Services Index
DFSRI	Dallas Fed Services Revenues Index
DG	Deposit Growth
DGO	Durable Goods Orders
DGOED	Durable Goods Orders Excluding Defense
DGOET	Durable Goods Orders Excluding Transportation
DIR	Deposit Interest Rate
DPI	Disposable Personal Income
DRPI	Dairy Price Index
DS	API Distillate Stocks
DT	CBI Distributive Trades
EC	ADP Employment Change
ED	External Debt
EDBR	Ease Of Doing Business Ranking
EHS	Existing Home Sales
ELP	Electricity Production
EMC	Employment Change
EMCI	Employment Cost Index
EMP	Employed Persons
EMR	Employment Rate
EOI	Economic Optimism Index
EP	Export Prices
ESI	ZEW Economic Sentiment Index
EWS	Economy Watchers Survey
EXP	Exports
EXPYY	Exports, YoY
FAI	Fixed Asset Investment
FBI	Foreign Bond Investment
FDI	Foreign Direct Investment
FE	Fiscal Expenditure
FER	Foreign Exchange Reserves
FI	Food Inflation, YoY
FO	Factory Orders
FOET	Factory Orders Excluding Transportation
FPI	Food Price Index

continues on next page

Table 6 – continued from previous page

field	Metric
FSI	Foreign Stock Investment
FTE	Full Time Employment
FYGDGP	Full Year GDP Growth
GASP	Gasoline Prices
GBP	Government Budget
GBV	Government Budget Value
GCI	Competitiveness Index
GCR	Competitiveness Rank
GD	Government Debt
GDG	Government Debt To GDP
GDP	Gross Domestic Product
GDPA	GDP From Agriculture
GDPC	GDP From Construction
GDPCP	GDP Constant Prices
GDPD	GDP Deflator
GDPGA	GDP Growth Annualized
GDPMAN	GDP From Manufacturing
GDPMIN	GDP From Mining
GDPPA	GDP From Public Administration
GDPPC	GDP Per Capita
GDPPCP	GDP Per Capita, PPP
GDQQ	GDP Growth Rate
GDPS	GDP From Services
GDPSA	GDP Sales
GDPT	GDP From Transport
GDPU	GDP From Utilities
GDYYY	GDP, YoY
GDTPI	Global Dairy Trade Price Index
GFCF	Gross Fixed Capital Formation
GNP	Gross National Product
GP	Gold Production
GPA	Government Payrolls
GPRO	Gasoline Production
GR	Government Revenues
GRES	Gold Reserves
GS	API Gasoline Stocks
GSC	Grain Stocks Corn
GSCH	Gasoline Stocks Change
GSG	Government Spending To GDP
GSP	Government Spending
GSS	Grain Stocks Soy
GSW	Grain Stocks Wheat
GTB	Goods Trade Balance
HB	Hospital Beds
HDG	Households Debt To GDP
HDI	Households Debt To Income
HICP	Harmonised Index of Consumer Prices
HIRMM	Harmonised Inflation Rate, MoM
HIRYY	Harmonised Inflation Rate, YoY
HMI	NAHB Housing Market Index

continues on next page

Table 6 – continued from previous page

field	Metric
HOR	Home Ownership Rate
HOS	Heating Oil Stocks
HOSP	Hospitals
HPI	House Price Index
HPIMM	House Price Index, MoM
HPIYY	House Price Index, YoY
HS	Home Loans
HSP	Household Spending
HST	Housing Starts
IC	Changes In Inventories
ICUB	ICU Beds
IE	Inflation Expectations
IFOCC	IFO Assessment Of The Business Situation
IFOE	IFO Business Developments Expectations
IJC	Initial Jobless Claims
IMP	Imports
IMPYY	Imports, YoY
INBR	Interbank Rate
INTR	Interest Rate
IPA	IP Addresses
IPMM	Industrial Production, MoM
IPRI	Import Prices
IPYY	Industrial Production, YoY
IRMM	Inflation Rate, MoM
IRYY	Inflation Rate, YoY
IS	Industrial Sentiment
ISP	Internet Speed
JA	Job Advertisements
JAR	Jobs To Applications Ratio
JC	Challenger Job Cuts
JC4W	Jobless Claims, 4-Week Average
JO	Job Offers
JV	Job Vacancies
KFMI	Kansas Fed Manufacturing Index
LB	Loans To Banks
LC	Labor Costs
LEI	Leading Economic Index
LFPR	Labor Force Participation Rate
LG	Loan Growth, YoY
LIVRR	Liquidity Injections Via Reverse Repo
LMIC	LMI Logistics Managers Index Current
LMICI	LMI Inventory Costs
LMIF	LMI Logistics Managers Index Future
LMITP	LMI Transportation Prices
LMIWP	LMI Warehouse Prices
LPS	Loans To Private Sector
LR	Central Bank Lending Rate
LTUR	Long Term Unemployment Rate
LWF	Living Wage Family
LWI	Living Wage Individual

continues on next page

Table 6 – continued from previous page

field	Metric
M0	Money Supply M0
M1	Money Supply M1
M2	Money Supply M2
M3	Money Supply M3
MA	Mortgage Approvals
MAPL	Mortgage Applications
MCE	Michigan Consumer Expectations
MCEC	Michigan Current Economic Conditions
MD	Medical Doctors
ME	Military Expenditure
MGDPYY	Monthly GDP, YoY
MIE1Y	Michigan Inflation Expectations
MIE5Y	Michigan 5 Year Inflation Expectations
MIP	Mining Production, YoY
MMI	MBA Mortgage Market Index
MO	Machinery Orders
MP	Manufacturing Payrolls
MPI	Meat Price Index
MPRMM	Manufacturing Production, MoM
MPRYY	Manufacturing Production, YoY
MR	Mortgage Rate
MRI	MBA Mortgage Refinance Index
MS	Manufacturing Sales
MTO	Machine Tool Orders
MW	Minimum Wages
NDCGOEA	Orders For Non-defense Capital Goods Excluding Aircraft
NEGTB	Goods Trade Deficit With Non-EU Countries
NFP	Nonfarm Payrolls
NGI	Natural Gas Imports
NGIR	Natural Gas Imports from Russia
NGSC	Natural Gas Stocks Change
NHPI	Nationwide House Price Index
NHS	New Home Sales
NHSMM	New Home Sales, MoM
NMPMI	Non-Manufacturing PMI
NO	New Orders
NODXMM	Non-Oil Domestic Exports, MoM
NODXYY	Non-Oil Domestic Exports, YoY
NOE	Non-Oil Exports
NPP	Nonfarm Payrolls Private
NURS	Nurses
NYESMI	NY Empire State Manufacturing Index
OE	Oil Exports
OPI	Oils Price Index
PCEPI	PCE Price Index
PDG	Private Debt To GDP
PFMI	Philadelphia Fed Manufacturing Index
PHSIMM	Pending Home Sales Index, MoM
PHSIYY	Pending Home Sales Index, YoY
PI	Personal Income

continues on next page

Table 6 – continued from previous page

field	Metric
PIN	Private Investment
PIND	MBA Purchase Index
PITR	Personal Income Tax Rate
POP	Population
PPI	Producer Price Index
PPII	Producer Price Index Input
PPIMM	Producer Price Inflation, MoM
PPIYY	Producer Prices Index, YoY
PRI	API Product Imports
PROD	Productivity
PS	Personal Savings
PSC	Private Sector Credit
PSP	Personal Spending
PTE	Part Time Employment
PUAC	Pandemic Unemployment Assistance Claims
RAM	Retirement Age Men
RAW	Retirement Age Women
RCR	Refinery Crude Runs
REM	Remittances
RFMI	Richmond Fed Manufacturing Index
RFMSI	Richmond Fed Manufacturing Shipments Index
RFSI	Richmond Fed Services Index
RI	Redbook Index
RIEA	Retail Inventories Excluding Autos
RPI	Retail Price Index
RR	Repo Rate
RRR	Reverse Repo Rate
RSEA	Retail Sales Excluding Autos
RSEF	Retail Sales Excluding Fuel
RSMM	Retail Sales, MoM
RSYY	Retail Sales, YoY
RTI	Reuters Tankan Index
SBSI	Small Business Sentiment Index
SFHP	Single Family Home Prices
SP	Steel Production
SPI	Sugar Price Index
SS	Services Sentiment
SSR	Social Security Rate
SSRC	Social Security Rate For Companies
SSRE	Social Security Rate For Employees
STR	Sales Tax Rate
TA	Tourist Arrivals
TAXR	Tax Revenue
TCB	Treasury Cash Balance
TCPI	Tokyo CPI
TI	Terrorism Index
TII	Tertiary Industry Index
TOT	Terms Of Trade
TR	Tourism Revenues
TVS	Total Vehicle Sales

continues on next page

Table 6 – continued from previous page

field	Metric
UC	Unemployment Change
UP	Unemployed Persons
UR	Unemployment Rate
WAG	Wages
WES	Weapons Sales
WG	Wage Growth, YoY
WHS	Wages High Skilled
WI	Wholesale Inventories
WLS	Wages Low Skilled
WM	Wages In Manufacturing
WPI	Wholesale Price Index
WS	Wholesale Sales
YUR	Youth Unemployment Rate
ZCC	ZEW Current Conditions

#### 4.14.13 `request.seed()`

TradingView aggregates a vast amount of data from its many providers, including price and volume information on tradable instruments, financials, economic data, and more, which users can retrieve in Pine Script™ using the functions discussed in the sections above, as well as multiple built-in variables.

To further expand the horizons of possible data one can analyze on TradingView, we have Pine Seeds, which allows users to supply custom *user-maintained* EOD data feeds via GitHub for use on TradingView charts and within Pine Script™ code.

---

**Note:** This section contains only a *brief* overview of Pine Seeds. For in-depth information about Pine Seeds functionality, setting up a repo, data formats, and more, consult the documentation [here](#).

---

To retrieve data from a Pine Seeds data feed within a script, one can use the `request.seed()` function.

Below is the function's signature:

```
request.seed(source, symbol, expression, ignore_invalid_symbol) → series <type>
```

The `source` parameter specifies the unique name of the user-maintained GitHub repository that contains the data feed. For details on creating a repo, see [this page](#).

The `symbol` parameter represents the file name from the “data/” directory of the `source` repository, excluding the “.csv” file extension. See [this page](#) for information about the structure of the data stored in repositories.

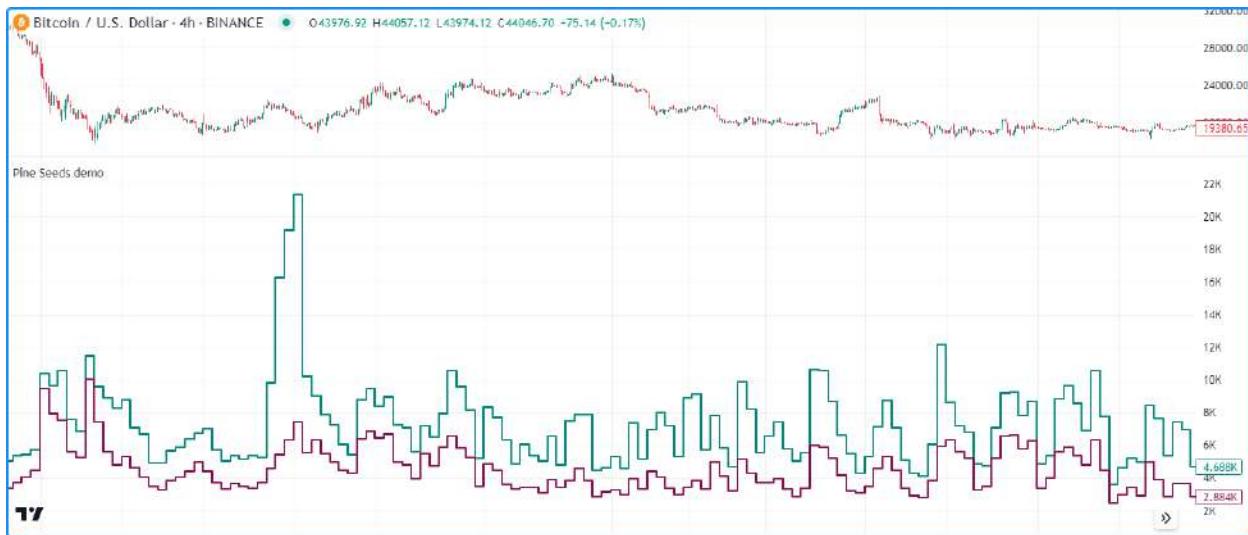
The `expression` parameter is the series to evaluate using data extracted from the requested context. It is similar to the equivalent in `request.security()` and `request.security_lower_tf()`. Data feeds stored in user-maintained repos contain `time`, `open`, `high`, `low`, `close`, and `volume` information, meaning expressions used as the `expression` argument can use the corresponding built-in variables, including variables derived from them (e.g., `bar_index`, `ohlc4`, etc.) to request their values from the context of the custom data.

---

**Note:** As with `request.security()` and `request.security_lower_tf()`, `request.seed()` duplicates the scopes necessary to evaluate its `expression` in another context, which contributes toward compilation limits and script memory demands. See the [Limitations](#) page's section on `scope count` limits for more information.

---

The script below visualizes sample data from the `seed_crypto_santiment` demo repo. It uses two calls to `request.seed()` to retrieve the `close` values from the repo's `BTC_SENTIMENT_POSITIVE_TOTAL` and `BTC_SENTIMENT_NEGATIVE_TOTAL` data feeds and *plots* the results on the chart as step lines:



```

1 // @version=5
2 indicator("Pine Seeds demo", format=format.volume)
3
4 // @variable The total positive sentiment for BTC extracted from the "seed_crypto_
5 // @variable santiment" repository.
6 float positiveTotal = request.seed("seed_crypto_santiment", "BTC_SENTIMENT_POSITIVE_
7 // @variable TOTAL", close)
8 // @variable The total negative sentiment for BTC extracted from the "seed_crypto_
9 // @variable santiment" repository.
10 float negativeTotal = request.seed("seed_crypto_santiment", "BTC_SENTIMENT_NEGATIVE_
11 // Plot the data.
12 plot(positiveTotal, "Positive sentiment", color.teal, 2, plot.style_steline)
13 plot(negativeTotal, "Negative sentiment", color.maroon, 2, plot.style_steline)

```

#### Note that:

- This example requests data from the repo highlighted in the Pine Seeds documentation. It exists solely for example purposes, and its data *does not* update on a regular basis.
- Unlike most other `request.*()` functions, `request.seed()` does not have a `gaps` parameter. It will always return `na` values when no new data exists.
- Pine Seeds data is searchable from the chart's symbol search bar. To load a data feed on the chart, enter the “`Repo:File`” pair, similar to searching for an “`Exchange:Symbol`” pair.

**TradingView**



## 4.15 Plots

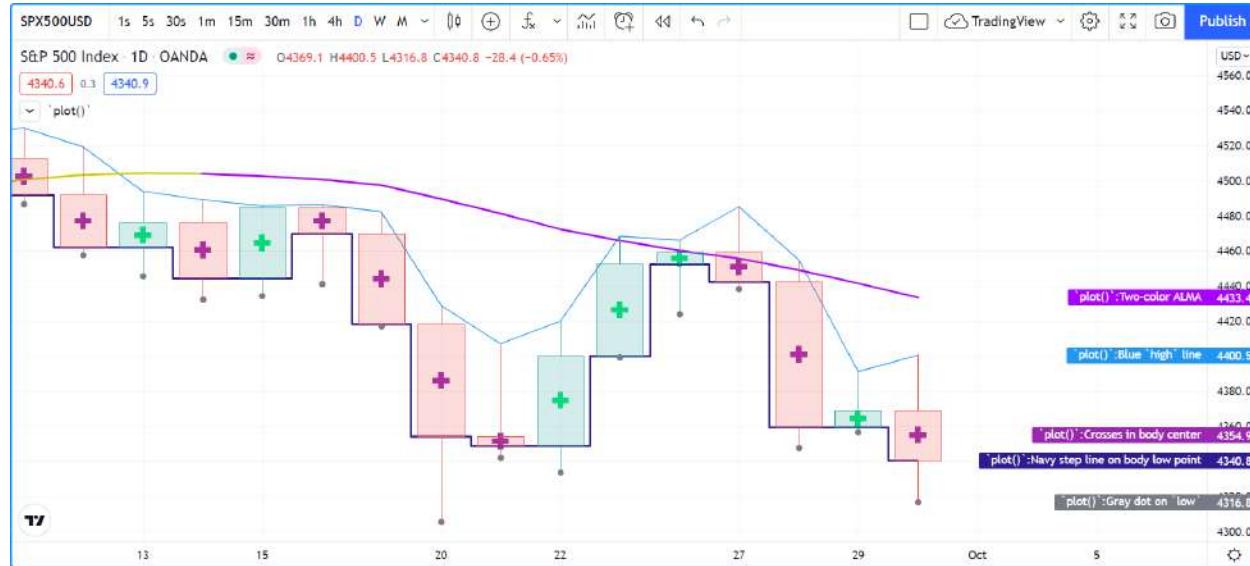
- *Introduction*
- *plot()` parameters*
- *Plotting conditionally*
- *Levels*
- *Offsets*
- *Plot count limit*
- *Scale*

### 4.15.1 Introduction

The `plot()` function is the most frequently used function used to display information calculated using Pine scripts. It is versatile and can plot different styles of lines, histograms, areas, columns (like volume columns), fills, circles or crosses.

The use of `plot()` to create fills is explained in the page on [Fills](#).

This script showcases a few different uses of `plot()` in an overlay script:



```

1 //@version=5
2 indicator(`plot()`, "", true)
3 plot(high, "Blue `high` line")
4 plot(math.avg(close, open), "Crosses in body center", close > open ? color.lime : color.purple, 6, plot.style_cross)
5 plot(math.min(open, close), "Navy step line on body low point", color.navy, 3, plot.style_stepline)
6 plot(low, "Gray dot on `low`", color.gray, 3, plot.style_circles)
7
8 color VIOLET = #AA00FF
9 color GOLD   = #CCCC00
10 ma = ta.alma(hl2, 40, 0.85, 6)
  
```

(continues on next page)

(continued from previous page)

```

11 var almaColor = color.silver
12 almaColor := ma > ma[2] ? GOLD : ma < ma[2] ? VIOLET : almaColor
13 plot(ma, "Two-color ALMA", almaColor, 2)

```

Note that:

- The first `plot()` call plots a 1-pixel blue line across the bar highs.
- The second plots crosses at the mid-point of bodies. The crosses are colored lime when the bar is up and purple when it is down. The argument used for `linewidth` is 6 but it is not a pixel value; just a relative size.
- The third call plots a 3-pixel wide step line following the low point of bodies.
- The fourth call plot a gray circle at the bars' `low`.
- The last plot requires some preparation. We first define our bull/bear colors, calculate an Arnaud Legoux Moving Average, then make our color calculations. We initialize our color variable on bar zero only, using `var`. We initialize it to `color.silver`, so on the dataset's first bars, until one of our conditions causes the color to change, the line will be silver. The conditions that change the color of the line require it to be higher/lower than its value two bars ago. This makes for less noisy color transitions than if we merely looked for a higher/lower value than the previous one.

This script shows other uses of `plot()` in a pane:



```

1 // @version=5
2 indicator("Volume change", format = format.volume)
3
4 color GREEN      = #008000
5 color GREEN_LIGHT = color.new(GREEN, 50)
6 color GREEN_LIGHTER = color.new(GREEN, 85)
7 color PINK       = #FF0080
8 color PINK_LIGHT = color.new(PINK, 50)
9 color PINK_LIGHTER = color.new(PINK, 90)
10
11 bool barUp = ta.rising(close, 1)
12 bool barDn = ta.falling(close, 1)
13 float volumeChange = ta.change(volume)
14
15 volumeColor = barUp ? GREEN_LIGHTER : barDn ? PINK_LIGHTER : color.gray

```

(continues on next page)

(continued from previous page)

```

16 plot(volume, "Volume columns", volumeColor, style = plot.style_columns)
17
18 volumeChangeColor = barUp ? volumeChange > 0 ? GREEN : GREEN_LIGHT : volumeChange > 0 ?
19   ↪? PINK : PINK_LIGHT
20 plot(volumeChange, "Volume change columns", volumeChangeColor, 12, plot.style_
21   ↪histogram)
22
23 plot(0, "Zero line", color.gray)

```

Note that:

- We are plotting normal `volume` values as wide columns above the zero line (see the `style = plot.style_columns` in our `plot()` call).
- Before plotting the columns we calculate our `volumeColor` by using the values of the `barUp` and `barDn` boolean variables. They become respectively `true` when the current bar's `close` is higher/lower than the previous one. Note that the “Volume” built-in does not use the same condition; it identifies an up bar with `close > open`. We use the `GREEN_LIGHTER` and `PINK_LIGHTER` colors for the volume columns.
- Because the first plot plots columns, we do not use the `linewidth` parameter, as it has no effect on columns.
- Our script's second plot is the **change** in volume, which we have calculated earlier using `ta.change(volume)`. This value is plotted as a histogram, for which the `linewidth` parameter controls the width of the column. We make this width 12 so that histogram elements are thinner than the columns of the first plot. Positive/negative `volumeChange` values plot above/below the zero line; no manipulation is required to achieve this effect.
- Before plotting the histogram of `volumeChange` values, we calculate its color value, which can be one of four different colors. We use the bright `GREEN` or `PINK` colors when the bar is up/down AND the volume has increased since the last bar (`volumeChange > 0`). Because `volumeChange` is positive in this case, the histogram's element will be plotted above the zero line. We use the bright `GREEN_LIGHT` or `PINK_LIGHT` colors when the bar is up/down AND the volume has NOT increased since the last bar. Because `volumeChange` is negative in this case, the histogram's element will be plotted below the zero line.
- Finally, we plot a zero line. We could just as well have used `hline(0)` there.
- We use `format = format.volume` in our `indicator()` call so that large values displayed for this script are abbreviated like those of the built-in “Volume” indicator.

`plot()` calls must always be placed in a line's first position, which entails they are always in the script's global scope. They can't be placed in user-defined functions or structures like `if`, `for`, etc. Calls to `plot()` can, however, be designed to plot conditionally in two ways, which we cover in the Conditional plots section of this page.

A script can only plot in its own visual space, whether it is in a pane or on the chart as an overlay. Scripts running in a pane can only *color bars* in the chart area.

## 4.15.2 `plot()` parameters

The `plot()` function has the following signature:

```

plot(series, title, color, linewidth, style, trackprice, histbase, offset, join,
      ↪editable, show_last, display) → plot

```

The parameters of `plot()` are:

### series

It is the only mandatory parameter. Its argument must be of “series int/float” type. Note that because the auto-casting rules in Pine Script™ convert in the int ↴ float ↴ bool direction, a “bool” type variable cannot be used as

is; it must be converted to an “int” or a “float” for use as an argument. For example, if newDay is of “bool” type, then `newDay ? 1 : 0` can be used to plot 1 when the variable is `true`, and zero when it is `false`.

#### **title**

Requires a “const string” argument, so it must be known at compile time. The string appears:

- In the script’s scale when the “Chart settings/Scales/Indicator Name Label” field is checked.
- In the Data Window.
- In the “Settings/Style” tab.
- In the dropdown of `input.source()` fields.
- In the “Condition” field of the “Create Alert” dialog box, when the script is selected.
- As the column header when exporting chart data to a CSV file.

#### **color**

Accepts “series color”, so can be calculated on the fly, bar by bar. Plotting with `na` as the color, or any color with a transparency of 100, is one way to hide plots when they are not needed.

#### **linewidth**

Is the plotted element’s size, but it does not apply to all styles. When a line is plotted, the unit is pixels. It has no impact when `plot.style_columns` is used.

#### **style**

The available arguments are:

- `plot.style_line` (the default): It plots a continuous line using the `linewidth` argument in pixels for its width. `na` values will not plot as a line, but they will be bridged when a value that is not `na` comes in. Non-`na` values are only bridged if they are visible on the chart.
- `plot.style_linebr`: Allows the plotting of discontinuous lines by not plotting on `na` values, and not joining gaps, i.e., bridging over `na` values.
- `plot.style_stpline`: Plots using a staircase effect. Transitions between changes in values are done using a vertical line drawn in middle of bars, as opposed to a point-to-point diagonal joining the midpoints of bars. Can also be used to achieve an effect similar to that of `plot.style_linebr`, but only if care is taken to plot no color on `na` values.
- `plot.style_area`: plots a line of `linewidth` width, filling the area between the line and the `histbase`. The `color` argument is used for both the line and the fill. You can make the line a different color by using another `plot()` call. Positive values are plotted above the `histbase`, negative values below it.
- `plot.style_areabr`: This is similar to `plot.style_area` but it doesn’t bridge over `na` values. Another difference is how the indicator’s scale is calculated. Only the plotted values serve in the calculation of the y range of the script’s visual space. If only high values situated far away from the `histbase` are plotted, for example, those values will be used to calculate the y scale of the script’s visual space. Positive values are plotted above the `histbase`, negative values below it.
- `plot.style_columns`: Plots columns similar to those of the “Volume” built-in indicator. The `linewidth` value does **not** affect the width of the columns. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the y scale of the script’s visual space.
- `plot.style_histogram`: Plots columns similar to those of the “Volume” built-in indicator, except that the `linewidth` value is used to determine the width of the histogram’s bars in pixels. Note that since `linewidth` requires an “input int” value, the width of the histogram’s bars cannot vary bar to bar. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the y scale of the script’s visual space.

- `plot.style_circles` and `plot.style_cross`: These plot a shape that is not joined across bars unless `join = true` is also used. For these styles, the `linewidth` argument becomes a relative sizing measure — its units are not pixels.

### **trackprice**

The default value of this is `false`. When it is `true`, a dotted line made up of small squares will be plotted the full width of the script's visual space. It is often used in conjunction with `show_last = 1, offset = -99999` to hide the actual plot and only leave the residual dotted line.

### **histbase**

It is the reference point used with `plot.style_area`, `plot.style_columns` and `plot.style_histogram`. It determines the level separating positive and negative values of the `series` argument. It cannot be calculated dynamically, as an “input int/float” is required.

### **offset**

This allows shifting the plot in the past/future using a negative/positive offset in bars. The value cannot change during the script's execution.

### **join**

This only affect styles `plot.style_circles` or `plot.style_cross`. When `true`, the shapes are joined by a one-pixel line.

### **editable**

This boolean parameter controls whether or not the plot's properties can be edited in the “Settings/Style” tab. Its default value is `true`.

### **show\_last**

Allows control over how many of the last bars the plotted values are visible. An “input int” argument is required, so it cannot be calculated dynamically.

### **display**

The default is `display.all`. When it is set to `display.none`, plotted values will not affect the scale of the script's visual space. The plot will be invisible and will not appear in indicator values or the Data Window. It can be useful in plots destined for use as external inputs for other scripts, or for plots used with the `{}{plot(" [plot_title] ")}` placeholder in `alertcondition()` calls, e.g.:

```
1 //@version=5
2 indicator("")
3 r = ta.rsi(close, 14)
4 xUp = ta.crossover(r, 50)
5 plot(r, "RSI", display = display.none)
6 alertcondition(xUp, "xUp alert", message = 'RSI is bullish at: {{plot("RSI")}}')
```

### 4.15.3 Plotting conditionally

`plot()` calls cannot be used in conditional structures such as `if`, but they can be controlled by varying their plotted values, or their color. When no plot is required, you can either plot `na` values, or plot values using `na` color or any color with 100 transparency (which also makes it invisible).

#### Value control

One way to control the display of plots is to plot `na` values when no plot is needed. Sometimes, values returned by functions such as `request.security()` will return `na` values, when `gaps = barmerge.gaps_on` is used, for example. In both these cases it is sometimes useful to plot discontinuous lines. This script shows a few ways to do it:



```

1 // @version=5
2 indicator("Discontinuous plots", "", true)
3 bool plotValues = bar_index % 3 == 0
4 plot(plotValues ? high : na, color = color.fuchsia, linewidth = 6, style = plot.style_
    ↴linebr)
5 plot(plotValues ? high : na)
6 plot(plotValues ? math.max(open, close) : na, color = color.navy, linewidth = 6,_
    ↴style = plot.style_cross)
7 plot(plotValues ? math.min(open, close) : na, color = color.navy, linewidth = 6,_
    ↴style = plot.style_circles)
8 plot(plotValues ? low : na, color = plotValues ? color.green : na, linewidth = 6,_
    ↴style = plot.style_stepline)

```

Note that:

- We define the condition determining when we plot using `bar_index % 3 == 0`, which becomes `true` when the remainder of the division of the bar index by 3 is zero. This will happen every three bars.
- In the first plot, we use `plot.style_linebr`, which plots the fuchsia line on highs. It is centered on the bar's horizontal midpoint.
- The second plot shows the result of plotting the same values, but without using special care to break the line. What's happening here is that the thin blue line of the plain `plot()` call is automatically bridged over `na` values (or `gaps`), so the plot does not interrupt.

- We then plot navy blue crosses and circles on the body tops and bottoms. The `plot.style_circles` and `plot.style_cross` style are a simple way to plot discontinuous values, e.g., for stop or take profit levels, or support & resistance levels.
- The last plot in green on the bar lows is done using `plot.style_stepline`. Note how its segments are wider than the fuchsia line segments plotted with `plot.style_linebr`. Also note how on the last bar, it only plots halfway until the next bar comes in.
- The plotting order of each plot is controlled by their order of appearance in the script. See

This script shows how you can restrict plotting to bars after a user-defined date. We use the `input.time()` function to create an input widget allowing script users to select a date and time, using Jan 1st 2021 as its default value:

```

1 //@version=5
2 indicator("", "", true)
3 startInput = input.time(timestamp("2021-01-01"))
4 plot(time > startInput ? close : na)
```

### Color control

The [Conditional coloring](#) section of the page on colors discusses color control for plots. We'll look here at a few examples.

The value of the `color` parameter in `plot()` can be a constant, such as one of the built-in [constant colors](#) or a [color literal](#). In Pine Script™, the qualified type of such colors is called “**const color**” (see the [Type system](#) page). They are known at compile time:

```

1 //@version=5
2 indicator("", "", true)
3 plot(close, color = color.gray)
```

The color of a plot can also be determined using information that is only known when the script begins execution on the first historical bar of a chart (bar zero, i.e., `bar_index == 0` or `barstate.isfirst == true`), as will be the case when the information needed to determine a color depends on the chart the script is running on. Here, we calculate a plot color using the `syminfo.type` built-in variable, which returns the type of the chart's symbol. The qualified type of `plotColor` in this case will be “**simple color**”:

```

1 //@version=5
2 indicator("", "", true)
3 plotColor = switch syminfo.type
4     "stock"      => color.purple
5     "futures"    => color.red
6     "index"      => color.gray
7     "forex"      => color.fuchsia
8     "crypto"     => color.lime
9     "fund"       => color.orange
10    "dr"         => color.aqua
11    "cfd"        => color.blue
12 plot(close, color = plotColor)
13 printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t,
14                                ↳ 0, 0, txt, bgcolor = color.yellow)
14 printTable(syminfo.type)
```

Plot colors can also be chosen through a script's inputs. In this case, the `lineColorInput` variable is of the “**input color**” type:

```

1 //@version=5
2 indicator("", "", true)
```

(continues on next page)

(continued from previous page)

```

3 color lineColorInput = input(#1848CC, "Line color")
4 plot(close, color = lineColorInput)

```

Finally, plot colors can also be *dynamic* values, i.e., calculated values that can change on each bar. These values are of the “**series color**” type:

```

1 //@version=5
2 indicator("", "", true)
3 plotColor = close >= open ? color.lime : color.red
4 plot(close, color = plotColor)

```

When plotting pivot levels, one common requirement is to avoid plotting level transitions. Using [lines](#) is one alternative, but you can also use [plot\(\)](#) like this:



```

1 //@version=5
2 indicator("Pivot plots", "", true)
3 pivotHigh = fixnan(ta.pivothigh(3,3))
4 plot(pivotHigh, "High pivot", ta.change(pivotHigh) ? na : color.olive, 3)
5 plotchar(ta.change(pivotHigh), "ta.change(pivotHigh)", ".", location.top, size = size.
  ↪small)

```

Note that:

- We use `pivotHigh = fixnan(ta.pivothigh(3,3))` to hold our pivot values. Because `ta.pivothigh()` only returns a value when a new pivot is found, we use `fixnan()` to fill the gaps with the last pivot value returned. The gaps here refer to the `na` values `ta.pivothigh()` returns when no new pivot is found.
- Our pivots are detected three bars after they occur because we use the argument 3 for both the `leftbars` and `rightbars` parameters in our `ta.pivothigh()` call.
- The last plot is plotting a continuous value, but it is setting the plot’s color to `na` when the pivot’s value changes, so the plot isn’t visible then. Because of this, a visible plot will only appear on the bar following the one where we plotted using `na` color.
- The blue dot indicates when a new high pivot is detected and no plot is drawn between the preceding bar and that one. Note how the pivot on the bar indicated by the arrow has just been detected in the realtime bar, three bars later, and how no plot is drawn. The plot will only appear on the next bar, making the plot visible **four bars** after the actual pivot.

#### 4.15.4 Levels

Pine Script™ has an `hline()` function to plot horizontal lines (see the page on [Levels](#)). `hline()` is useful because it has some line styles unavailable with `plot()`, but it also has some limitations, namely that it does not accept “series color”, and that its `price` parameter requires an “input int/float”, so cannot vary during the script’s execution.

You can plot levels with `plot()` in a few different ways. This shows a CCI indicator with levels plotted using `plot()`:



```

1 //@version=5
2 indicator("CCI levels with `plot()`")
3 plot(ta.cci(close, 20))
4 plot(0, "Zero", color.gray, 1, plot.style_circles)
5 plot(bar_index % 2 == 0 ? 100 : na, "100", color.lime, 1, plot.style_linebr)
6 plot(bar_index % 2 == 0 ? -100 : na, "-100", color.fuchsia, 1, plot.style_linebr)
7 plot( 200, "200", color.green, 2, trackprice = true, show_last = 1, offset = -99999)
8 plot(-200, "-200", color.red, 2, trackprice = true, show_last = 1, offset = -99999)
9 plot( 300, "300", color.new(color.green, 50), 1)
10 plot(-300, "-300", color.new(color.red, 50), 1)

```

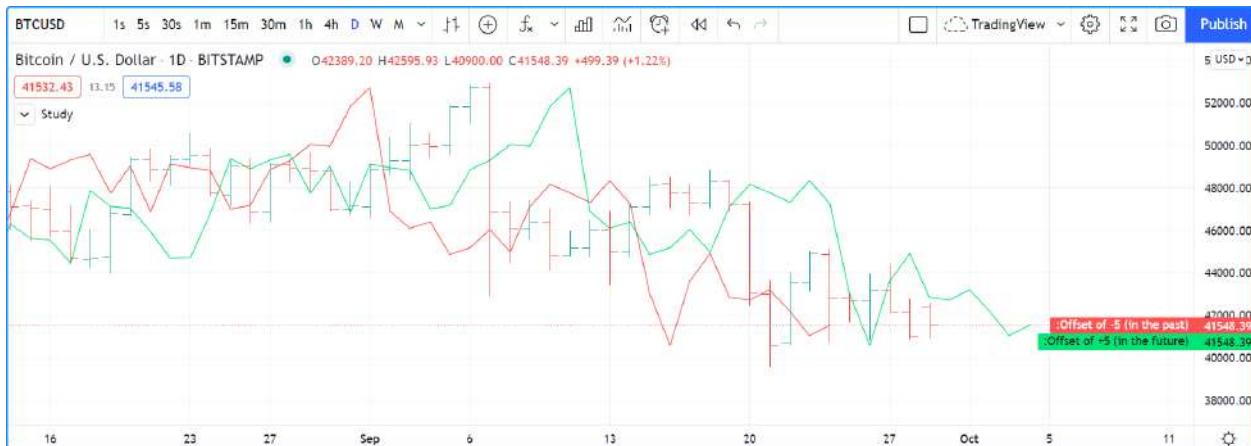
Note that:

- The zero level is plotted using `plot.style_circles`.
- The 100 levels are plotted using a conditional value that only plots every second bar. In order to prevent the `na` values from being bridged, we use the `plot.style_linebr` line style.
- The 200 levels are plotted using `trackprice = true` to plot a distinct pattern of small squares that extends the full width of the script’s visual space. The `show_last = 1` in there displays only the last plotted value, which would appear as a one-bar straight line if the next trick wasn’t also used: the `offset = -99999` pushes that one-bar segment far away in the past so that it is never visible.
- The 300 levels are plotted using a continuous line, but a lighter transparency is used to make them less prominent.

## 4.15.5 Offsets

The `offset` parameter specifies the shift used when the line is plotted (negative values shift in the past, positive values shift into the future). For example:

```
//@version=5
indicator("", "", true)
plot(close, color = color.red, offset = -5)
plot(close, color = color.lime, offset = 5)
```



As can be seen in the screenshot, the *red* series has been shifted to the left (since the argument's value is negative), while the *green* series has been shifted to the right (its value is positive).

## 4.15.6 Plot count limit

Each script is limited to a maximum plot count of 64. All `plot*` () calls and `alertcondition()` calls count in the plot count of a script. Some types of calls count for more than one in the total plot count.

`plot()` calls count for one in the total plot count if they use a “const color” argument for the `color` parameter, which means it is known at compile time, e.g.:

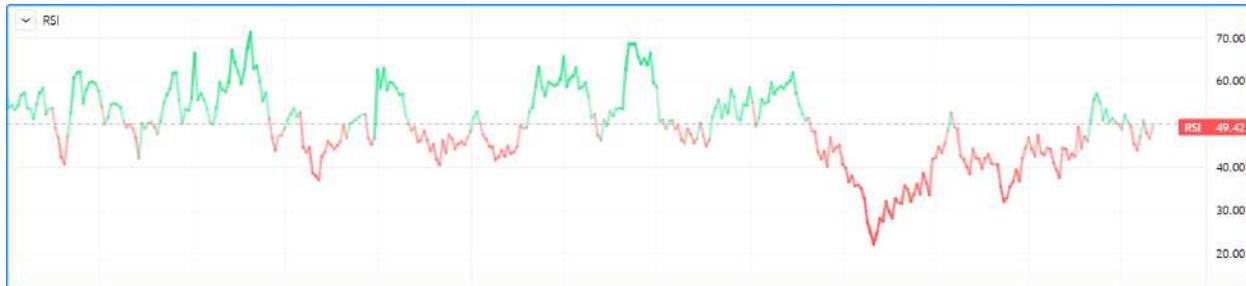
```
plot(close, color = color.green)
```

When they use another qualified type, such as any one of these, they will count for two in the total plot count:

```
plot(close, color = syminfo.mintick > 0.0001 ? color.green : color.red) // "simple
˓→color"
plot(close, color = input.color(color.purple)) // "input color"
plot(close, color = close > open ? color.green : color.red) // "series color"
plot(close, color = color.new(color.silver, close > open ? 40 : 0)) // "series color"
```

## 4.15.7 Scale

Not all values can be plotted everywhere. Your script's visual space is always bound by upper and lower limits that are dynamically adjusted with the values plotted. An `RSI` indicator will plot values between 0 and 100, which is why it is usually displayed in a distinct *pane* — or area — above or below the chart. If RSI values were plotted as an overlay on the chart, the effect would be to distort the symbol's normal price scale, unless it just happened to be close to RSI's 0 to 100 range. This shows an RSI signal line and a centerline at the 50 level, with the script running in a separate pane:



```
1 //@version=5
2 indicator("RSI")
3 myRSI = ta.rsi(close, 20)
4 bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70), color.
5     new(color.lime, 0))
6 bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0), color.
7     new(color.red, 70))
8 myRSIColor = myRSI > 50 ? bullColor : bearColor
9 plot(myRSI, "RSI", myRSIColor, 3)
10 hline(50)
```

Note that the y axis of our script's visual space is automatically sized using the range of values plotted, i.e., the values of RSI. See the page on [Colors](#) for more information on the `color.from_gradient()` function used in the script.

If we try to plot the symbol's `close` values in the same space by adding the following line to our script:

```
plot(close)
```

This is what happens:



The chart is on the BTCUSD symbol, whose `close` prices are around 40000 during this period. Plotting values in the 40000 range makes our RSI plots in the 0 to 100 range indiscernible. The same distorted plots would occur if we placed the `RSI` indicator on the chart as an overlay.

## Merging two indicators

If you are planning to merge two signals in one script, first consider the scale of each. It is impossible, for example, to correctly plot an **RSI** and a **MACD** in the same script's visual space because RSI has a fixed range (0 to 100) while MACD doesn't, as it plots moving averages calculated on price.\_

If both your indicators used fixed ranges, you can shift the values of one of them so they do not overlap. We could, for example, plot both **RSI** (0 to 100) and the **True Strength Indicator (TSI)** (-100 to +100) by displacing one of them. Our strategy here will be to compress and shift the TSI values so they plot over **RSI**:



```

1 // @version=5
2 indicator("RSI and TSI")
3 myRSI = ta.rsi(close, 20)
4 bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70), color.
5   new(color.lime, 0))
6 bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0), color.
7   new(color.red, 70))
8 myRSIColor = myRSI > 50 ? bullColor : bearColor
9 plot(myRSI, "RSI", myRSIColor, 3)
10 hline(100)
11 hline(50)
12 hline(0)
13
14 // 1. Compress TSI's range from -100/100 to -50/50.
15 // 2. Shift it higher by 150, so its -50 min value becomes 100.
16 myTSI = 150 + (100 * ta.tsiclose, 13, 25) / 2
17 plot(myTSI, "TSI", color.blue, 2)
18 plot(ta.ema(myTSI, 13), "TSI EMA", #FF006E)
19 hline(200)
20 hline(150)

```

Note that:

- We have added levels using **hline** to situate both signals.
- In order for both signal lines to oscillate on the same range of 100, we divide the **TSI** value by 2 because it has a 200 range (-100 to +100). We then shift this value up by 150 so it oscillates between 100 and 200, making 150 its centerline.
- The manipulations we make here are typical of the compromises required to bring two indicators with different scales in the same visual space, even when their values, contrary to **MACD**, are bounded in a fixed range.





## 4.16 Repainting

- *Introduction*
- *Historical vs realtime calculations*
- *Plotting in the past*
- *Dataset variations*

### 4.16.1 Introduction

We define repainting as: **script behavior causing historical vs realtime calculations or plots to behave differently**.

Repainting behavior is widespread and many factors can cause it. Following our definition, our estimate is that more than 95% of indicators in existence exhibit some form of repainting behavior. Commonly used indicators such as MACD and RSI, for example, show confirmed values on historical bars, but will fluctuate on a realtime, unconfirmed chart bar until it closes. Therefore, they behave *differently* in historical and realtime states.

**Not all repainting behavior is inherently useless or misleading**, nor does such behavior prevent knowledgeable traders from using indicators with such behavior. For example, who would think of discrediting a volume profile indicator solely because it updates its values on realtime bars?

One may encounter any of the following forms of repainting in the scripts they use, depending on what a script's calculations entail:

- **Widespread but often acceptable:** A script may use values that update with realtime price changes on the unconfirmed bar. For example, if one uses the `close` variable in calculations performed on an open chart bar, its values will reflect the most recent price in the bar. However, the script will only commit a new data point to its historical series once the bar closes. Another common case is using `request.security()` to fetch higher-timeframe data on realtime bars, as explained in the *Historical and realtime behavior* section of the *Other timeframes and data* page. As with the unconfirmed chart bar in the chart's timeframe, `request.security()` can track unconfirmed values from a higher-timeframe context on realtime bars, which can lead to repainting after the script restarts its execution. There is often nothing wrong with using such scripts, provided you understand how they work. When electing to use such scripts to issue alerts or trade orders, however, it's important to understand the difference between their realtime and historical behavior and decide for yourself whether it provides utility for your needs.
- **Potentially misleading:** Scripts that plot values into the past, calculate results on realtime bars that one cannot replicate on historical bars, or relocate past events are potentially misleading. For example, Ichimoku, most scripts based on pivots, most strategies using `calc_on_every_tick = true`, scripts using `request.security()` when it behaves differently on realtime bars, many scripts using `varip`, many scripts using `timenow`, and some scripts that use `barstate.*` variables can exhibit misleading repainting behavior.
- **Unacceptable:** Scripts that leak future information into the past, strategies that execute on *non-standard charts*, and scripts using realtime intrabars to generate alerts or orders, are examples that can produce heavily misleading repainting behavior.

- **Unavoidable:** Revisions of the data feed from a provider and variations in the starting bar of the chart's history can cause repainting behavior that may be unavoidable in a script.

The first two types of repainting can be perfectly acceptable if:

1. You are aware of the behavior.
2. You can live with it, or
3. You can circumvent it.

It should now be clear that not **all** repainting behavior is wrong and requires avoiding at all costs. In many situations, some forms of repainting may be exactly what a script needs. What's important is to know when repainting behavior is **not** acceptable for one's needs. To avoid repainting that's not acceptable, it's important to understand how a tool works or how you should design the tools you build. If you [publish](#) scripts, ensure you mention any potentially misleading behavior along with the other limitations of your script in the publication's description.

---

**Note:** We will not discuss the perils of using strategies on non-standard charts, as this problem is not related to repainting. See the [Backtesting on Non-Standard Charts: Caution!](#) script for a discussion of the subject.

---

## For script users

One can decide to use repainting indicators if they understand the behavior, and whether that behavior meets their analysis requirements. Don't be one of those newcomers who slap "repaint" sentences on published scripts in an attempt to discredit them, as doing so reveals a lack of foundational knowledge on the subject.

Simply asking whether a script repaints is relatively meaningless, given that there are forms of repainting behavior that are perfectly acceptable in a script. Therefore, such a question will not beget a meaningful answer. One should instead ask *specific* questions about a script's potential repainting behavior, such as:

- Does the script calculate/display in the same way on historical and realtime bars?
- Do alerts from the script wait for the end of a realtime bar before triggering?
- Do signal markers shown by the script wait for the end of a realtime bar before showing?
- Does the script plot/draw values into the past?
- Does the strategy use `calc_on_every_tick = true`?
- Do the script's `request.security()` calls leak future information into the past on historical bars?

What's important is that you understand how the tools you use work, and whether their behavior is compatible with your objectives, repainting or not. As you will learn if you read this page, repainting is a complex matter. It has many faces and many causes. Even if you don't program in Pine Script™, this page will help you understand the array of causes that can lead to repainting, and hopefully enable more meaningful discussions with script authors.

## For Pine Script™ programmers

As discussed above, not all forms of repainting behavior must be avoided at all costs, nor is all potential repainting behavior necessarily avoidable. We hope this page helps you better understand the dynamics at play so that you can design your trading tools with these behaviors in mind. This page's content should help make you aware of common coding mistakes that produce misleading repainting results.

Whatever your design decisions are, if you [publish](#) your script, explain the script to traders so they can understand how it behaves.

This page covers three broad categories of repainting causes:

- *Historical vs realtime calculations*
- *Plotting in the past*
- *Dataset variations*

## 4.16.2 Historical vs realtime calculations

### Fluid data values

Historical data does not include records of intermediary price movements on bars; only `open`, `high`, `low` and `close` values (OHLC).

On realtime bars (bars running when the instrument's market is open), however, the `high`, `low` and `close` values are not fixed; they can change values many times before the realtime bar closes and its HLC values are fixed. They are *fluid*. This leads to a script sometimes working differently on historical data and in real time, where only the `open` price will not change during the bar.

Any script using values like `high`, `low` and `close` in realtime is subject to producing calculations that may not be repeatable on historical bars — thus repaint.

Let's look at this simple script. It detects crosses of the `close` value (in the realtime bar, this corresponds to the current price of the instrument) over and under an `EMA`:



```

1 //@version=5
2 indicator("Repainting", "", true)
3 ma = ta.ema(close, 5)
4 xUp = ta.crossover(close, ma)
5 xDn = ta.crossunder(close, ma)
6 plot(ma, "MA", color.black, 2)
7 bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

```

#### Note that:

- The script uses `bgcolor()` to color the background green when `close` crosses over the EMA, and red on crosses under the EMA.
- The screen snapshot shows the script in realtime on a 30sec chart. A cross over the EMA has been detected, thus the background of the realtime bar is green.

- The problem here is that nothing guarantees this condition will hold true until the end of the realtime bar. The arrow points to the timer showing that 21 seconds remain in the realtime bar, and anything could happen until then.
- We are witnessing a repainting script.

To prevent this repainting, we must rewrite our script so that it does not use values that fluctuate during the realtime bar. This will require using values from a bar that has elapsed (typically the preceding bar), or the `open` price, which does not vary in realtime.

We can achieve this in many ways. This method adds a `and barstate.isconfirmed` condition to our cross detections, which requires the script to be executing on the bar's last iteration, when it closes and prices are confirmed. It is a simple way to avoid repainting:

```

1 //@version=5
2 indicator("Repainting", "", true)
3 ma = ta.ema(close, 5)
4 xUp = ta.crossover(close, ma) and barstate.isconfirmed
5 xDn = ta.crossunder(close, ma) and barstate.isconfirmed
6 plot(ma, "MA", color.black, 2)
7 bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

This uses the crosses detected on the previous bar:

```

1 //@version=5
2 indicator("Repainting", "", true)
3 ma = ta.ema(close, 5)
4 xUp = ta.crossover(close, ma)[1]
5 xDn = ta.crossunder(close, ma)[1]
6 plot(ma, "MA", color.black, 2)
7 bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

This uses only confirmed `close` and EMA values for its calculations:

```

1 //@version=5
2 indicator("Repainting", "", true)
3 ma = ta.ema(close[1], 5)
4 xUp = ta.crossover(close[1], ma)
5 xDn = ta.crossunder(close[1], ma)
6 plot(ma, "MA", color.black, 2)
7 bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

This detects crosses between the realtime bar's `open` and the value of the EMA from the previous bars. Notice that the EMA is calculated using `close`, so it repaints. We must ensure we use a confirmed value to detect crosses, thus `ma[1]` in the cross detection logic:

```

1 //@version=5
2 indicator("Repainting", "", true)
3 ma = ta.ema(close, 5)
4 xUp = ta.crossover(open, ma[1])
5 xDn = ta.crossunder(open, ma[1])
6 plot(ma, "MA", color.black, 2)
7 bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

**All these methods have one thing in common: while they prevent repainting, they will also trigger signals later than repainting scripts. This is an inevitable compromise if one wants to avoid repainting. You can't have your cake and eat it too.**

## Repainting `request.security()` calls

The `request.security()` function behaves differently on historical and realtime bars. On historical bars, it only returns *confirmed* values from its requested context, whereas it can return *unconfirmed* values on realtime bars. When the script restarts its execution, the bars that had a realtime state become historical bars, and will therefore only contain the values it confirmed on those bars. If the values returned by `request.security()` fluctuate on realtime bars without confirmation from the context, the script will repaint them when it restarts its execution. See the *Historical and realtime behavior* section of the [Other timeframes and data](#) page for a detailed explanation.

One can ensure higher-timeframe data requests only return confirmed values on all bars, regardless of bar state, by offsetting the `expression` argument by at least one bar with the history-referencing operator `[]` and using `barmerge.lookahead_on` for the `lookahead` argument in the `request.security()` call, as explained [here](#).

The script below demonstrates the difference between repainting and non-repainting HTF data requests. It contains two `request.security()` calls. The first function call requests `close` data from the `higherTimeframe` without additional specification, and the second call requests the same series with an offset and `barmerge.lookahead_on`.

As we see on all realtime bars (the ones with an orange background), the `repaintingClose` contains values that fluctuate without confirmation from the `higherTimeframe`, meaning it will *repaint* when the script restarts its execution. The `nonRepaintingClose`, on the other hand, behaves the same on realtime and historical bars, i.e., it only changes its value when new, confirmed data is available:



```

1 // @version=5
2 indicator("Repainting vs non-repainting `request.security()` demo", overlay = true)
3
4 // @variable The timeframe to request data from.
5 string higherTimeframe = input.timeframe("30", "Timeframe")
6
7 if timeframe.in_seconds() > timeframe.in_seconds(higherTimeframe)
8     runtime.error("The 'Timeframe' input is smaller than the chart's timeframe. Choose a higher timeframe.")
9
10 // @variable The current `close` requested from the `higherTimeframe`. Fluctuates without confirmation on realtime bars.
11 float repaintingClose = request.security(syminfo.tickerid, higherTimeframe, close)
12 // @variable The last confirmed `close` requested from the `higherTimeframe`.
13 // Behaves the same on historical and realtime bars.
14 float nonRepaintingClose = request.security(
15     syminfo.tickerid, higherTimeframe, close[1], lookahead = barmerge.lookahead_on

```

(continues on next page)

(continued from previous page)

```

16 )
17
18 // Plot the values.
19 plot(repaintingClose, "Repainting close", color.new(color.purple, 50), 8)
20 plot(nonRepaintingClose, "Non-repainting close", color.teal, 3)
21 // Plot a shape when a new `higherTimeframe` starts.
22 plotshape(timeframe.change(higherTimeframe), "Timeframe change marker", shape.square, ←
    ↪location.top, size = size.small)
23 // Color the background on realtime bars.
24 bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar←
    ↪highlight")

```

**Note that:**

- We used the `plotshape()` function to mark the chart when there's a `change` on the `higherTimeframe`.
- This script produces a `runtime error` if the `higherTimeframe` is lower than the chart's timeframe.
- On historical bars, the `repaintingClose` has a new value at the *end* of each timeframe, and the `non-RepaintingClose` has a new value at the *start* of each timeframe.

For the sake of easy reusability, below is a simple a `noRepaintSecurity()` function that one can apply in their scripts to request non-repainting higher-timeframe values:

```

1 // @function Requests non-repainting `expression` values from the context of the←
    ↪`symbol` and `timeframe`.
2 noRepaintSecurity(symbol, timeframe, expression) =>
3     request.security(symbol, timeframe, expression[1], lookahead = barmerge.lookahead_←
    ↪on)

```

**Note that:**

- The `[1]` offset to the series and the use of `lookahead = barmerge.lookahead_on` are interdependent. One **cannot** be removed without compromising the integrity of the function.
- Unlike a plain `request.security()` call, this wrapper function cannot accept tuple `expression` arguments. For multi-element use cases, one can pass a *user-defined type* whose fields contain the desired elements to request.

**Using `request.security()` at lower timeframes**

Some scripts use `request.security()` to request data from a timeframe **lower** than the chart's timeframe. This can be useful when functions specifically designed to handle intrabars at lower timeframes are sent down the timeframe. When this type of user-defined function requires the detection of the intrabars' first bar, as most do, the technique will only work on historical bars. This is due to the fact that realtime intrabars are not yet sorted. The impact of this is that such scripts cannot reproduce in real time their behavior on historical bars. Any logic generating alerts, for example, will be flawed, and constant refreshing will be required to recalculate elapsed realtime bars as historical bars.

When used at lower timeframes than the chart's without specialized functions able to distinguish between intrabars, `request.security()` will only return the value of the **last** intrabar in the dilation of the chart's bar, which is usually not useful, and will also not reproduce in real time, so lead to repainting.

For all these reasons, unless you understand the subtleties of using `request.security()` at lower timeframes than the chart's, it is best to avoid using the function at those timeframes. Higher-quality scripts will have logic to detect such anomalies and prevent the display of results which would be invalid when a lower timeframe is used.

For more reliable lower-timeframe data requests, use `request.security_lower_tf()`, as explained in *this* section of the *Other timeframes and data* page.

### Future leak with `request.security()`

When `request.security()` is used with `lookahead = barmerge.lookahead_on` to fetch prices without offsetting the series by [1], it will return data from the future on historical bars, which is dangerously misleading.

While historical bars will magically display future prices before they should be known, no lookahead is possible in realtime because the future there is unknown, as it should, so no future bars exist.

This is an example:



```

1 // FUTURE LEAK! DO NOT USE!
2 //@version=5
3 indicator("Future leak", "", true)
4 futureHigh = request.security(syminfo.tickerid, "1D", high, lookahead = barmerge.
5   ↪lookahead_on)
6 plot(futureHigh)

```

Note how the higher timeframe line is showing the timeframe's `high` value before it occurs. The solution to avoid this effect is to use the function as demonstrated in [this section](#).

Using lookahead to produce misleading results is not allowed in script publications, as explained in the [lookahead](#) section of the [Other timeframes and data](#) page. Script publications that use this misleading technique **will be moderated**.

### `varip`

Scripts using the `varip` declaration mode for variables (see our section on `varip` for more information) save information across realtime updates, which cannot be reproduced on historical bars where only OHLC information is available. Such scripts may be useful in realtime, including to generate alerts, but their logic cannot be backtested, nor can their plots on historical bars reflect calculations that will be done in realtime.

## Bar state built-ins

Scripts using `bar states` may or may not repaint. As we have seen in the previous section, using `barstate.isconfirmed` is actually one way to **avoid** repainting that **will** reproduce on historical bars, which are always “confirmed”. Uses of other bar states such as `barstate.isnew`, however, will lead to repainting. The reason is that on historical bars, `barstate.isnew` is `true` on the bar’s `close`, yet in realtime, it is `true` on the bar’s `open`. Using the other bar state variables will usually cause some type of behavioral discrepancy between historical and realtime bars.

### `'timenow'`

The `timenow` built-in returns the current time. Scripts using this variable cannot show consistent historical and realtime behavior, so they necessarily repaint.

## Strategies

Strategies using `calc_on_every_tick = true` execute on each realtime update, while strategies run on the `close` of historical bars. They will most probably not generate the same order executions, and so repaint. Note that when this happens, it also invalidates backtesting results, as they are not representative of the strategy’s behavior in realtime.

### 4.16.3 Plotting in the past

Scripts detecting pivots after 5 bars have elapsed will often go back in the past to plot pivot levels or values on the actual pivot, 5 bars in the past. This will often cause unsuspecting traders looking at plots on historical bars to infer that when the pivot happens in realtime, the same plots will appear on the pivot when it occurs, as opposed to when it is detected.

Let’s look at a script showing the price of high pivots by placing the price in the past, 5 bars after the pivot was detected:

```

1 //@version=5
2 indicator("Plotting in the past", "", true)
3 pH1 = ta.pivothigh(5, 5)
4 if not na(pH1)
5   label.new(bar_index[5], na, str.tostring(pH1, format.mintick) + "\n↑", yloc =
6     yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.
7     normal)
```



Note that:

- This script repaints because an elapsed realtime bar showing no price may get a price placed on it if it is identified as a pivot, 5 bars after the actual pivot occurs.
- The display looks great, but it can be misleading.

The best solution to this problem when developing script for others is to plot **without** an offset by default, but give the option for script users to turn on plotting in the past through inputs, so they are necessarily aware of what the script is doing, e.g.:

```

1 //@version=5
2 indicator("Plotting in the past", "", true)
3 plotInThePast = input(false, "Plot in the past")
4 pHi = ta.pivothigh(5, 5)
5 if not na(pHi)
6   label.new(bar_index[plotInThePast ? 5 : 0], na, str.tostring(pHi, format.mintick)_
→+ "\n\"", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black,_
→size = size.normal)

```

### 4.16.4 Dataset variations

#### Starting points

Scripts begin executing on the chart's first historical bar, and then execute on each bar sequentially, as is explained in this manual's page on Pine Script™'s [execution model](#). If the first bar changes, then the script will often not calculate the same way it did when the dataset began at a different point in time.

The following factors have an impact on the quantity of bars you see on your charts, and their *starting point*:

- The type of account you hold
- The historical data available from the data supplier
- The alignment requirements of the dataset, which determine its *starting point*

These are the account-specific bar limits:

- 40,000 bars for the Ultimate plan.
- 30,000 bars for the Elite plan.
- 25,000 bars for the Expert plan.
- 20,000 bars for the Premium plan.
- 10,000 bars for Essential and Plus plans.
- 5000 bars for the Basic plan.

Starting points are determined using the following rules, which depend on the chart's timeframe:

- **1, 5, 10, 15, 30 seconds**: aligns to the beginning of a day.
- **1 - 14 minutes**: aligns to the beginning of a week.
- **15 - 29 minutes**: aligns to the beginning of a month.
- **30 - 1439 minutes**: aligns to the beginning of a year.
- **1440 minutes and higher**: aligns to the first available historical data point.

As time goes by, these factors cause your chart's history to start at different points in time. This often has an impact on your scripts calculations, because changes in calculation results in early bars can ripple through all the other bars in the

dataset. Using functions like `ta.valuewhen()`, `ta.barssince()` or `ta.ema()`, for example, will yield results that vary with early history.

### Revision of historical data

Historical and realtime bars are built using two different data feeds supplied by exchanges/brokers: historical data, and realtime data. When realtime bars elapse, exchanges/brokers sometimes make what are usually small adjustments to bar prices, which are then written to their historical data. When the chart is refreshed or the script is re-executed on those elapsed realtime bars, they will then be built and calculated using the historical data, which will contain those usually small price revisions, if any have been made.

Historical data may also be revised for other reasons, e.g., for stock splits.



## 4.17 Sessions

- *Introduction*
- *Session strings*
- *Session states*
- *Using sessions with `request.security()`*

### 4.17.1 Introduction

Session information is usable in three different ways in Pine Script™:

1. **Session strings** containing from-to start times and day information that can be used in functions such as `time()` and `time_close()` to detect when bars are in a particular time period, with the option of limiting valid sessions to specific days. The `input.session()` function provides a way to allow script users to define session values through a script's "Inputs" tab (see the *Session input* section for more information).
2. **Session states** built-in variables such as `session.ismarket` can identify which session a bar belongs to.
3. When fetching data with `request.security()` you can also choose to return data from *regular* sessions only or *extended* sessions. In this case, the definition of **regular and extended sessions** is that of the exchange. It is part of the instrument's properties — not user-defined, as in point #1. This notion of *regular* and *extended* sessions is the same one used in the chart's interface, in the "Chart Settings/Symbol/Session" field, for example.

The following sections cover both methods of using session information in Pine Script™.

Note that:

- Not all user accounts on TradingView have access to extended session information.

- There is no special “session” type in Pine Script™. Instead, session strings are of “string” type but must conform to the session string syntax.

## 4.17.2 Session strings

### Session string specifications

Session strings used with `time()` and `time_close()` must have a specific format. Their syntax is:

```
<time_period>:<days>
```

Where:

- `<time_period>` uses times in “hhmm” format, with “hh” in 24-hour format, so 1700 for 5PM. The time periods are in the “hhmm-hhmm” format, and a comma can separate multiple time periods to specify combinations of discrete periods.

**For example, - <days> is a set of digits from 1 to 7 that specifies on which days the session is valid.**

1 is Sunday, 7 is Saturday.

---

**Note:** The default days are: 1234567, which is different in Pine Script™ v5 than in earlier versions where 23456 (weekdays) is used. For v5 code to reproduce the behavior from previous versions, it should explicitly mention weekdays, as in "0930-1700:23456".

---

These are examples of session strings:

**"24x7"**

A 7-day, 24-hour session beginning at midnight.

**"0000-0000:1234567"**

Equivalent to the previous example.

**"0000-0000"**

Equivalent to the previous two examples because the default days are 1234567.

**"0000-0000:23456"**

The same as the previous example, but only Monday to Friday.

**"2000-1630:1234567"**

An overnight session that begins at 20:00 and ends at 16:30 the next day. It is valid on all days of the week.

**"0930-1700:146"**

A session that begins at 9:30 and ends at 17:00 on Sundays (1), Wednesdays (4), and Fridays (6).

**"1700-1700:23456"**

An *overnight session*. The Monday session starts Sunday at 17:00 and ends Monday at 17:00. It is valid Monday through Friday.

**"1000-1001:26"**

A weird session that lasts only one minute on Mondays (2) and Fridays (6).

**"0900-1600,1700-2000"**

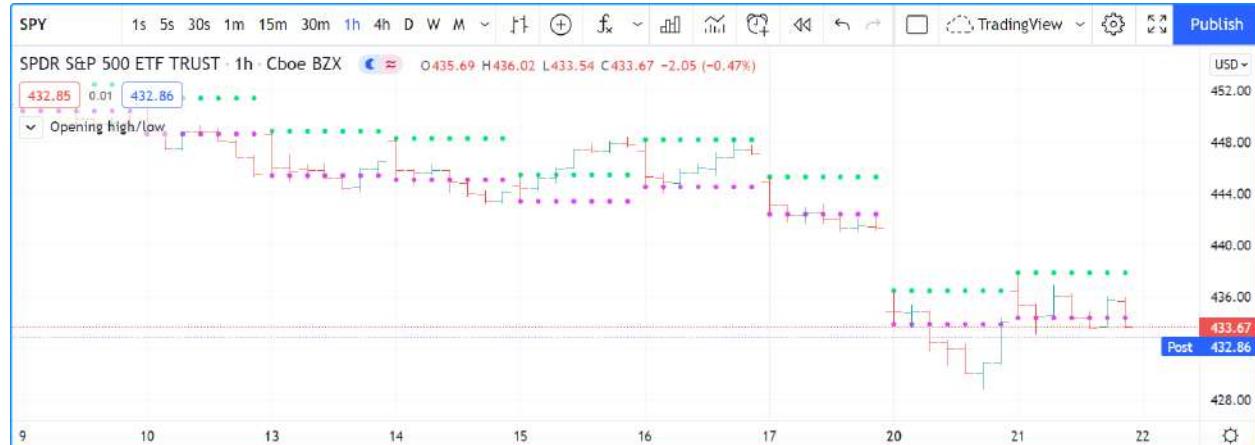
A session that begins at 9:00, breaks from 16:00 to 17:00, and continues until 20:00. Applies to every day of the week.

## Using session strings

Session properties defined with session strings are independent of the exchange-defined sessions determining when an instrument can be traded. Programmers have complete liberty in creating whatever session definitions suit their purpose, which is usually to detect when bars belong to specific time periods. This is accomplished in Pine Script™ by using one of the following two signatures of the `time()` function:

```
time(timeframe, session, timezone) → series int
time(timeframe, session) → series int
```

Here, we use `time()` with a `session` argument to display the market's opening `high` and `low` values on an intraday chart:



```
1 //@version=5
2 indicator("Opening high/low", overlay = true)
3
4 sessionInput = input.session("0930-0935")
5
6 sessionBegins(sess) =>
7     t = time("", sess)
8     timeframe.isintraday and (not barstate.isfirst) and na(t[1]) and not na(t)
9
10 var float hi = na
11 var float lo = na
12 if sessionBegins(sessionInput)
13     hi := high
14     lo := low
15
16 plot(lo, "lo", color.fuchsia, 2, plot.style_circles)
17 plot(hi, "hi", color.lime, 2, plot.style_circles)
```

Note that:

- We use a session input to allow users to specify the time they want to detect. We are only looking for the session's beginning time on bars, so we use a five-minute gap between the beginning and end time of our "0930-0935" default value.
- We create a `sessionBegins()` function to detect the beginning of a session. Its `time("", sess)` call uses an empty string for the function's `timeframe` parameter, which means it uses the chart's `timeframe`, whatever that is. The function returns `true` when:
  - The chart uses an intraday timeframe (seconds or minutes).
  - The script isn't on the chart's first bar, which we ensure with `(not barstate.isfirst)`. This check

prevents the code from always detecting a session beginning on the first bar because `na(t[1])` and `not na(t)` is always `true` there.

- The `time()` call has returned `na` on the previous bar because it wasn't in the session's time period, and it has returned a value that is not `na` on the current bar, which means the bar is **in** the session's time period.

### 4.17.3 Session states

Three built-in variables allow you to distinguish the type of session the current bar belongs to. They are only helpful on intraday timeframes:

- `session.ismarket` returns `true` when the bar belongs to regular trading hours.
- `session.ispremarket` returns `true` when the bar belongs to the extended session preceding regular trading hours.
- `session.ispostmarket` returns `true` when the bar belongs to the extended session following regular trading hours.

### 4.17.4 Using sessions with `request.security()`

When your TradingView account provides access to extended sessions, you can choose to see their bars with the “Settings/Symbol/Session” field. There are two types of sessions:

- **regular** (which does not include pre- and post-market data), and
- **extended** (which includes pre- and post-market data).

Scripts using the `request.security()` function to access data can return extended session data or not. This is an example where only regular session data is fetched:



```

1 //@version=5
2 indicator("Example 1: Regular Session Data")
3 regularSessionData = request.security("NASDAQ:AAPL", timeframe.period, close,
4                                     barmerge.gaps_on)
5 plot(regularSessionData, style = plot.style_linebr)

```

If you want the `request.security()` call to return extended session data, you must first use the `ticker.new()` function to build the first argument of the `request.security()` call:



```

1 // @version=5
2 indicator("Example 2: Extended Session Data")
3 t = ticker.new("NASDAQ", "AAPL", session.extended)
4 extendedSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on)
5 plot(extendedSessionData, style = plot.style_linebr)

```

Note that the previous chart's gaps in the script's plot are now filled. Also, keep in mind that our example scripts do not produce the background coloring on the chart; it is due to the chart's settings showing extended hours.

The `ticker.new()` function has the following signature:

```
ticker.new(prefix, ticker, session, adjustment) → simple string
```

Where:

- `prefix` is the exchange prefix, e.g., "NASDAQ"
- `ticker` is a symbol name, e.g., "AAPL"
- `session` can be `session.extended` or `session.regular`. Note that this is **not** a session string.
- `adjustment` adjusts prices using different criteria: `adjustment.none`, `adjustment.splits`, `adjustment.dividends`.

Our first example could be rewritten as:

```

1 // @version=5
2 indicator("Example 1: Regular Session Data")
3 t = ticker.new("NASDAQ", "AAPL", session.regular)
4 regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on)
5 plot(regularSessionData, style = plot.style_linebr)

```

If you want to use the same session specifications used for the chart's main symbol, omit the third argument in `ticker.new()`; it is optional. If you want your code to declare your intention explicitly, use the `syminfo.session` built-in variable. It holds the session type of the chart's main symbol:

```

1 // @version=5
2 indicator("Example 1: Regular Session Data")
3 t = ticker.new("NASDAQ", "AAPL", syminfo.session)
4 regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on)
5 plot(regularSessionData, style = plot.style_linebr)

```





## 4.18 Strategies

- *Introduction*
- *A simple strategy example*
- *Applying a strategy to a chart*
- *Strategy tester*
- *Broker emulator*
- *Orders and entries*
- *Position sizing*
- *Closing a market position*
- *OCA groups*
- *Currency*
- *Altering calculation behavior*
- *Simulating trading costs*
- *Risk management*
- *Margin*
- *Strategy Alerts*
- *Notes on testing strategies*

### 4.18.1 Introduction

Pine Script™ strategies simulate the execution of trades on historical and real-time data to facilitate the backtesting and forward testing of trading systems. They include many of the same capabilities as Pine Script™ indicators while providing the ability to place, modify, and cancel hypothetical orders and analyze the results.

When a script uses the `strategy()` function for its declaration, it gains access to the `strategy.*` namespace, where it can call functions and variables for simulating orders and accessing essential strategy information. Additionally, the script will display information and simulated results externally in the “Strategy Tester” tab.

## 4.18.2 A simple strategy example

The following script is a simple strategy that simulates the entry of long or short positions upon the crossing of two moving averages:

```

1 // @version=5
2 strategy("test", overlay = true)
3
4 // Calculate two moving averages with different lengths.
5 float fastMA = ta.sma(close, 14)
6 float slowMA = ta.sma(close, 28)
7
8 // Enter a long position when `fastMA` crosses over `slowMA`.
9 if ta.crossover(fastMA, slowMA)
10    strategy.entry("buy", strategy.long)
11
12 // Enter a short position when `fastMA` crosses under `slowMA`.
13 if ta.crossunder(fastMA, slowMA)
14    strategy.entry("sell", strategy.short)
15
16 // Plot the moving averages.
17 plot(fastMA, "Fast MA", color.aqua)
18 plot(slowMA, "Slow MA", color.orange)

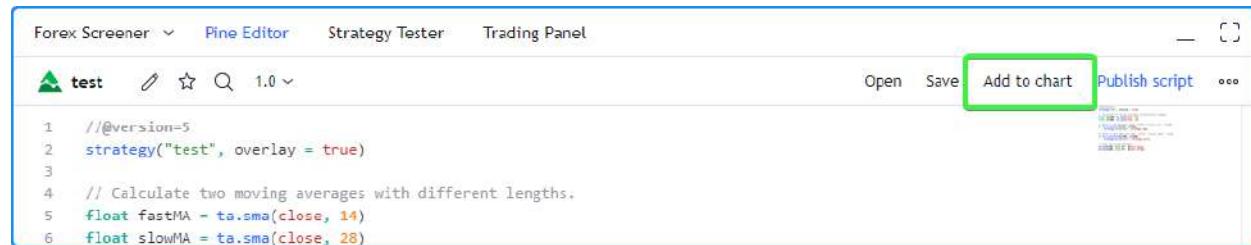
```

### Note that:

- The `strategy("test", overlay = true)` line declares that the script is a strategy named “test” with visual outputs overlaid on the main chart pane.
- `strategy.entry()` is the command that the script uses to simulate “buy” and “sell” orders. When the script places an order, it also plots the order id on the chart and an arrow to indicate the direction.
- Two `plot()` functions plot the moving averages with two different colors for visual reference.

## 4.18.3 Applying a strategy to a chart

To test a strategy, apply it to the chart. You can use a built-in strategy from the “Indicators & Strategies” dialog box or write your own in the Pine Editor. Click “Add to chart” from the “Pine Editor” tab to apply a script to the chart:



After a strategy script is compiled and applied to a chart, it will plot order marks on the main chart pane and display simulated performance results in the “Strategy Tester” tab below:



**Note:** The results from a strategy applied to non-standard charts (Heikin Ashi, Renko, Line Break, Kagi, Point & Figure, and Range) do not reflect actual market conditions by default. Strategy scripts will use the synthetic price values from these charts during simulation, which often do not align with actual market prices and will thus produce unrealistic backtest results. We therefore highly recommend using standard chart types for backtesting strategies. Alternatively, on Heikin Ashi charts, users can simulate orders using actual prices by enabling the “Fill orders using standard OHLC” option in the Strategy properties.

---

### 4.18.4 Strategy tester

The Strategy Tester module is available to all scripts declared with the `strategy()` function. Users can access this module from the “Strategy Tester” tab below their charts, where they can conveniently visualize their strategies and analyze hypothetical performance results.

## Overview

The **Overview** tab of the Strategy Tester presents essential performance metrics and equity and drawdown curves over a simulated sequence of trades, providing a quick look at strategy performance without diving into granular detail. The chart in this section shows the strategy's **equity curve** as a baseline plot centered at the initial value, the **buy and hold equity curve** as a line plot, and the **drawdown curve** as a histogram plot. Users can toggle these plots and scale them as absolute values or percentages using the options below the chart.



**Note that:**

- The overview chart uses two scales; the left is for the equity curves, and the right is for the drawdown curve.
- When a user clicks a point on these plots, this will direct the main chart view to the point where the trade was closed.

## Performance summary

The **Performance Summary** tab of the module presents a comprehensive overview of a strategy's performance metrics. It displays three columns: one for all trades, one for all longs, and one for all shorts, to provide traders with more detailed insights on a strategy's long, short, and overall simulated trading performance.

Overview	Performance Summary	List of Trades	Properties	Deep Backtesting <small>BETA</small>
Title	All	Long	Short	
Net Profit	2 778.27 USDT 0.2%	2 049.48 USDT 0.2%	728.79 USDT 0.07%	
Gross Profit	27 631.14 USDT 2.76%	14 308.03 USDT 1.43%	13 323.11 USDT 1.33%	
Gross Loss	24 852.87 USDT 2.49%	12 258.55 USDT 1.23%	12 594.32 USDT 1.26%	
Max Run-up	3 054.07 USDT 0.3%			
Max Drawdown	2 612.62 USDT 0.26%			
Buy & Hold Return	1 527 822.71 USDT 152.78%			
Sharpe Ratio	-2.456			
Sortino Ratio	-0.927			
Profit Factor	1.112	1.167	1.058	
Max Contracts Held	1	1	1	
Open PL	143.09 USDT 0.01%			
Commission Paid	0.00 USDT	0.00 USDT	0.00 USDT	
Total Closed Trades	787	394	393	
Total Open Trades	1	0	1	
Number Winning Trades	306	161	145	
Number Losing Trades	480	232	248	

## List of trades

The [List of Trades](#) tab provides a granular look at the trades simulated by a strategy with essential information, including the date and time of execution, the type of order used (entry or exit), the number of contracts/shares/lots/units traded, and the price, as well as some key trade performance metrics.

Trade # ↑	Type	Signal	Date/Time	Price	Contracts	Profit	Cum. Profit	Run-up	Drawdown
1	Exit Long	sell	2021-01-05 16:00	1 052.83 USDT	1	282.06 USDT 36.59%	282.06 USDT 0.03%	392.20 USDT 50.88%	20.65 USDT 2.68%
	Entry Long	buy	2021-01-02 14:00	770.77 USDT					
2	Exit Short	buy	2021-01-05 19:00	1 074.00 USDT	1	-21.17 USDT -2.01%	260.89 USDT 0%	16.54 USDT 1.57%	42.03 USDT 3.99%
	Entry Short	sell	2021-01-05 16:00	1 052.83 USDT					
3	Exit Long	sell	2021-01-08 06:00	1 148.29 USDT	1	74.29 USDT 6.92%	335.18 USDT 0.01%	215.00 USDT 20.02%	34.00 USDT 3.17%
	Entry Long	buy	2021-01-05 19:00	1 074.00 USDT					
4	Exit Short	buy	2021-01-08 19:00	1 182.55 USDT	1	-34.26 USDT -2.98%	300.92 USDT 0%	22.88 USDT 1.99%	125.46 USDT 10.92%
	Entry Short	sell	2021-01-08 06:00	1 148.29 USDT					
5	Exit Long	sell	2021-01-09 06:00	1 185.40 USDT	1	2.85 USDT 0.24%	303.77 USDT 0%	52.12 USDT 4.41%	47.04 USDT 3.98%
	Entry Long	buy	2021-01-08 19:00	1 182.55 USDT					
6	Exit Short	buy	2021-01-09 16:00	1 219.96 USDT	1	-34.56 USDT -2.92%	269.21 USDT 0%	11.75 USDT 0.99%	51.60 USDT 4.35%
	Entry Short	sell	2021-01-09 06:00	1 185.40 USDT					
7	Exit Long	sell	2021-01-10 22:00	1 270.59 USDT	1	50.63 USDT 4.15%	319.84 USDT 0.01%	128.37 USDT 10.52%	49.96 USDT 4.1%

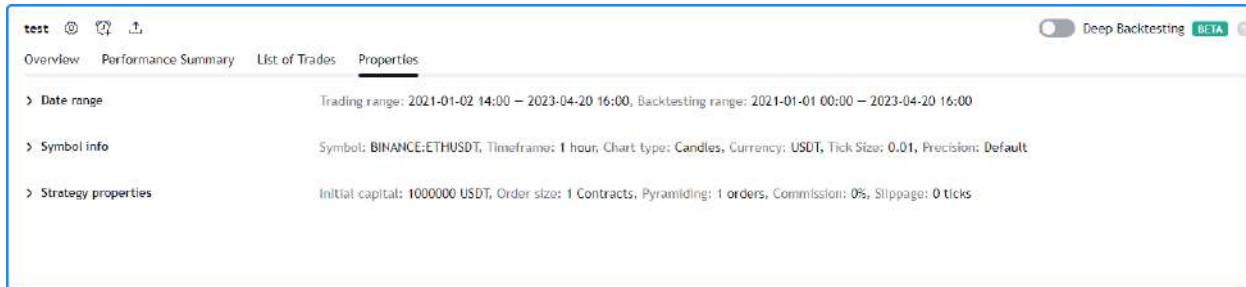
Note that:

- Users can navigate the times of specific trades on their charts by clicking on them in this list.
- By clicking the “Trade #” field above the list, users can organize the trades in ascending order starting from the first or descending order starting from the last.

## Properties

The Properties tab provides detailed information about a strategy’s configuration and the dataset to which it is applied. It includes the strategy’s date range, symbol information, script settings, and strategy properties.

- **Date Range** - Includes the range of dates with simulated trades and the total available backtesting range.
- **Symbol Info** - Contains the symbol name and broker/exchange, the chart’s timeframe and type, the tick size, the point value for the chart, and the base currency.
- **Strategy Inputs** - Outlines the various parameters and variables used in the strategy script available in the “Inputs” tab of the script settings.
- **Strategy Properties** - Provides an overview of the configuration of the trading strategy. It includes essential details such as the initial capital, base currency, order size, margin, pyramiding, commission, and slippage. Additionally, this section highlights any modifications made to strategy calculation behavior.

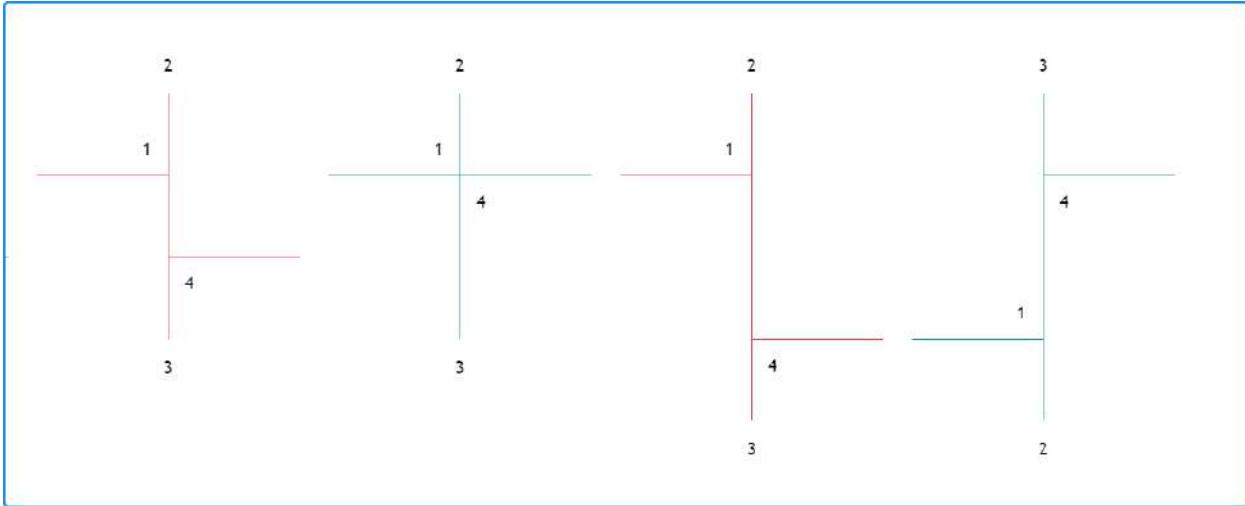


### 4.18.5 Broker emulator

TradingView utilizes a *broker emulator* to simulate the performance of trading strategies. Unlike in real-life trading, the emulator strictly uses available chart prices for order simulation. Consequently, the simulation can only place historical trades after a bar closes, and it can only place real-time trades on a new price tick. For more information on this behavior, please refer to the Pine Script™ [Execution model](#).

Since the emulator can only use chart data, it makes assumptions about intrabar price movement. It uses a bar’s open, high, and low prices to infer intrabar activity while calculating order fills with the following logic:

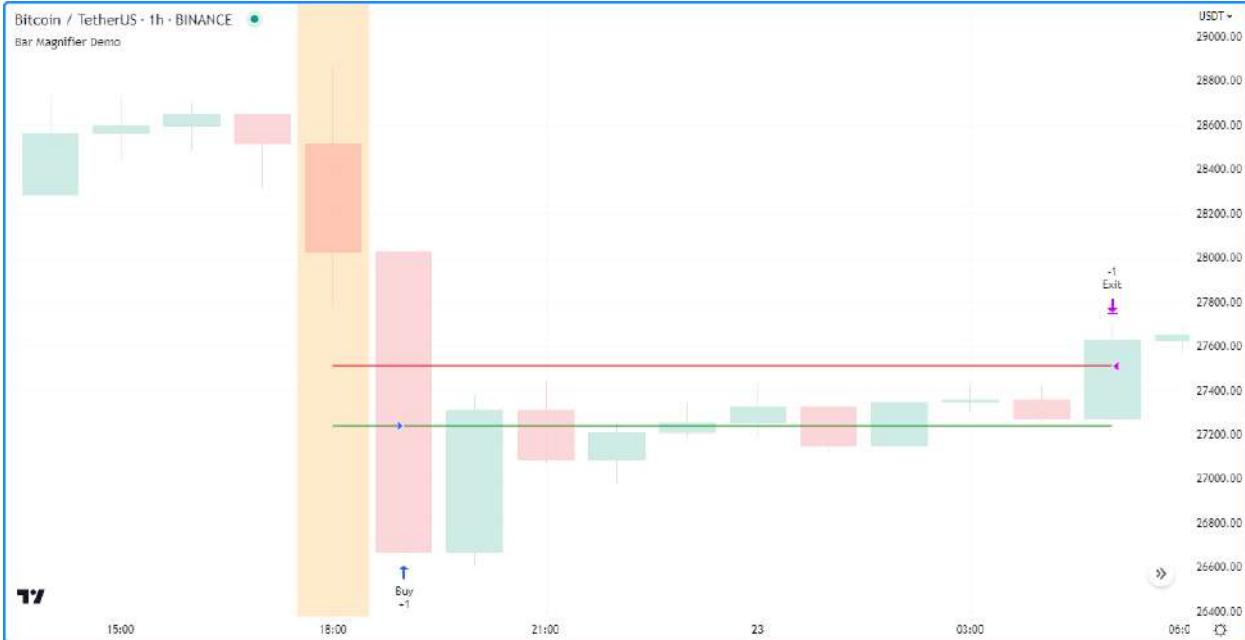
- If the high price is closer to the opening price than the low price, it assumes that the price moved in this order on the bar: open → high → low → close.
- If the low price is closer to the opening price than the high price, it assumes that the price moved in this order on the bar: open → low → high → close.
- The broker emulator assumes no gaps exist between prices within bars; in the “eyes” of the emulator, the full range of intrabar prices is available for order execution.



### Bar magnifier

Premium account holders can override the broker emulator's intrabar assumptions via the `use_bar_magnifier` parameter of the `strategy()` function or the “Use bar magnifier” input in the “Properties” tab of the script settings. The [Bar Magnifier](#) inspects data on timeframes smaller than the chart's to obtain more granular information about price action within a bar, thus allowing more precise order fills during simulation.

To demonstrate, the following script places a “Buy” limit order at the `entryPrice` and an “Exit” limit order at the `exitPrice` when the `time` value crosses the `orderTime`, and draws two horizontal lines to visualize the order prices. The script also highlights the background using the `orderColor` to indicate when the strategy placed the orders:



```

1 //@version=5
2 strategy("Bar Magnifier Demo", overlay = true, use_bar_magnifier = false)
3
4 //@variable The UNIX timestamp to place the order at.
5 int orderTime = timestamp("UTC", 2023, 3, 22, 18)

```

(continues on next page)

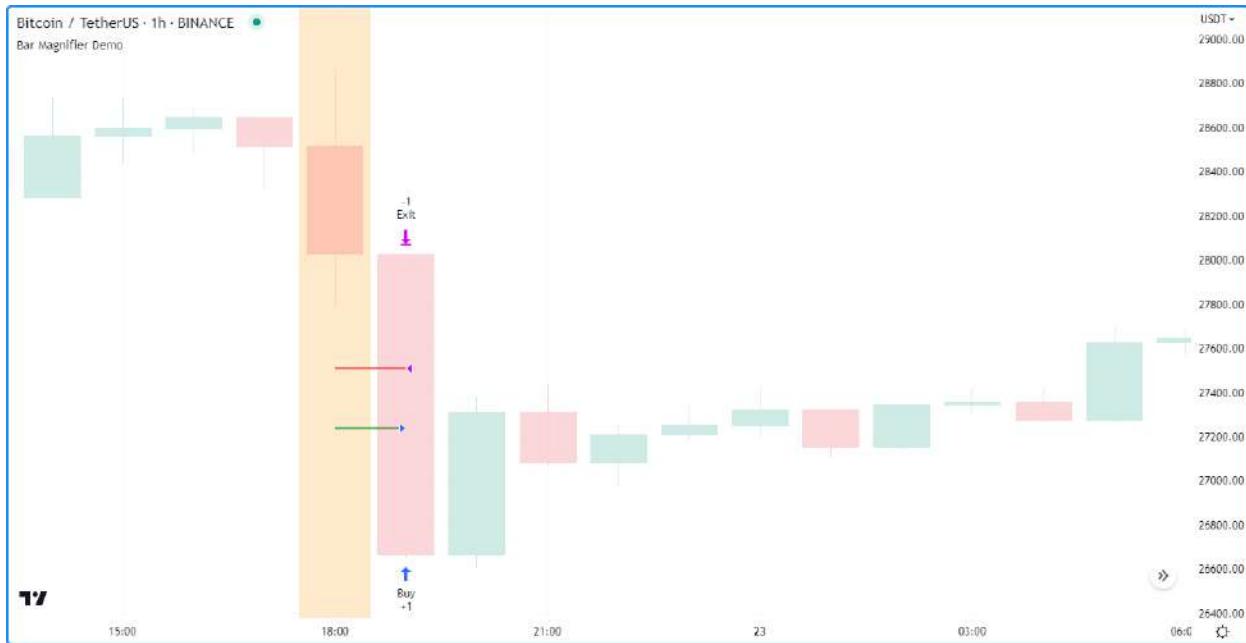
(continued from previous page)

```

6 // @variable Returns `color.orange` when `time` crosses the `orderTime`, false otherwise.
7 color orderColor = na
8
9 // Entry and exit prices.
10 float entryPrice = h12 - (high - low)
11 float exitPrice = entryPrice + (high - low) * 0.25
12
13 // Entry and exit lines.
14 var line entryLine = na
15 var line exitLine = na
16
17 if ta.cross(time, orderTime)
18     // Draw new entry and exit lines.
19     entryLine := line.new(bar_index, entryPrice, bar_index + 1, entryPrice, color = color.green, width = 2)
20     exitLine := line.new(bar_index, exitPrice, bar_index + 1, exitPrice, color = color.red, width = 2)
21
22     // Update order highlight color.
23     orderColor := color.new(color.orange, 80)
24
25     // Place limit orders at the `entryPrice` and `exitPrice`.
26     strategy.entry("Buy", strategy.long, limit = entryPrice)
27     strategy.exit("Exit", "Buy", limit = exitPrice)
28
29 // Update lines while the position is open.
30 else if strategy.position_size > 0.0
31     entryLine.set_x2(bar_index + 1)
32     exitLine.set_x2(bar_index + 1)
33
34 bgcolor(orderColor)
35

```

As we see in the chart above, the broker emulator assumed that intrabar prices moved from open to high, then high to low, then low to close on the bar the “Buy” order filled on, meaning the emulator assumed that the “Exit” order couldn’t fill on the same bar. However, after including `use_bar_magnifier = true` in the declaration statement, we see a different story:



**Note:** The maximum amount of intrabars that a script can request is 200,000. Some symbols with lengthier history may not have full intrabar coverage for their beginning chart bars with this limitation, meaning that simulated trades on those bars will not be affected by the bar magnifier.

#### 4.18.6 Orders and entries

Just like in real-life trading, Pine strategies use orders to manage positions. In this context, an *order* is a command to simulate a market action, and a *trade* is the result after the order fills. Thus, to enter or exit positions using Pine, users must create orders with parameters that specify how they'll behave.

To take a closer look at how orders work and how they become trades, let's write a simple strategy script:

```

1 //@version=5
2 strategy("My strategy", overlay = true, margin_long = 100, margin_short = 100)
3
4 //@function Displays text passed to `txt` when called.
5 debugLabel(txt) =>
6     label.new(
7         bar_index, high, text = txt, color=color.lime, style = label.style_label_
8         ↪lower_right,
9             textcolor = color.black, size = size.large
10        )
11
12 longCondition = bar_index % 20 == 0 // true on every 20th bar
13 if (longCondition)
14     debugLabel("Long entry order created")
15     strategy.entry("My Long Entry Id", strategy.long)
16 strategy.close_all()

```

In this script, we've defined a `longCondition` that is true whenever the `bar_index` is divisible by 20, i.e., every 20th bar. The strategy uses this condition within an `if` structure to simulate an entry order with `strategy.entry()` and draws a label at the entry price with the user-defined `debugLabel()` function. The script calls `strategy.close_all()` from the

global scope to simulate a market order that closes any open position. Let's see what happens once we add the script to our chart:



The blue arrows on the chart indicate entry locations, and the purple ones mark the points where the strategy closed positions. Notice that the labels precede the actual entry point rather than occurring on the same bar - this is orders in action. By default, Pine strategies wait for the next available price tick before filling orders, as filling an order on the same tick isn't realistic. Also, they recalculate on the close of every historical bar, meaning the next available tick to fill an order at is the open of the next bar in this case. As a result, by default, all orders are delayed by one chart bar.

It's important to note that although the script calls `strategy.close_all()` from the global scope, forcing execution on every bar, the function call does nothing if the strategy isn't simulating an open position. If there is an open position, the command issues a market order to close it, which executes on the next available tick. For example, when the `longCondition` is true on bar 20, the strategy places an entry order to fill at the next tick, which is at the open of bar 21. Once the script recalculates its values on that bar's close, the function places an order to close the position, which fills at the open of bar 22.

## Order types

Pine Script™ strategies allow users to simulate different order types for their particular needs. The main notable types are *market*, *limit*, *stop*, and *stop-limit*.

## Market orders

Market orders are the most basic type of orders. They command a strategy to buy or sell a security as soon as possible, regardless of the price. Consequently, they always execute on the next available price tick. By default, all `strategy.*()` functions that generate orders specifically produce market orders.

The following script simulates a long market order when the `bar_index` is divisible by `2 * cycleLength`. Otherwise, it simulates a short market order when the `bar_index` is divisible by `cycleLength`, resulting in a strategy with alternating long and short trades once every `cycleLength` bars:



```

1 //@version=5
2 strategy("Market order demo", overlay = true, margin_long = 100, margin_short = 100)
3
4 //@variable Number of bars between long and short entries.
5 cycleLength = input.int(10, "Cycle length")
6
7 //@function Displays text passed to `txt` when called.
8 debugLabel(txt, lblColor) => label.new(
9     bar_index, high, text = txt, color = lblColor, textcolor = color.white,
10    style = label.style_label_lower_right, size = size.large
11 )
12
13 //@variable Returns `true` every `2 * cycleLength` bars.
14 longCondition = bar_index % (2 * cycleLength) == 0
15 //@variable Returns `true` every `cycleLength` bars.
16 shortCondition = bar_index % cycleLength == 0
17
18 // Generate a long market order with a `color.green` label on `longCondition`.
19 if longCondition
20     debugLabel("Long market order created", color.green)
21     strategy.entry("My Long Entry Id", strategy.long)
22 // Otherwise, generate a short market order with a `color.red` label on
23 // `shortCondition`.
24 else if shortCondition
25     debugLabel("Short market order created", color.red)
26     strategy.entry("My Short Entry Id", strategy.short)

```

## Limit orders

Limit orders command a strategy to enter a position at a specific price or better (lower than specified for long orders and higher for short ones). When the current market price is better than the order command's limit parameter, the order will fill without waiting for the market price to reach the limit level.

To simulate limit orders in a script, pass a price value to an order placement command with a `limit` parameter. The following example places a limit order 800 ticks below the bar close 100 bars before the `last_bar_index`:



```

1 // @version=5
2 strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100)
3
4 // @function Displays text passed to `txt` and a horizontal line at `price` when
5 // called.
6 debugLabel(price, txt) =>
7     label.new(
8         bar_index, price, text = txt, color = color.teal, textcolor = color.white,
9         style = label.style_label_lower_right, size = size.large
10    )
11    line.new(
12        bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.
13        right,
14        style = line.style_dashed
15    )
16
17 // Generate a long limit order with a label and line 100 bars before the `last_bar_
18 // index`.
19 if last_bar_index - bar_index == 100
20     limitPrice = close - syminfo.mintick * 800
21     debugLabel(limitPrice, "Long Limit order created")
22     strategy.entry("Long", strategy.long, limit = limitPrice)

```

Note how the script placed the label and started the line several bars before the trade. As long as the price remained above the `limitPrice` value, the order could not fill. Once the market price reached the limit, the strategy executed the trade mid-bar. If we had set the `limitPrice` to 800 ticks *above* the bar close rather than *below*, the order would fill immediately since the price is already at a better value:



```

1 //@version=5
2 strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100)
3
4 // Function Displays text passed to `txt` and a horizontal line at `price` when
5 // called.
6 debugLabel(price, txt) =>
7     label.new(
8         bar_index, price, text = txt, color = color.teal, textcolor = color.white,
9         style = label.style_label_lower_right, size = size.large
10    )
11    line.new(
12        bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.
13        right,
14        style = line.style_dashed
15    )
16
17 // Generate a long limit order with a label and line 100 bars before the `last_bar_
18 // index`.
19 if last_bar_index - bar_index == 100
20     limitPrice = close + syminfo.mintick * 800
21     debugLabel(limitPrice, "Long Limit order created")
22     strategy.entry("Long", strategy.long, limit = limitPrice)

```

### Stop and stop-limit orders

Stop orders command a strategy to simulate another order after price reaches the specified `stop` price or a worse value (higher than specified for long orders and lower for short ones). They are essentially the opposite of limit orders. When the current market price is worse than the `stop` parameter, the strategy will trigger the subsequent order without waiting for the current price to reach the stop level. If the order placement command includes a `limit` argument, the subsequent order will be a limit order at the specified value. Otherwise, it will be a market order.

The script below places a stop order 800 ticks above the `close` 100 bars ago. In this example, the strategy entered a long position when the market price crossed the `stop` price some bars after it placed the order. Notice that the initial price at the time of the order was better than the one passed to `stop`. An equivalent limit order would have filled on the

following chart bar:



```

1 //@version=5
2 strategy("Stop order demo", overlay = true, margin_long = 100, margin_short = 100)
3
4 //@function Displays text passed to `txt` when called and shows the `price` level on_
5 //the chart.
6 debugLabel(price, txt) =>
7     label.new(
8         bar_index, high, text = txt, color = color.teal, textcolor = color.white,
9         style = label.style_label_lower_right, size = size.large
10    )
11    line.new(bar_index, high, bar_index, price, style = line.style_dotted, color =
12 //color.teal)
13    line.new(
14        bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.
15 //right,
16        style = line.style_dashed
17    )
18
19 // Generate a long stop order with a label and lines 100 bars before the last bar.
20 if last_bar_index - bar_index == 100
21     stopPrice = close + syminfo.mintick * 800
22     debugLabel(stopPrice, "Long Stop order created")
23     strategy.entry("Long", strategy.long, stop = stopPrice)

```

Order placement commands that use both `limit` and `stop` arguments produce stop-limit orders. This order type waits for the price to cross the stop level, then places a limit order at the specified `limit` price.

Let's modify our previous script to simulate and visualize a stop-limit order. In this example, we use the `low` value from 100 bars ago as the `limit` price in the `entry` command. This script also displays a label and price level to indicate when the strategy crosses the `stopPrice`, i.e., when the strategy activates the limit order. Notice how the market price initially reaches the limit level, but the strategy doesn't simulate a trade because the price must cross the `stopPrice` to place the pending limit order at the `limitPrice`:



```

1 //@version=5
2 strategy("Stop-Limit order demo", overlay = true, margin_long = 100, margin_short = -100)
3
4 //@function Displays text passed to `txt` when called and shows the `price` level on the chart.
5 debugLabel(price, txt, lblColor, lineWidth = 1) =>
6     label.new(
7         bar_index, high, text = txt, color = lblColor, textcolor = color.white,
8         style = label.style_label_lower_right, size = size.large
9     )
10    line.new(bar_index, close, bar_index, price, style = line.style_dotted, color = -lblColor, width = lineWidth)
11    line.new(
12        bar_index, price, bar_index + 1, price, color = lblColor, extend = extend.right,
13        style = line.style_dashed, width = lineWidth
14    )
15
16 var float stopPrice = na
17 var float limitPrice = na
18
19 // Generate a long stop-limit order with a label and lines 100 bars before the last bar.
20 if last_bar_index - bar_index == 100
21     stopPrice := close + syminfo.mintick * 800
22     limitPrice := low
23     debugLabel(limitPrice, "", color.gray)
24     debugLabel(stopPrice, "Long Stop-Limit order created", color.teal)
25     strategy.entry("Long", strategy.long, stop = stopPrice, limit = limitPrice)
26
27 // Draw a line and label once the strategy activates the limit order.
28 if high >= stopPrice
29     debugLabel(limitPrice, "Limit order activated", color.green, 2)
30     stopPrice := na

```

## Order placement commands

Pine Script™ strategies feature several functions to simulate the placement of orders, known as *order placement commands*. Each command serves a unique purpose and behaves differently from the others.

### `'strategy.entry()'`

This command simulates entry orders. By default, strategies place market orders when calling this function, but they can also create stop, limit, and stop-limit orders when utilizing the `stop` and `limit` parameters.

To simplify opening positions, `strategy.entry()` features several unique behaviors. One such behavior is that this command can reverse an open market position without additional function calls. When an order placed using `strategy.entry()` fills, the function will automatically calculate the amount the strategy needs to close the open market position and trade `qty` contracts/shares/lots/units in the opposite direction by default. For example, if a strategy has an open position of 15 shares in the `strategy.long` direction and calls `strategy.entry()` to place a market order in the `strategy.short` direction, the amount the strategy will trade to place the order is 15 shares plus the `qty` of the new short order.

The example below demonstrates a strategy that uses only `strategy.entry()` calls to place entry orders. It creates a long market order with a `qty` value of 15 shares once every 100 bars and a short market order with a `qty` of 5 once every 25 bars. The script highlights the background blue and red for occurrences of the respective `buyCondition` and `sellCondition`:



```

1 //@version=5
2 strategy("Entry demo", "test", overlay = true)
3
4 //@variable Is `true` on every 100th bar.
5 buyCondition = bar_index % 100 == 0
6 //@variable Is `true` on every 25th bar except for those that are divisible by 100.
7 sellCondition = bar_index % 25 == 0 and not buyCondition
8
9 if buyCondition
10     strategy.entry("buy", strategy.long, qty = 15)
11 if sellCondition
12     strategy.entry("sell", strategy.short, qty = 5)

```

(continues on next page)

(continued from previous page)

```
13 bgcolor(buyCondition ? color.new(color.blue, 90) : na)
14 bgcolor(sellCondition ? color.new(color.red, 90) : na)
15
```

As we see in the chart above, the order marks show that the strategy traded 20 shares on each order fill rather than 15 and 5. Since `strategy.entry()` automatically reverses positions, unless otherwise specified via the `strategy.risk.allow_entry_in()` function, it adds the open position size (15 for long entries) to the new order's size (5 for short entries) when it changes the direction, resulting in a traded quantity of 20 shares.

Notice that in the above example, although the `sellCondition` occurs three times before another `buyCondition`, the strategy only places a “sell” order on the first occurrence. Another unique behavior of the `strategy.entry()` command is that it's affected by a script's *pyramiding* setting. Pyramiding specifies the number of consecutive orders the strategy can fill in the same direction. Its value is 1 by default, meaning the strategy only allows one consecutive order to fill in either direction. Users can set the strategy pyramiding values via the `pyramiding` parameter of the `strategy()` function call or the “Pyramiding” input in the “Properties” tab of the script settings.

If we add `pyramiding = 3` to our previous script's declaration statement, the strategy will allow up to three consecutive trades in the same direction, meaning it can simulate new market orders on each occurrence of `sellCondition`:



### `'strategy.order()'`

This command simulates a basic order. Unlike most order placement commands, which contain internal logic to simplify interfacing with strategies, `strategy.order()` uses the specified parameters without accounting for most additional strategy settings. Orders placed by `strategy.order()` can open new positions and modify or close existing ones.

The following script uses only `strategy.order()` calls to create and modify entries. The strategy simulates a long market order for 15 units every 100 bars, then three short orders for five units every 25 bars. The script highlights the background blue and red to indicate when the strategy simulates “buy” and “sell” orders:



```

1 //@version=5
2 strategy("Order demo", "test", overlay = true)
3
4 //@variable Is `true` on every 100th bar.
5 buyCond = bar_index % 100 == 0
6 //@variable Is `true` on every 25th bar except for those that are divisible by 100.
7 sellCond = bar_index % 25 == 0 and not buyCond
8
9 if buyCond
10     strategy.order("buy", strategy.long, qty = 15) // Enter a long position of 15
11         ↵units.
12 if sellCond
13     strategy.order("sell", strategy.short, qty = 5) // Exit 5 units from the long
14         ↵position.
15 bgcolor(buyCond ? color.new(color.blue, 90) : na)
16 bgcolor(sellCond ? color.new(color.red, 90) : na)

```

This particular strategy will never simulate a short position, as unlike `strategy.entry()`, `strategy.order()` does not automatically reverse positions. When using this command, the resulting market position is the net sum of the current market position and the filled order quantity. After the strategy fills the “buy” order for 15 units, it executes three “sell” orders that reduce the open position by five units each, and  $15 - 5 * 3 = 0$ . The same script would behave differently using `strategy.entry()`, as per the example shown in the [section above](#).

### `strategy.exit()`

This command simulates exit orders. It's unique in that it allows a strategy to exit a market position or form multiple exits in the form of stop-loss, take-profit, and trailing stop orders via the `loss`, `stop`, `profit`, `limit`, and `trail_*` parameters.

The most basic use of the `strategy.exit()` command is the creation of levels where the strategy will exit a position due to losing too much money (`stop-loss`), earning enough money (`take-profit`), or both (bracket).

The stop-loss and take-profit functionalities of this command are associated with two parameters. The function's `loss` and `profit` parameters specify stop-loss and take-profit values as a defined number of ticks away from the entry order's price, while its `stop` and `limit` parameters provide specific stop-loss and take-profit price values. The absolute parameters in the function call supersede the relative ones. If a `strategy.exit()` call contains `profit` and `limit` arguments, the command will prioritize the `limit` value and ignore the `profit` value. Likewise, it will only consider the `stop` value when the function call contains `stop` and `loss` arguments.

---

**Note:** Despite sharing the same names with parameters from `strategy.entry()` and `strategy.order()` commands, the `limit` and `stop` parameters work differently in `strategy.exit()`. In the first case, using `limit` and `stop` in the command will create a single stop-limit order that opens a limit order after crossing the stop price. In the second case, the command will create a separate limit and stop order to exit from an open position.

---

All exit orders from `strategy.exit()` with a `from_entry` argument are bound to the `id` of a corresponding entry order; strategies cannot simulate exit orders when there is no open market position or active entry order associated with a `from_entry` ID.

The following strategy places a “buy” entry order via `strategy.entry()` and a stop-loss and take-profit order via the `strategy.exit()` command every 100 bars. Notice that the script calls `strategy.exit()` twice. The “exit1” command references a “buy1” entry order, and “exit2” references the “buy” order. The strategy will only simulate exit orders from “exit2” because “exit1” references an order ID that doesn't exist:



```

1 //@version=5
2 strategy("Exit demo", "test", overlay = true)
3

```

(continues on next page)

(continued from previous page)

```

4 // @variable Is `true` on every 100th bar.
5 buyCondition = bar_index % 100 == 0
6
7 // @variable Stop-loss price for exit commands.
8 var float stopLoss = na
9 // @variable Take-profit price for exit commands.
10 var float takeProfit = na
11
12 // Place orders upon `buyCondition`.
13 if buyCondition
14     if strategy.position_size == 0.0
15         stopLoss := close * 0.99
16         takeProfit := close * 1.01
17         strategy.entry("buy", strategy.long)
18         strategy.exit("exit1", "buy1", stop = stopLoss, limit = takeProfit) // Does
19             ↪nothing. "buy1" order doesn't exist.
20         strategy.exit("exit2", "buy", stop = stopLoss, limit = takeProfit)
21
22 // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the
23             ↪strategy simulates an exit.
24 if low <= stopLoss or high >= takeProfit
25     stopLoss := na
26     takeProfit := na
27
28 plot(stopLoss, "SL", color.red, style = plot.style_circles)
29 plot(takeProfit, "TP", color.green, style = plot.style_circles)

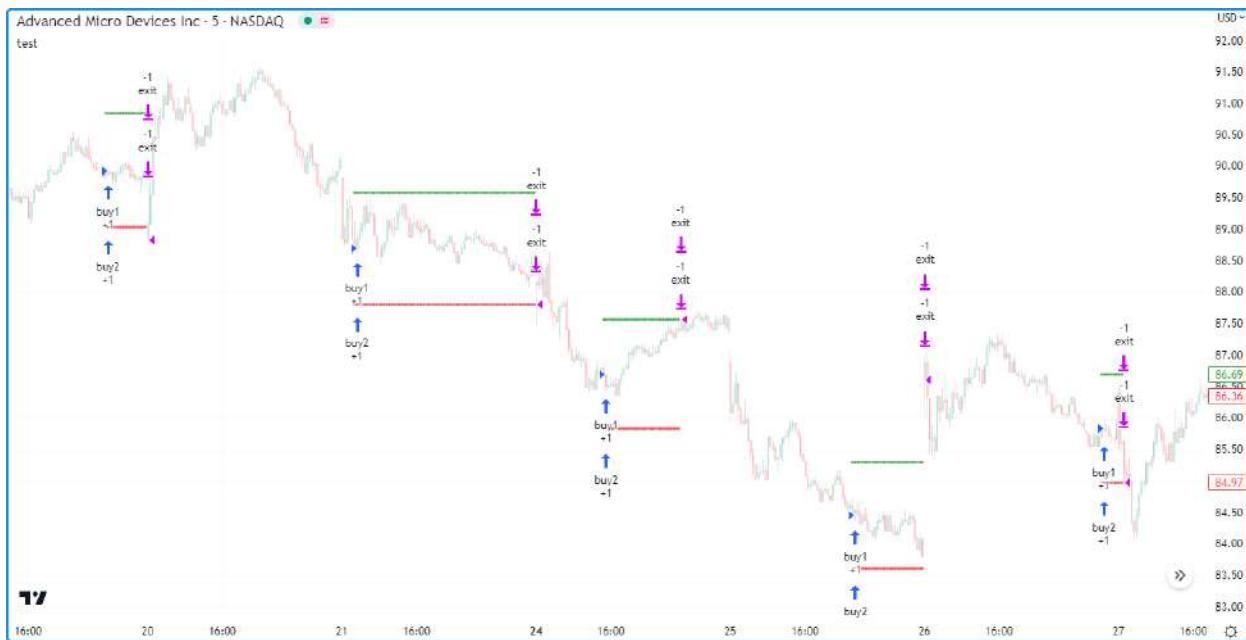
```

**Note that:**

- Limit and stop orders from each exit command do not necessarily fill at the specified prices. Strategies can fill limit orders at better prices and stop orders at worse prices, depending on the range of values available to the broker emulator.

If a user does not provide a `from_entry` argument in the `strategy.exit()` call, the function will create exit orders for each open entry.

In this example, the strategy creates “buy1” and “buy2” entry orders and calls `strategy.exit()` without a `from_entry` argument every 100 bars. As we can see from the order marks on the chart, once the market price reaches the `stopLoss` or `takeProfit` values, the strategy fills an exit order for both “buy1” and “buy2” entries:



```

1 //@version=5
2 strategy("Exit all demo", "test", overlay = true, pyramiding = 2)
3
4 //@variable Is `true` on every 100th bar.
5 buyCondition = bar_index % 100 == 0
6
7 //@variable Stop-loss price for exit commands.
8 var float stopLoss    = na
9 //@variable Take-profit price for exit commands.
10 var float takeProfit = na
11
12 // Place orders upon `buyCondition`.
13 if buyCondition
14     if strategy.position_size == 0.0
15         stopLoss    := close * 0.99
16         takeProfit := close * 1.01
17         strategy.entry("buy1", strategy.long)
18         strategy.entry("buy2", strategy.long)
19         strategy.exit("exit", stop = stopLoss, limit = takeProfit) // Places orders to
20             ↪exit all open entries.
21
22 // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the
23             ↪strategy simulates an exit.
24 if low <= stopLoss or high >= takeProfit
25     stopLoss    := na
26     takeProfit := na
27
28 plot(stopLoss, "SL", color.red, style = plot.style_circles)
29 plot(takeProfit, "TP", color.green, style = plot.style_circles)

```

It is possible for a strategy to exit from the same entry ID more than once, which facilitates the formation of multi-level exit strategies. When performing multiple exit commands, each order's quantity must be a portion of the traded quantity, with their sum not exceeding the open position. If the `qty` of the function is less than the size of the current market position, the strategy will simulate a partial exit. If the `qty` value exceeds the open position quantity, it will reduce the order since it cannot fill more contracts/shares/lots/units than the open position.

In the example below, we've modified our previous "Exit demo" script to simulate two stop-loss and take-profit orders per entry. The strategy places a "buy" order with a `qty` of two shares, "exit1" stop-loss and take-profit orders with a `qty` of one share, and "exit2" stop-loss and take profit orders with a `qty` of three shares:



```

1 //@version=5
2 strategy("Multiple exit demo", "test", overlay = true)
3
4 //@variable Is `true` on every 100th bar.
5 buyCondition = bar_index % 100 == 0
6
7 //@variable Stop-loss price for "exit1" commands.
8 var float stopLoss1 = na
9 //@variable Stop-loss price for "exit2" commands.
10 var float stopLoss2 = na
11 //@variable Take-profit price for "exit1" commands.
12 var float takeProfit1 = na
13 //@variable Take-profit price for "exit2" commands.
14 var float takeProfit2 = na
15
16 // Place orders upon `buyCondition`.
17 if buyCondition
18     if strategy.position_size == 0.0
19         stopLoss1 := close * 0.99
20         stopLoss2 := close * 0.98
21         takeProfit1 := close * 1.01
22         takeProfit2 := close * 1.02
23         strategy.entry("buy", strategy.long, qty = 2)
24         strategy.exit("exit1", "buy", stop = stopLoss1, limit = takeProfit1, qty = 1)
25         strategy.exit("exit2", "buy", stop = stopLoss2, limit = takeProfit2, qty = 3)
26
27 // Set `stopLoss1` and `takeProfit1` to `na` when price touches either.
28 if low <= stopLoss1 or high >= takeProfit1
29     stopLoss1 := na
30     takeProfit1 := na
31 // Set `stopLoss2` and `takeProfit2` to `na` when price touches either.

```

(continues on next page)

(continued from previous page)

```

32 if low <= stopLoss2 or high >= takeProfit2
33     stopLoss2 := na
34     takeProfit2 := na
35
36 plot(stopLoss1, "SL1", color.red, style = plot.style_circles)
37 plot(stopLoss2, "SL2", color.red, style = plot.style_circles)
38 plot(takeProfit1, "TP1", color.green, style = plot.style_circles)
39 plot(takeProfit2, "TP2", color.green, style = plot.style_circles)

```

As we can see from the order marks on the chart, the strategy filled “exit2” orders despite the specified `qty` value exceeding the traded amount. Rather than using this quantity, the script reduced the orders’ sizes to match the remaining position.

### Note that:

- All orders generated from a `strategy.exit()` call belong to the same `strategy.oca.reduce` group, meaning that when either order fills, the strategy reduces all others to match the open position.

It’s important to note that orders produced by this command reserve a portion of the open market position to exit. `strategy.exit()` cannot place an order to exit a portion of the position already reserved for exit by another exit command.

The following script simulates a “buy” market order for 20 shares 100 bars ago with “limit” and “stop” orders of 19 and 20 shares respectively. As we see on the chart, the strategy executed the “stop” order first. However, the traded quantity was only one share. Since the script placed the “limit” order first, the strategy reserved its `qty` (19 shares) to close the open position, leaving only one share to be closed by the “stop” order:



```

1 //@version=5
2 strategy("Reserved exit demo", "test", overlay = true)
3
4 //@variable "stop" exit order price.
5 var float stop    = na
6 //@variable "limit" exit order price
7 var float limit   = na
8 //@variable Is `true` 100 bars before the `last_bar_index`.
9 longCondition = last_bar_index - bar_index == 100

```

(continues on next page)

(continued from previous page)

```

10
11 if longCondition
12   stop := close * 0.99
13   limit := close * 1.01
14   strategy.entry("buy", strategy.long, 20)
15   strategy.exit("limit", limit = limit, qty = 19)
16   strategy.exit("stop", stop = stop, qty = 20)
17
18 bool showPlot = strategy.position_size != 0
19 plot(showPlot ? stop : na, "Stop", color.red, 2, plot.style_linebr)
20 plot(showPlot ? limit : na, "Limit 1", color.green, 2, plot.style_linebr)

```

Another key feature of the `strategy.exit()` function is that it can create *trailing stops*, i.e., stop-loss orders that trail behind the market price by a specified amount whenever the price moves to a better value in the favorable direction. These orders have two components: the activation level and the trail offset. The activation level is the value the market price must cross to activate the trailing stop calculation, expressed in ticks via the `trail_points` parameter or as a price value via the `trail_price` parameter. If an exit call contains both arguments, the `trail_price` argument takes precedence. The trail offset is the distance the stop will follow behind the market price, expressed in ticks via the `trail_offset` parameter. For `strategy.exit()` to create and activate trailing stops, the function call must contain `trail_offset` and either `trail_price` or `trail_points` arguments.

The example below shows a trailing stop in action and visualizes its behavior. The strategy simulates a long entry order on the bar 100 bars before the last bar on the chart, then a trailing stop on the next bar. The script has two inputs: one controls the activation level offset (i.e., the amount past the entry price required to activate the stop), and the other controls the trail offset (i.e., the distance to follow behind the market price when it moves to a better value in the desired direction).

The green dashed line on the chart shows the level the market price must cross to trigger the trailing stop order. After the price crosses this level, the script plots a blue line to signify the trailing stop. When the price rises to a new high value, which is favorable for the strategy since it means the position's value is increasing, the stop also rises to maintain a distance of `trailingStopOffset` ticks behind the current price. When the price decreases or doesn't reach a new high point, the stop value stays the same. Eventually, the price crosses below the stop, triggering the exit:



```

1 //@version=5
2 strategy("Trailing stop order demo", overlay = true, margin_long = 100, margin_short =
(continues on next page)

```

(continued from previous page)

```

1  ↵= 100)
2
3 // @variable Offset used to determine how far above the entry price (in ticks) the
4 // activation level will be located.
5 activationLevelOffset = input(1000, "Activation Level Offset (in ticks)")
6 // @variable Offset used to determine how far below the high price (in ticks) the
7 // trailing stop will trail the chart.
8 trailingStopOffset = input(2000, "Trailing Stop Offset (in ticks)")
9
10 // @function Displays text passed to `txt` when called and shows the `price` level on
11 // the chart.
12 debugLabel(price, txt, lblColor, hasLine = false) =>
13     label.new(
14         bar_index, price, text = txt, color = lblColor, textcolor = color.white,
15         style = label.style_label_lower_right, size = size.large
16     )
17     if hasLine
18         line.new(
19             bar_index, price, bar_index + 1, price, color = lblColor, extend =
20             extend.right,
21             style = line.style_dashed
22         )
23
24 // @variable The price at which the trailing stop activation level is located.
25 var float trailPriceActivationLevel = na
26 // @variable The price at which the trailing stop itself is located.
27 var float trailingStop = na
28 // @variable Calculates the value that Trailing Stop would have if it were active at
29 // the moment.
30 theoreticalStopPrice = high - trailingStopOffset * syminfo.mintick
31
32 // Generate a long market order to enter 100 bars before the last bar.
33 if last_bar_index - bar_index == 100
34     strategy.entry("Long", strategy.long)
35
36 // Generate a trailing stop 99 bars before the last bar.
37 if last_bar_index - bar_index == 99
38     trailPriceActivationLevel := open + syminfo.mintick * activationLevelOffset
39     strategy.exit(
40         "Trailing Stop", from_entry = "Long", trail_price =
41         trailPriceActivationLevel,
42         trail_offset = trailingStopOffset
43     )
44     debugLabel(trailPriceActivationLevel, "Trailing Stop Activation Level", color.
45         green, true)
46
47 // Visualize the trailing stop mechanic in action.
48 // If there is an open trade, check whether the Activation Level has been achieved.
49 // If it has been achieved, track the trailing stop by assigning its value to a
50 // variable.
51 if strategy.opentrades == 1
52     if na(trailingStop) and high > trailPriceActivationLevel
53         debugLabel(trailPriceActivationLevel, "Activation level crossed", color.green)
54         trailingStop := theoreticalStopPrice
55         debugLabel(trailingStop, "Trailing Stop Activated", color.blue)
56
57     else if theoreticalStopPrice > trailingStop

```

(continues on next page)

(continued from previous page)

```

51     trailingStop := theoreticalStopPrice
52
53 // Visualize the movement of the trailing stop.
54 plot(trailingStop, "Trailing Stop")

```

### **`strategy.close()` and `strategy.close\_all()`**

These commands simulate exit positions using market orders. The functions close trades upon being called rather than at a specific price.

The example below demonstrates a simple strategy that places a “buy” order via `strategy.entry()` once every 50 bars that it closes with a market order using `strategy.close()` 25 bars afterward:



```

1 // @version=5
2 strategy("Close demo", "test", overlay = true)
3
4 // @variable Is `true` on every 50th bar.
5 buyCond = bar_index % 50 == 0
6 // @variable Is `true` on every 25th bar except for those that are divisible by 50.
7 sellCond = bar_index % 25 == 0 and not buyCond
8
9 if buyCond
10     strategy.entry("buy", strategy.long)
11 if sellCond
12     strategy.close("buy")
13
14 bgcolor(buyCond ? color.new(color.blue, 90) : na)
15 bgcolor(sellCond ? color.new(color.red, 90) : na)

```

Unlike most other order placement commands, the `id` parameter of `strategy.close()` references an existing entry ID to close. If the specified `id` does not exist, the command will not execute an order. If a position was formed from multiple entries with the same ID, the command will exit all entries simultaneously.

To demonstrate, the following script places a “buy” order once every 25 bars. The script closes all “buy” entries once every 100 bars. We’ve included `pyramiding = 3` in the `strategy()` declaration statement to allow the strategy to simulate up to three orders in the same direction:



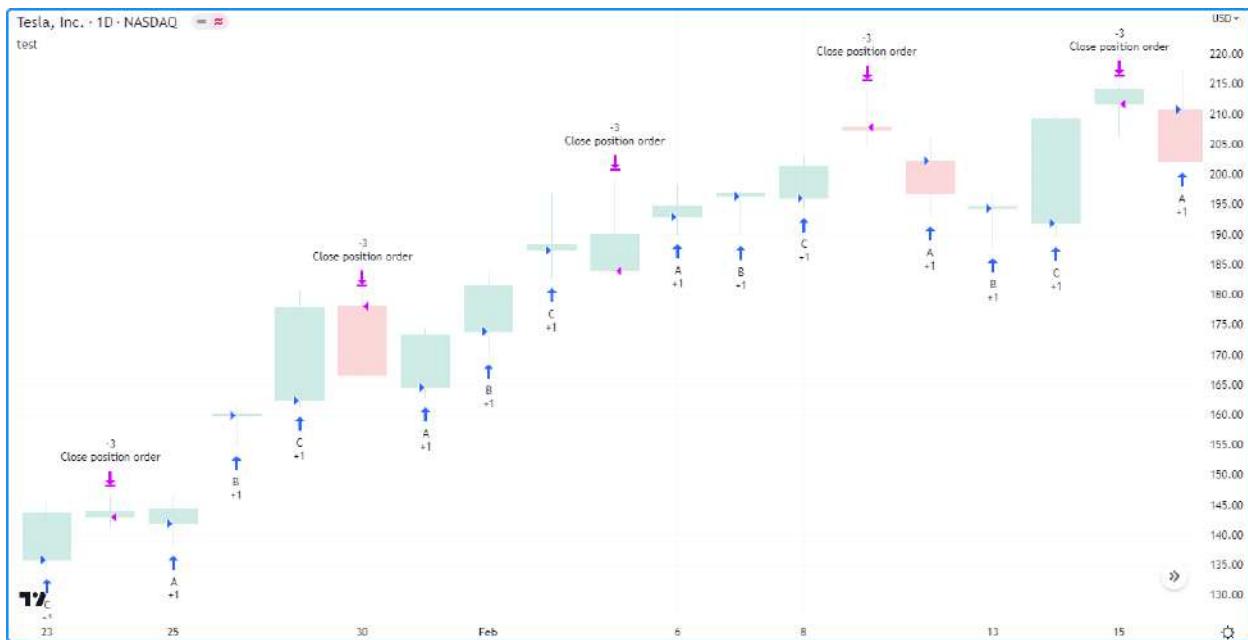
```

1 //@version=5
2 strategy("Multiple close demo", "test", overlay = true, pyramiding = 3)
3
4 //@variable Is `true` on every 100th bar.
5 sellCond = bar_index % 100 == 0
6 //@variable Is `true` on every 25th bar except for those that are divisible by 100.
7 buyCond = bar_index % 25 == 0 and not sellCond
8
9 if buyCond
10     strategy.entry("buy", strategy.long)
11 if sellCond
12     strategy.close("buy")
13
14 bgcolor(buyCond ? color.new(color.blue, 90) : na)
15 bgcolor(sellCond ? color.new(color.red, 90) : na)

```

For cases where a script has multiple entries with different IDs, the `strategy.close_all()` command can come in handy since it closes all entries, irrespective of their IDs.

The script below places “A”, “B”, and “C” entry orders sequentially based on the number of open trades, then closes all of them with a single market order:



```

1 //@version=5
2 strategy("Close multiple ID demo", "test", overlay = true, pyramiding = 3)
3
4 switch strategy.opentrades
5   0 => strategy.entry("A", strategy.long)
6   1 => strategy.entry("B", strategy.long)
7   2 => strategy.entry("C", strategy.long)
8   3 => strategy.close_all()

```

### ``strategy.cancel()` and `strategy.cancel_all()``

These commands allow a strategy to cancel pending orders, i.e., those generated by `strategy.exit()` or by `strategy.order()` or `strategy.entry()` when they use `limit` or `stop` arguments.

The following strategy simulates a “buy” limit order 500 ticks below the closing price 100 bars ago, then cancels the order on the next bar. The script draws a horizontal line at the `limitPrice` and colors the background green and orange to indicate when the limit order is placed and canceled respectively. As we can see, nothing happened once the market price crossed the `limitPrice` because the strategy already canceled the order:



```

1 //@version=5
2 strategy("Cancel demo", "test", overlay = true)
3
4 //@variable Draws a horizontal line at the `limit` price of the "buy" order.
5 var line limitLine = na
6
7 //@variable Returns `color.green` when the strategy places the "buy" order, `color.
8   ↪orange` when it cancels the order.
9 color bgColor = na
10
11 if last_bar_index - bar_index == 100
12     float limitPrice = close - syminfo.mintick * 500
13     strategy.entry("buy", strategy.long, limit = limitPrice)
14     limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend =_
15       ↪extend.right)
16     bgColor := color.new(color.green, 50)
17
18 if last_bar_index - bar_index == 99
19     strategy.cancel("buy")
20     bgColor := color.new(color.orange, 50)
21
22 bgcolor(bgColor)

```

As with `strategy.close()`, the `id` parameter of `strategy.cancel()` refers to the ID of an existing entry. This command will do nothing if the `id` parameter references an ID that doesn't exist. When there are multiple pending orders with the same ID, this command will cancel all of them at once.

In this example, we've modified the previous script to place a "buy" limit order on three consecutive bars starting from 100 bars ago. The strategy cancels all of them after the `bar_index` is 97 bars away from the most recent bar, resulting in it doing nothing when the price crosses any of the lines:



```

1 //@version=5
2 strategy("Multiple cancel demo", "test", overlay = true, pyramiding = 3)
3
4 //@variable Draws a horizontal line at the `limit` price of the "buy" order.
5 var line limitLine = na
6
7 //@variable Returns `color.green` when the strategy places the "buy" order, `color.
8   orange` when it cancels the order.
9 color bgColor = na
10
11 if last_bar_index - bar_index <= 100 and last_bar_index - bar_index >= 98
12   float limitPrice = close - syminfo.mintick * 500
13   strategy.entry("buy", strategy.long, limit = limitPrice)
14   limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend =_
15     extend.right)
16   bgColor := color.new(color.green, 50)
17
18 if last_bar_index - bar_index == 97
19   strategy.cancel("buy")
20   bgColor := color.new(color.orange, 50)
21
22 bgcolor(bgColor)

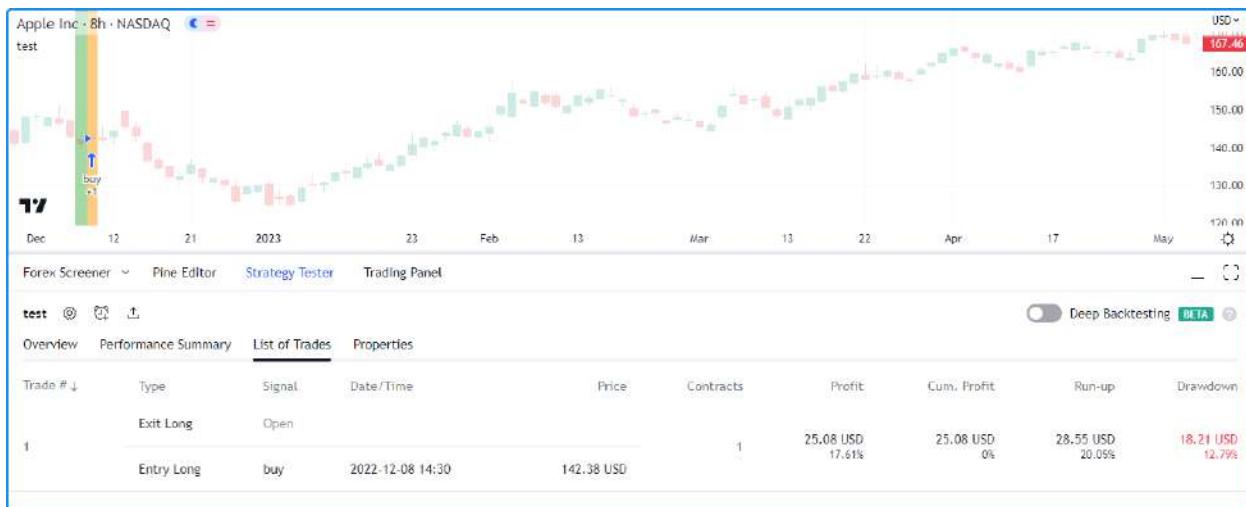
```

#### Note that:

- We added `pyramiding = 3` to the script's declaration statement to allow three `strategy.entry()` orders to fill. Alternatively, the script would achieve the same output by using `strategy.order()` since it isn't sensitive to the `pyramiding` setting.

It's important to note that neither `strategy.cancel()` nor `strategy.cancel_all()` can cancel *market* orders, as the strategy executes them immediately upon the next tick. Strategies cannot cancel orders after they've been filled. To close an open position, use `strategy.close()` or `strategy.close_all()`.

This example simulates a “buy” market order 100 bars ago, then attempts to cancel all pending orders on the next bar. Since the strategy already filled the “buy” order, the `strategy.cancel_all()` command does nothing in this case, as there are no pending orders to cancel:



```

1 //@version=5
2 strategy("Cancel market demo", "test", overlay = true)
3
4 //@variable Returns `color.green` when the strategy places the "buy" order, `color.
5   ↪orange` when it tries to cancel.
6 color bgColor = na
7
8 if last_bar_index - bar_index == 100
9   strategy.entry("buy", strategy.long)
10  bgColor := color.new(color.green, 50)
11
12 if last_bar_index - bar_index == 99
13  strategy.cancel_all()
14  bgColor := color.new(color.orange, 50)
15
bgcolor(bgColor)

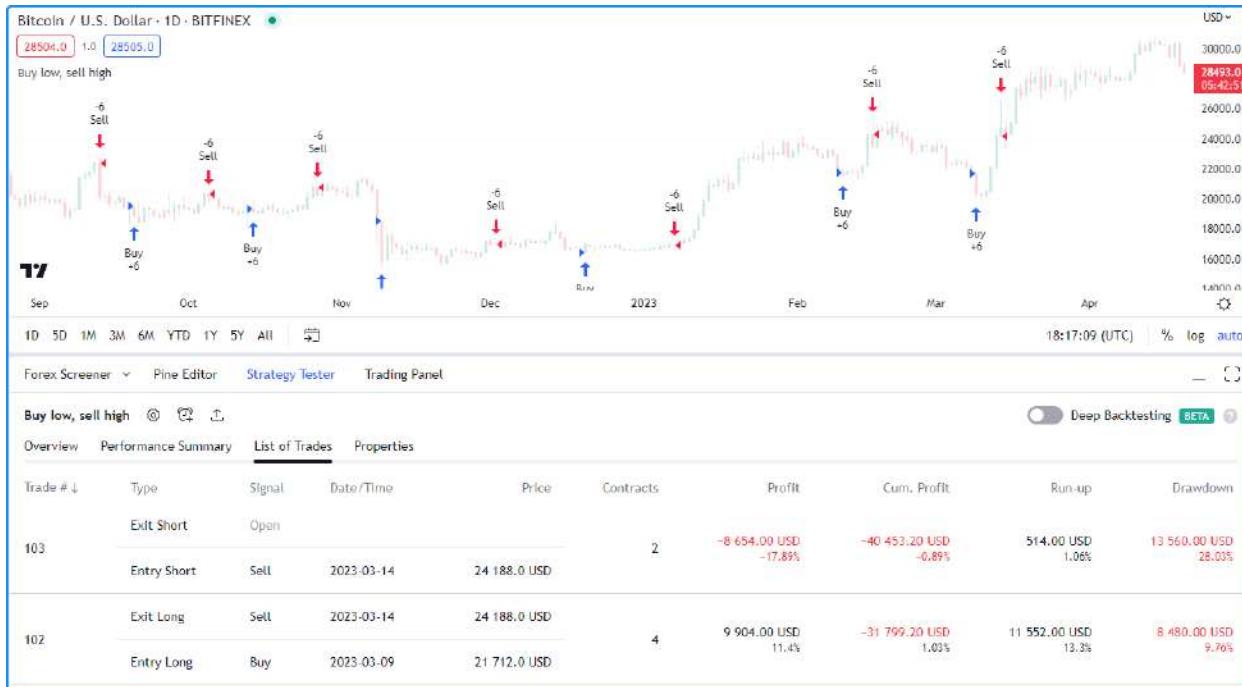
```

### 4.18.7 Position sizing

Pine Script™ strategies feature two ways to control the sizes of simulated trades:

- Set a default fixed quantity type and value for all orders using the `default_qty_type` and `default_qty_value` arguments in the `strategy()` function, which also sets the default values in the “Properties” tab of the script settings.
- Specify the `qty` argument when calling `strategy.entry()`. When a user supplies this argument to the function, the script ignores the strategy’s default quantity value and type.

The following example simulates “Buy” orders of `longAmount` size whenever the `low` price equals the lowest value, and “Sell” orders of `shortAmount` size when the `high` price equals the highest value:



```

1 //@version=5
2 strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash,
3           ↴default_qty_value = 5000)
4
5 int    length      = input.int(20, "Length")
6 float longAmount  = input.float(4.0, "Long Amount")
7 float shortAmount = input.float(2.0, "Short Amount")
8
9 float highest = ta.highest(length)
10 float lowest   = ta.lowest(length)
11
12 switch
13     low == lowest  => strategy.entry("Buy", strategy.long, longAmount)
14     high == highest => strategy.entry("Sell", strategy.short, shortAmount)

```

Notice that in the above example, although we've specified the `default_qty_type` and `default_qty_value` arguments in the declaration statement, the script does not use these defaults for the simulated orders. Instead, it sizes them as a `longAmount` and `shortAmount` of units. If we want the script to use the default type and value, we must remove the `qty` specification from the `strategy.entry()` calls:



```

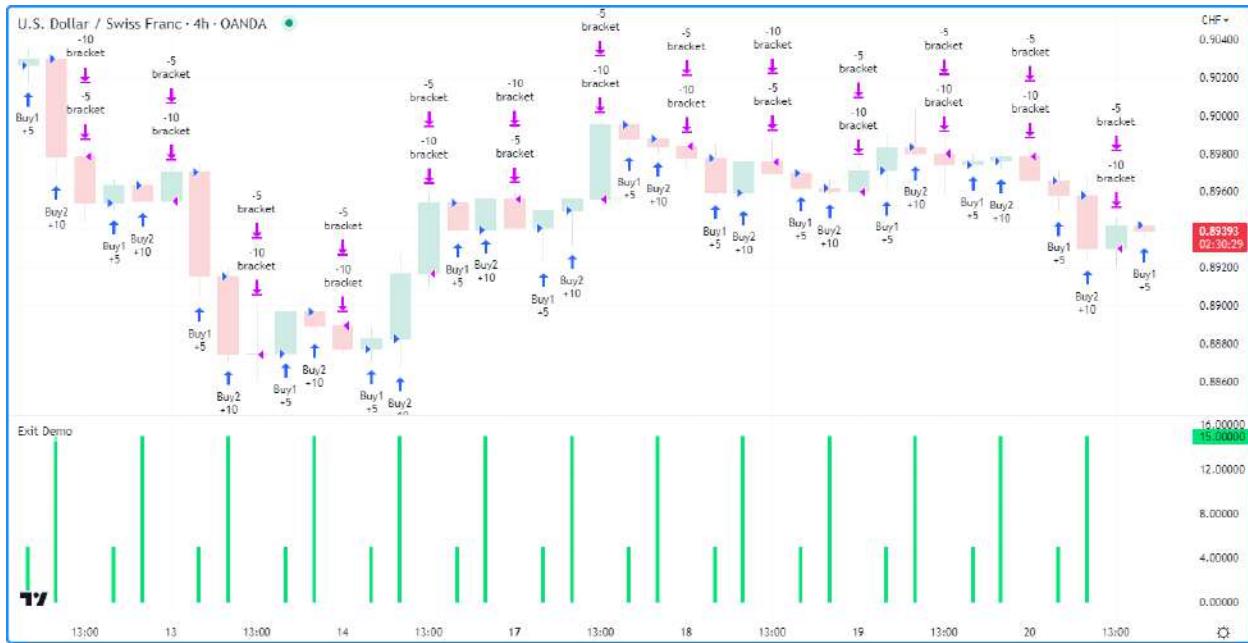
1 //@version=5
2 strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash,
3           ↴default_qty_value = 5000)
4
5 int length = input.int(20, "Length")
6
7 float highest = ta.highest(length)
8 float lowest = ta.lowest(length)
9
10 switch
11     low == lowest => strategy.entry("Buy", strategy.long)
        high == highest => strategy.entry("Sell", strategy.short)

```

### 4.18.8 Closing a market position

Although it is possible to simulate an exit from a specific entry order shown in the List of Trades tab of the *Strategy Tester* module, all orders are linked according to FIFO (first in, first out) rules. If the user does not specify the `from_entry` parameter of a `strategy.exit()` call, the strategy will exit the open market position starting from the first entry order that opened it.

The following example simulates two orders sequentially: “Buy1” at the market price for the last 100 bars and “Buy2” once the position size matches the size of “Buy1”. The strategy only places an exit order when the `positionSize` is 15 units. The script does not supply a `from_entry` argument to the `strategy.exit()` command, so the strategy places exit orders for all open positions each time it calls the function, starting with the first. It plots the `positionSize` in a separate pane for visual reference:



```

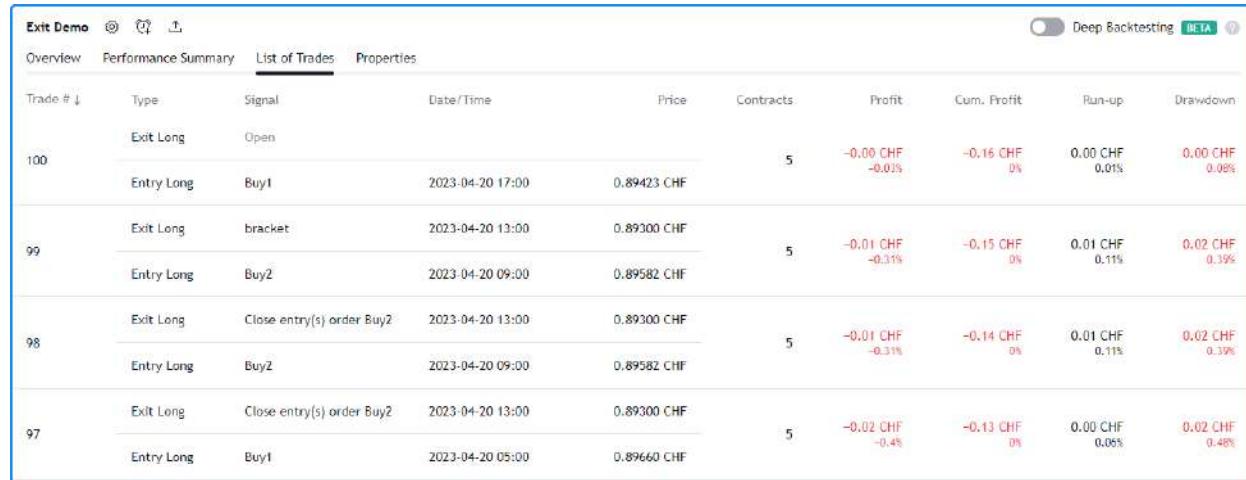
1 //@version=5
2 strategy("Exit Demo", pyramiding = 2)
3
4 float positionSize = strategy.position_size
5
6 if positionSize == 0 and last_bar_index - bar_index <= 100
7     strategy.entry("Buy1", strategy.long, 5)
8 else if positionSize == 5
9     strategy.entry("Buy2", strategy.long, 10)
10 else if positionSize == 15
11     strategy.exit("bracket", loss = 10, profit = 10)
12
13 plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4, plot.
    ↵style_histogram)

```

#### Note that:

- We included `pyramiding = 2` in our script's declaration statement to allow it to simulate two consecutive orders in the same direction.

Suppose we wanted to exit “Buy2” before “Buy1”. Let’s see what happens if we instruct the strategy to close “Buy2” before “Buy1” when it fills both orders:



The screenshot shows a table of trades from a demo account. The columns include Trade #, Type, Signal, Date/Time, Price, Contracts, Profit, Cum. Profit, Run-up, and Drawdown. The trades are as follows:

- Trade 100: Exit Long at Open price 0.89423 CHF.
- Trade 99: Entry Long at Buy1 price 0.89582 CHF.
- Trade 98: Exit Long at bracket price 0.89300 CHF.
- Trade 97: Entry Long at Buy2 price 0.89582 CHF.
- Trade 97: Exit Long at Close entry(s) order Buy2 price 0.89300 CHF.
- Trade 97: Entry Long at Buy1 price 0.89660 CHF.

```

1 //@version=5
2 strategy("Exit Demo", pyramiding = 2)
3
4 float positionSize = strategy.position_size
5
6 if positionSize == 0 and last_bar_index - bar_index <= 100
7     strategy.entry("Buy1", strategy.long, 5)
8 else if positionSize == 5
9     strategy.entry("Buy2", strategy.long, 10)
10 else if positionSize == 15
11     strategy.close("Buy2")
12     strategy.exit("bracket", "Buy1", loss = 10, profit = 10)
13
14 plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4, plot.
    ↪style_histogram)

```

As we can see in the Strategy Tester’s “List of Trades” tab, rather than closing the “Buy2” position with `strategy.close()`, it closes the quantity of “Buy1” first, which is half the quantity of the close order, then closes half of the “Buy2” position, as the broker emulator follows FIFO rules by default. Users can change this behavior by specifying `close_entries_rule = "ANY"` in the `strategy()` function.

### 4.18.9 OCA groups

One-Cancels-All (OCA) groups allow a strategy to fully or partially cancel other orders upon the execution of order placement commands, including `strategy.entry()` and `strategy.order()`, with the same `oca_name`, depending on the `oca_type` that the user provides in the function call.

#### `'strategy.oca.cancel'`

The `strategy.oca.cancel` OCA type cancels all orders with the same `oca_name` upon the fill or partial fill of an order from the group.

For example, the following strategy executes orders upon `ma1` crossing `ma2`. When the `strategy.position_size` is 0, it places long and short stop orders on the `high` and `low` of the bar. Otherwise, it calls `strategy.close_all()` to close all open positions with a market order. Depending on the price action, the strategy may fill both orders before issuing a close order. Additionally, if the broker emulator’s intrabar assumption supports it, both orders may fill on the same bar. The `strategy.close_all()` command does nothing in such cases, as the script cannot invoke the action until after already executing both orders:



```

1 //@version=5
2 strategy("OCA Cancel Demo", overlay=true)
3
4 float ma1 = ta.sma(close, 5)
5 float ma2 = ta.sma(close, 9)
6
7 if ta.cross(ma1, ma2)
8     if strategy.position_size == 0
9         strategy.order("Long", strategy.long, stop = high)
10        strategy.order("Short", strategy.short, stop = low)
11    else
12        strategy.close_all()
13
14 plot(ma1, "Fast MA", color.aqua)
15 plot(ma2, "Slow MA", color.orange)

```

To eliminate scenarios where the strategy fills long and short orders before a close order, we can instruct it to cancel one order after it executes the other. In this example, we've set the `oca_name` for both `strategy.order()` commands to “Entry” and their `oca_type` to `strategy.oca.cancel`:



```

1 //@version=5
2 strategy("OCA Cancel Demo", overlay=true)
3
4 float ma1 = ta.sma(close, 5)
5 float ma2 = ta.sma(close, 9)
6
7 if ta.cross(ma1, ma2)
8     if strategy.position_size == 0
9         strategy.order("Long", strategy.long, stop = high, oca_name = "Entry", oca_
- type = strategy.oca.cancel)

```

(continues on next page)

(continued from previous page)

```

10     strategy.order("Short", strategy.short, stop = low, oca_name = "Entry", oca_
11     ↵type = strategy.oca.cancel)
12     else
13         strategy.close_all()
14
15 plot(ma1, "Fast MA", color.aqua)
plot(ma2, "Slow MA", color.orange)

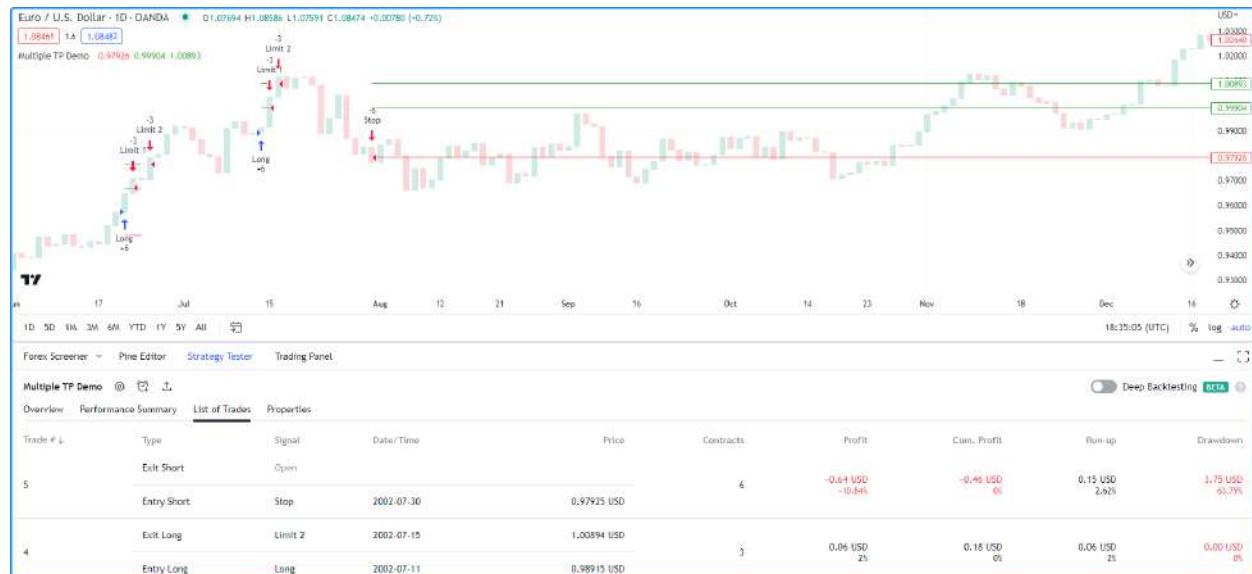
```

### 'strategy.oca.reduce'

The `strategy.oca.reduce` OCA type does not cancel orders. Instead, it reduces the size of orders with the same `oca_name` upon each new fill by the number of closed contracts/shares/lots/units, which is particularly useful for exit strategies.

The following example demonstrates an attempt at a long-only exit strategy that generates a stop-loss order and two take-profit orders for each new entry. Upon the crossover of two moving averages, it simulates a “Long” entry order using `strategy.entry()` with a `qty` of 6 units, then simulates stop/limit orders for 6, 3, and 3 units using `strategy.order()` at the `stop`, `limit1`, and `limit2` prices respectively.

After adding the strategy to our chart, we see it doesn’t work as intended. The issue with this script is that `strategy.order()` doesn’t belong to an OCA group by default, unlike `strategy.exit()`. Since we have not explicitly assigned the orders to an OCA group, the strategy does not cancel or reduce them when it fills one, meaning it’s possible to trade a greater quantity than the open position and reverse the direction:



```

1 //@version=5
2 strategy("Multiple TP Demo", overlay = true)
3
4 var float stop    = na
5 var float limit1 = na
6 var float limit2 = na
7
8 bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9))
9 if longCondition and strategy.position_size == 0
10    stop   := close * 0.99
11    limit1 := close * 1.01
12    limit2 := close * 1.02

```

(continues on next page)

(continued from previous page)

```

13 strategy.entry("Long", strategy.long, 6)
14 strategy.order("Stop", strategy.short, stop = stop, qty = 6)
15 strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3)
16 strategy.order("Limit 2", strategy.short, limit = limit2, qty = 3)
17
18 bool showPlot = strategy.position_size != 0
19 plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr)
20 plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr)
21 plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

```

For our strategy to work as intended, we must instruct it to reduce the number of units for the other stop-loss/take-profit orders so that they do not exceed the size of the remaining open position.

In the example below, we've set the `oca_name` for each order in our exit strategy to "Bracket" and the `oca_type` to `strategy.oca.reduce`. These settings tell the strategy to reduce the `qty` values of orders in the "Bracket" group by the `qty` filled when it executes one of them, preventing it from trading an excessive number of units and causing a reversal:



```

1 //@version=5
2 strategy("Multiple TP Demo", overlay = true)
3
4 var float stop    = na
5 var float limit1 = na
6 var float limit2 = na
7
8 bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9))
9 if longCondition and strategy.position_size == 0
10    stop   := close * 0.99
11    limit1 := close * 1.01
12    limit2 := close * 1.02
13    strategy.entry("Long", strategy.long, 6)
14    strategy.order("Stop", strategy.short, stop = stop, qty = 6, oca_name = "Bracket"
15      ↵, oca_type = strategy.oca.reduce)
16    strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3, oca_name =
17      ↵ "Bracket", oca_type = strategy.oca.reduce)
18    strategy.order("Limit 2", strategy.short, limit = limit2, qty = 6, oca_name =
19      ↵ "Bracket", oca_type = strategy.oca.reduce)
20
21 bool showPlot = strategy.position_size != 0
22 plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr)
23 plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr)
24 plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

```

### Note that:

- We changed the `qty` of the “Limit 2” order to 6 instead of 3 because the strategy will reduce its value by 3 when it fills the “Limit 1” order. Keeping the `qty` value of 3 would cause it to drop to 0 and never fill after filling the first limit order.

### `'strategy.oca.none'`

The `strategy.oca.none` OCA type specifies that an order executes independently of any OCA group. This value is the default `oca_type` for `strategy.order()` and `strategy.entry()` order placement commands.

---

**Note:** If two order placement commands have the same `oca_name` but different `oca_type` values, the strategy considers them to be from two distinct groups. i.e., OCA groups cannot combine `strategy.oca.cancel`, `strategy.oca.reduce`, and `strategy.oca.none` OCA types.

---

### 4.18.10 Currency

Pine Script™ strategies can use different base currencies than the instruments they calculate on. Users can specify the simulated account’s base currency by including a `currency.*` variable as the `currency` argument in the `strategy()` function, which will change the script’s `strategy.account_currency` value. The default `currency` value for strategies is `currency.NONE`, meaning that the script uses the base currency of the instrument on the chart.

When a strategy script uses a specified base currency, it multiplies the simulated profits by the FX\_IDC conversion rate from the previous trading day. For example, the strategy below places an entry order for a standard lot (100,000 units) with a profit target and stop-loss of 1 point on each of the last 500 chart bars, then plots the net profit alongside the inverted daily close of the symbol in a separate pane. We have set the base currency to `currency.EUR`. When we add this script to FX\_IDC:EURUSD, the two plots align, confirming the strategy uses the previous day’s rate from this symbol for its calculations:



```

1 //@version=5
2 strategy("Currency Test", currency = currency.EUR)
3
4 if last_bar_index - bar_index < 500
5     strategy.entry("LE", strategy.long, 100000)
6     strategy.exit("LX", "LE", profit = 1, loss = 1)
7 plot(math.abs(ta.change(strategy.netprofit)), "1 Point profit", color = color.fuchsia,

```

(continues on next page)

(continued from previous page)

```

1 ↵ linewidth = 4)
2 plot(request.security(syminfo.tickerid, "D", 1 / close)[1], "Previous day's inverted_
3 ↵ price", color = color.lime)

```

**Note that:**

- When trading on timeframes higher than daily, the strategy will use the closing price from one trading day before the bar closes for cross-rate calculation on historical bars. For example, on a weekly timeframe, it will base the cross-rate on the previous Thursday's closing value, though the strategy will still use the daily closing rate for real-time bars.

### 4.18.11 Altering calculation behavior

Strategies execute on all historical bars available from a chart, then automatically continue their calculations in real-time as new data is available. By default, strategy scripts only calculate once per confirmed bar. We can alter this behavior by changing the parameters of the `strategy()` function or clicking the checkboxes in the “Recalculate” section of the script’s “Properties” tab.

#### `'calc_on_every_tick'`

`calc_on_every_tick` is an optional setting that controls the calculation behavior on real-time data. When this parameter is enabled, the script will recalculate its values on each new price tick. By default, its value is false, meaning the script only executes calculations after a bar is confirmed.

Enabling this calculation behavior may be particularly useful when forward testing since it facilitates granular, real-time strategy simulation. However, it's important to note that this behavior introduces a data difference between real-time and historical simulations, as historical bars do not contain tick information. Users should exercise caution with this setting, as the data difference may cause a strategy to repaint its history.

The following script will simulate a new order each time that `close` reaches the highest or lowest value over the input length. Since `calc_on_every_tick` is enabled in the strategy declaration, the script will simulate new orders on each new real-time price tick after compilation:

```

1 //@version=5
2 strategy("Donchian Channel Break", overlay = true, calc_on_every_tick = true,_
3 ↵ pyramiding = 20)
4 int length = input.int(15, "Length")
5 float highest = ta.highest(close, length)
6 float lowest = ta.lowest(close, length)
7
8 if close == highest
9     strategy.entry("Buy", strategy.long)
10 if close == lowest
11     strategy.entry("Sell", strategy.short)
12
13 //@variable The starting time for real-time bars.
14 var realTimeStart = timenow
15
16 // Color the background of real-time bars.
17 bgcolor(time_close >= realTimeStart ? color.new(color.orange, 80) : na)
18
19

```

(continues on next page)

(continued from previous page)

```
20 plot(highest, "Highest", color = color.lime)
21 plot(lowest, "Lowest", color = color.red)
```

### Note that:

- The script uses a pyramiding value of 20 in its declaration, which allows the strategy to simulate a maximum of 20 trades in the same direction.
- To visually demarcate what bars are processed as real-time bars by the strategy, the script colors the background for all bars since the `timenow` when it was last compiled.

After applying the script to the chart and letting it calculate on some real-time bars, we may see an output like the following:



The script placed “Buy” orders on each new real-time tick the condition was valid on, resulting in multiple orders per bar. However, it may surprise users unfamiliar with this behavior to see the strategy’s outputs change after recompiling the script, as the bars that it previously executed real-time calculations on are now historical bars, which do not hold tick information:



### `'calc_on_order_fills'`

The optional `calc_on_order_fills` setting enables the recalculation of a strategy immediately after simulating an order fill, which allows the script to use more granular prices and place additional orders without waiting for a bar to be confirmed.

Enabling this setting can provide the script with additional data that would otherwise not be available until after a bar closes, such as the current average price of a simulated position on an unconfirmed bar.

The example below shows a simple strategy declared with `calc_on_order_fills` enabled that simulates a “Buy” order when the `strategy.position_size` is 0. The script uses the `strategy.position_avg_price` to calculate a `stopLoss` and `takeProfit` and simulates “Exit” orders when the price crosses them, regardless of whether the bar is confirmed. As a result, as soon as an exit is triggered, the strategy recalculates and places a new entry order because the `strategy.position_size` is once again equal to 0. The strategy places the order once the exit happens and executes it on the next tick after the exit, which will be one of the bar’s OHLC values, depending on the emulated intrabar movement:



```

1 //@version=5
2 strategy("Intraday exit", overlay = true, calc_on_order_fills = true)
3
4 float stopSize    = input.float(5.0, "SL %", minval = 0.0) / 100.0
5 float profitSize = input.float(5.0, "TP %", minval = 0.0) / 100.0
6
7 if strategy.position_size == 0.0
8     strategy.entry("Buy", strategy.long)
9
10 float stopLoss   = strategy.position_avg_price * (1.0 - stopSize)
11 float takeProfit = strategy.position_avg_price * (1.0 + profitSize)
12
13 strategy.exit("Exit", stop = stopLoss, limit = takeProfit)

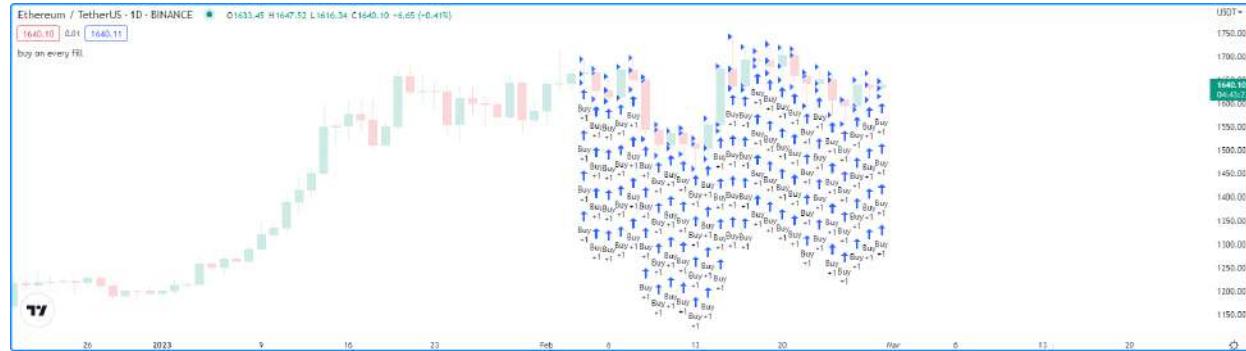
```

### Note that:

- With `calc_on_order_fills` turned off, the same strategy will only ever enter one bar after it triggers an exit order. First, the mid-bar exit will happen, but no entry order. Then, the strategy will simulate an entry order once the bar closes, which it will fill on the next tick after that, i.e., the open of the next bar.

It's important to note that enabling `calc_on_order_fills` may produce unrealistic strategy results, as the broker emulator may assume order prices that are not possible when trading in real-time. Users must exercise caution with this setting and carefully consider the logic in their scripts.

The following example simulates a “Buy” order after each new order fill and bar confirmation over a 25-bar window from the `last_bar_index` when the script loaded on the chart. With the setting enabled, the strategy simulates four entries per bar since the emulator considers each bar to have four ticks (open, high, low, close), which is unrealistic behavior, as it's not typically possible for an order to fill at the exact high or low of a bar:



```

1 //@version=5
2 strategy("buy on every fill", overlay = true, calc_on_order_fills = true, pyramiding=
3     ↵ = 100)
4
5 if last_bar_index - bar_index <= 25
6     strategy.entry("Buy", strategy.long)

```

### **`process\_orders\_on\_close`**

The default strategy behavior simulates orders at the close of each bar, meaning that the earliest opportunity to fill the orders and execute strategy calculations and alerts is upon the opening of the following bar. Traders can change this behavior to process a strategy using the closing value of each bar by enabling the `process_orders_on_close` setting.

This behavior is most useful when backtesting manual strategies in which traders exit positions before a bar closes or in scenarios where algorithmic traders in non-24x7 markets set up after-hours trading capability so that alerts sent after close still have hope of filling before the following day.

#### **Note that:**

- It's crucial to be aware that using strategies with `process_orders_on_close` in a live trading environment may lead to a repainting strategy, as alerts on the close of a bar still occur when the market closes, and orders may not fill until the next market open.

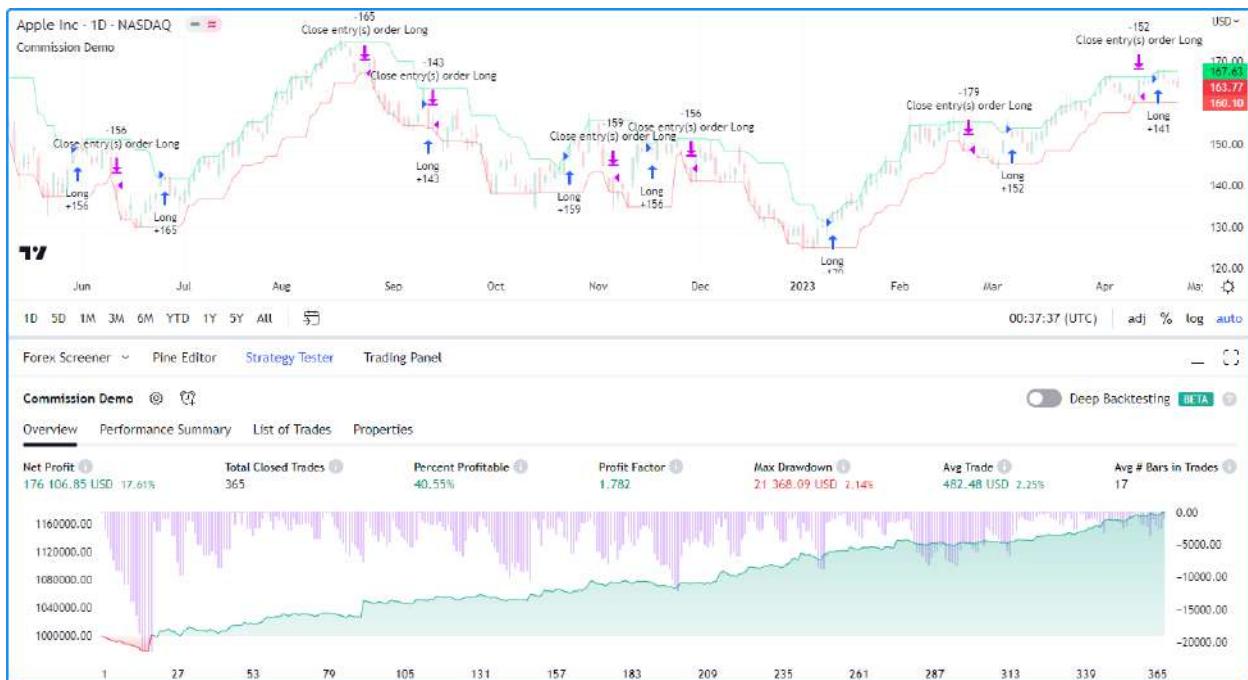
## **4.18.12 Simulating trading costs**

For a strategy performance report to contain relevant, meaningful data, traders should strive to account for potential real-world costs in their strategy results. Neglecting to do so may give traders an unrealistic view of strategy performance and undermine the credibility of test results. Without modeling the potential costs associated with their trades, traders may overestimate a strategy's historical profitability, potentially leading to suboptimal decisions in live trading. Pine Script™ strategies include inputs and parameters for simulating trading costs in performance results.

### **Commission**

Commission refers to the fee a broker/exchange charges when executing trades. Depending on the broker/exchange, some may charge a flat fee per trade or contract/share/lot/unit, and others may charge a percentage of the total transaction value. Users can set the commission properties of their strategies by including `commission_type` and `commission_value` arguments in the `strategy()` function or by setting the “Commission” inputs in the “Properties” tab of the strategy settings.

The following script is a simple strategy that simulates a “Long” position of 2% of equity when `close` equals the highest value over the `length`, and closes the trade when it equals the lowest value:



```

1 //@version=5
2 strategy("Commission Demo", overlay=true, default_qty_value = 2, default_qty_type =_
3   →strategy.percent_of_equity)
4
5 length = input.int(10, "Length")
6
7 float highest = ta.highest(close, length)
8 float lowest = ta.lowest(close, length)
9
10 switch close
11   highest => strategy.entry("Long", strategy.long)
12   lowest => strategy.close("Long")
13
14 plot(highest, color = color.new(color.lime, 50))
15 plot(lowest, color = color.new(color.red, 50))

```

Upon inspecting the results in the Strategy Tester, we see that the strategy had a positive equity growth of 17.61% over the testing range. However, the backtest results do not account for fees the broker/exchange may charge. Let's see what happens to these results when we include a small commission on every trade in the strategy simulation. In this example, we've included `commission_type = strategy.commission.percent` and `commission_value = 1` in the `strategy()` declaration, meaning it will simulate a commission of 1% on all executed orders:



```

1 //@version=5
2 strategy(
3     "Commission Demo", overlay=true, default_qty_value = 2, default_qty_type =_
4     →strategy.percent_of_equity,
5         commission_type = strategy.commission.percent, commission_value = 1
6 )
7
8 length = input.int(10, "Length")
9
10 float highest = ta.highest(close, length)
11 float lowest = ta.lowest(close, length)
12
13 switch close
14     highest => strategy.entry("Long", strategy.long)
15     lowest => strategy.close("Long")
16
17 plot(highest, color = color.new(color.lime, 50))
18 plot(lowest, color = color.new(color.red, 50))

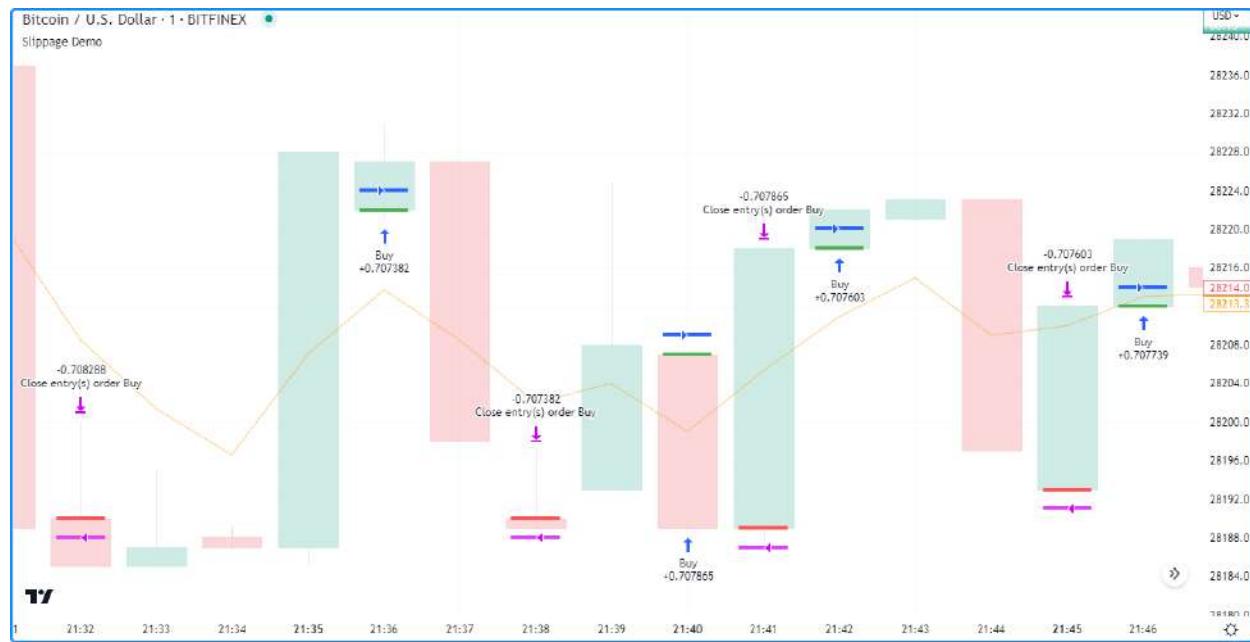
```

As we can see in the example above, after applying a 1% commission to the backtest, the strategy simulated a significantly reduced net profit of only 1.42% and a more volatile equity curve with an elevated max drawdown, highlighting the impact commission simulation can have on a strategy's test results.

## Slippage and unfilled limits

In real-life trading, a broker/exchange may fill orders at slightly different prices than a trader intended due to volatility, liquidity, order size, and other market factors, which can profoundly impact a strategy's performance. The disparity between expected prices and the actual prices at which the broker/exchange executes trades is what we refer to as slippage. Slippage is dynamic and unpredictable, making it impossible to simulate precisely. However, factoring in a small amount of slippage on each trade during a backtest or forward test may help the results better align with reality. Users can model slippage in their strategy results, sized as a fixed number of ticks, by including a `slippage` argument in the `strategy()` declaration or by setting the “Slippage” input in the “Properties” tab of the strategy settings.

The following example demonstrates how slippage simulation affects the fill prices of market orders in a strategy test. The script below places a “Buy” market order of 2% equity when the market price is above an EMA while the EMA is rising and closes the position when the price dips below the EMA while it’s falling. We’ve included `slippage = 20` in the `strategy()` function, which declares that the price of each simulated order will slip 20 ticks in the direction of the trade. The script uses `strategy.opentrades.entry_bar_index()` and `strategy.closedtrades.exit_bar_index()` to get the `entryIndex` and `exitIndex`, which it utilizes to obtain the `fillPrice` of the order. When the bar index is at the `entryIndex`, the `fillPrice` is the first `strategy.opentrades.entry_price()` value. At the `exitIndex`, `fillPrice` is the `strategy.closedtrades.exit_price()` value from the last closed trade. The script plots the expected fill price along with the simulated fill price after slippage to visually compare the difference:



```

1 //@version=5
2 strategy(
3     "Slippage Demo", overlay = true, slippage = 20,
4     default_qty_value = 2, default_qty_type = strategy.percent_of_equity
5 )
6
7 int length = input.int(5, "Length")
8
9 //@variable Exponential moving average with an input `length`.
10 float ma = ta.ema(close, length)
11
12 //@variable Returns `true` when `ma` has increased and `close` is greater than it, ↴
13 // `false` otherwise.
14 bool longCondition = close > ma and ma > ma[1]

```

(continues on next page)

(continued from previous page)

```

14 // @variable Returns `true` when `ma` has decreased and `close` is less than it, ↵
15 // `false` otherwise.
16 bool shortCondition = close < ma and ma < ma[1]
17
18 // Enter a long market position on `longCondition`, close the position on ↵
19 // `shortCondition`.
20 if longCondition
21     strategy.entry("Buy", strategy.long)
22 if shortCondition
23     strategy.close("Buy")
24
25 // @variable The `bar_index` of the position's entry order fill.
26 int entryIndex = strategy.opentrades.entry_bar_index(0)
27 // @variable The `bar_index` of the position's close order fill.
28 int exitIndex = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1)
29
30 // @variable The fill price simulated by the strategy.
31 float fillPrice = switch bar_index
32     entryIndex => strategy.opentrades.entry_price(0)
33     exitIndex  => strategy.closedtrades.exit_price(strategy.closedtrades - 1)
34
35 // @variable The expected fill price of the open market position.
36 float expectedPrice = fillPrice ? open : na
37
38 color expectedColor = na
39 color filledColor = na
40
41 if bar_index == entryIndex
42     expectedColor := color.green
43     filledColor   := color.blue
44 else if bar_index == exitIndex
45     expectedColor := color.red
46     filledColor   := color.fuchsia
47
48 plot(ma, color = color.new(color.orange, 50))
49
50 plotchar(fillPrice ? open : na, "Expected fill price", "-", location.absolute, ↵
51         expectedColor)
52 plotchar(fillPrice, "Fill price after slippage", "-", location.absolute, filledColor)

```

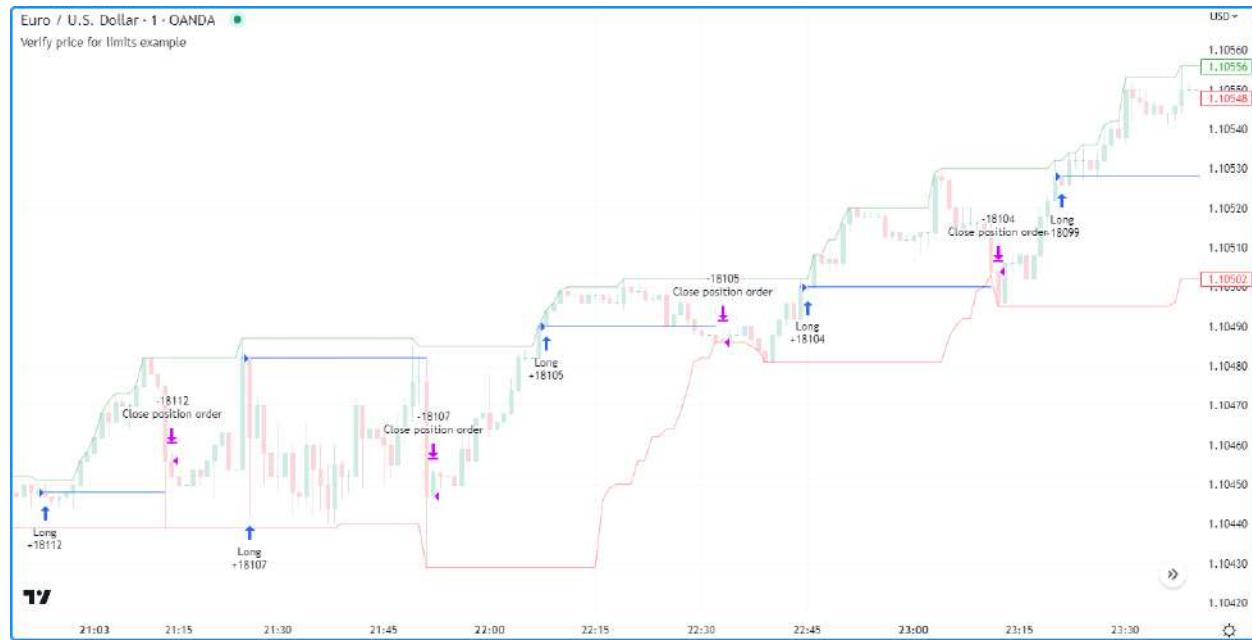
**Note that:**

- Since the strategy applies constant slippage to all order fills, some orders can fill outside the candle range in the simulation. Thus users should exercise caution with this setting, as excessive simulated slippage can produce unrealistically worse testing results.

Some traders may assume that they can avoid the adverse effects of slippage by using limit orders, as unlike market orders, they cannot execute at a worse price than the specified value. However, depending on the state of the real-life market, even if the market price reaches an order price, there's a chance that a limit order may not fill, as limit orders can only fill if a security has sufficient liquidity and price action around the value. To account for the possibility of unfilled orders in a backtest, users can specify the `backtest_fill_limits_assumption` value in the declaration statement or use the “Verify price for limit orders” input in the “Properties” tab to instruct the strategy to fill limit orders only after prices move a defined number of ticks past order prices.

The following example places a limit order of 2% equity at a bar's `h1cc4` when the `high` is the highest value over the past `length` bars and there are no pending entries. The strategy closes the market position and cancels all orders when the `low` is the lowest value. Each time the strategy triggers an order, it draws a horizontal line at the `limitPrice`,

which it updates on each bar until closing the position or canceling the order:



```

1 //@version=5
2 strategy(
3     "Verify price for limits example", overlay = true,
4     default_qty_type = strategy.percent_of_equity, default_qty_value = 2
5 )
6
7 int length = input.int(25, title = "Length")
8
9 // @variable Draws a line at the limit price of the most recent entry order.
10 var line limitLine = na
11
12 // Highest high and lowest low
13 highest = ta.highest(length)
14 lowest = ta.lowest(length)
15
16 // Place an entry order and draw a new line when the the `high` equals the `highest` ↴
17 // value and `limitLine` is `na`.
18 if high == highest and na(limitLine)
19     float limitPrice = hlcc4
20     strategy.entry("Long", strategy.long, limit = limitPrice)
21     limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice)
22
23 // Close the open market position, cancel orders, and set `limitLine` to `na` when ↴
24 // the `low` equals the `lowest` value.
25 if low == lowest
26     strategy.cancel_all()
27     limitLine := na
28     strategy.close_all()
29
30 // Update the `x2` value of `limitLine` if it isn't `na`.
31 if not na(limitLine)
32     limitLine.set_x2(bar_index + 1)

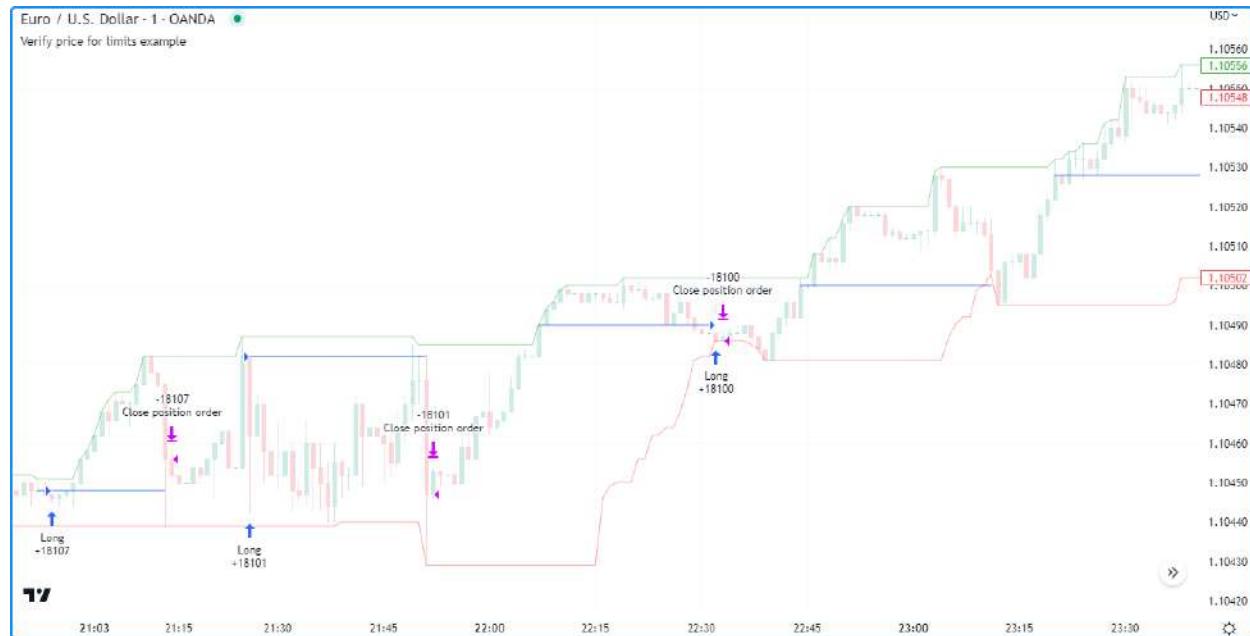
```

(continues on next page)

(continued from previous page)

```
33 plot(lowest, "Lowest Low", color = color.new(color.red, 50))
```

By default, the script assumes that all limit orders are guaranteed to fill. However, this is often not the case in real-life trading. Let's add price verification to our limit orders to account for potentially unfilled ones. In this example, we've included `backtest_fill_limits_assumption = 3` in the `strategy()` function call. As we can see, using limit verification omits some simulated order fills and changes the times of others since the entry orders can now only fill after the price penetrates the limit price by three ticks:



**Note:** It's important to notice that although the limit verification changed the *times* of some order fills, the strategy simulated them at the same *prices*. This “time-warping” effect is a compromise that preserves the prices of verified limit orders, but it can cause the strategy to simulate their fills at times that wouldn't necessarily be possible in the real world. Users should exercise caution with this setting and understand its limitations when analyzing strategy results.

### 4.18.13 Risk management

Designing a strategy that performs well, let alone one that does so in a broad class of markets, is a challenging task. Most are designed for specific market patterns/conditions and may produce uncontrollable losses when applied to other data. Therefore, a strategy's risk management qualities can be critical to its performance. Users can set risk management criteria in their strategy scripts using the special commands with the `strategy.risk` prefix.

Strategies can incorporate any number of risk management criteria in any combination. All risk management commands execute on every tick and order execution event, regardless of any changes to the strategy's calculation behavior. There is no way to disable any of these commands at a script's runtime. Irrespective of the risk rule's location, it will always apply to the strategy unless the user removes the call from the code.

#### `strategy.risk.allow_entry_in()`

This command overrides the market direction allowed for `strategy.entry()` commands. When a user specifies the trade direction with this function (e.g., `strategy.direction.long`), the strategy will only enter trades in that direction. However, it's important to note that if a script calls an entry command in the opposite direction while there's an open market position, the strategy will simulate a market order to exit the position.

**strategy.risk.max\_cons\_loss\_days()**

This command cancels all pending orders, closes the open market position, and stops all additional trade actions after the strategy simulates a defined number of trading days with consecutive losses.

**strategy.risk.max\_drawdown()**

This command cancels all pending orders, closes the open market position, and stops all additional trade actions after the strategy's drawdown reaches the amount specified in the function call.

**strategy.risk.max\_intraday\_filled\_orders()**

This command specifies the maximum number of filled orders per trading day (or per chart bar if the timeframe is higher than daily). Once the strategy executes the maximum number of orders for the day, it cancels all pending orders, closes the open market position, and halts trading activity until the end of the current session.

**strategy.risk.max\_intraday\_loss()**

This command controls the maximum loss the strategy will tolerate per trading day (or per chart bar if the timeframe is higher than daily). When the strategy's losses reach this threshold, it will cancel all pending orders, close the open market position, and stop all trading activity until the end of the current session.

**strategy.risk.max\_position\_size()**

This command specifies the maximum possible position size when using `strategy.entry()` commands. If the quantity of an entry command results in a market position that exceeds this threshold, the strategy will reduce the order quantity so that the resulting position does not exceed the limitation.

## 4.18.14 Margin

Margin is the minimum percentage of a market position a trader must hold in their account as collateral to receive and sustain a loan from their broker to achieve their desired leverage. The `margin_long` and `margin_short` parameters of the `strategy()` declaration and the “Margin for long/short positions” inputs in the “Properties” tab of the script settings allow strategies to specify margin percentages for long and short positions. For example, if a trader sets the margin for long positions to 25%, they must have enough funds to cover 25% of an open long position. This margin percentage also means the trader can potentially spend up to 400% of their equity on their trades.

If a strategy's simulated funds cannot cover the losses from a margin trade, the broker emulator triggers a margin call, which forcibly liquidates all or part of the position. The exact number of contracts/shares/lots/units that the emulator liquidates is four times what is required to cover a loss to prevent constant margin calls on subsequent bars. The emulator calculates the amount using the following algorithm:

1. Calculate the amount of capital spent on the position: `Money Spent = Quantity * Entry Price`
2. Calculate the Market Value of Security (MVS): `MVS = Position Size * Current Price`
3. Calculate the Open Profit as the difference between MVS and Money Spent. If the position is short, we multiply this by -1.
4. Calculate the strategy's equity value: `Equity = Initial Capital + Net Profit + Open Profit`
5. Calculate the margin ratio: `Margin Ratio = Margin Percent / 100`
6. Calculate the margin value, which is the cash required to cover the trader's portion of the position: `Margin = MVS * Margin Ratio`
7. Calculate the available funds: `Available Funds = Equity - Margin`
8. Calculate the total amount of money the trader has lost: `Loss = Available Funds / Margin Ratio`
9. Calculate how many contracts/shares/lots/units the trader would need to liquidate to cover the loss. We truncate this value to the same decimal precision as the minimum position size for the current symbol: `Cover Amount = TRUNCATE(Loss / Current Price)`.
10. Calculate how many units the broker will liquidate to cover the loss: `Margin Call = Cover Amount * 4`

To examine this calculation in detail, let's add the built-in Supertrend Strategy to the NASDAQ:TSLA chart on the 1D timeframe and set the "Order size" to 300% of equity and the "Margin for long positions" to 25% in the "Properties" tab of the strategy settings:



The first entry happened at the bar's opening price on 16 Sep 2010. The strategy bought 682,438 shares (Position size) at 4.43 USD (Entry price). Then, on 23 Sep 2010, when the price dipped to 3.9 (Current price), the emulator forcibly liquidated 111,052 shares via margin call.

```

Money spent: 682438 * 4.43 = 3023200.34
MVS: 682438 * 3.9 = 2661508.2
Open Profit: -361692.14
Equity: 1000000 + 0 - 361692.14 = 638307.86
Margin Ratio: 25 / 100 = 0.25
Margin: 2661508.2 * 0.25 = 665377.05
Available Funds: 638307.86 - 665377.05 = -27069.19
Money Lost: -27069.19 / 0.25 = -108276.76
Cover Amount: TRUNCATE(-108276.76 / 3.9) = TRUNCATE(-27763.27) = -27763
Margin Call Size: -27763 * 4 = - 111052
  
```

## 4.18.15 Strategy Alerts

Regular Pine Script™ indicators have two different mechanisms to set up custom alert conditions: the `alertcondition()` function, which tracks one specific condition per function call, and the `alert()` function, which tracks all its calls simultaneously, but provides greater flexibility in the number of calls, alert messages, etc.

Pine Script™ strategies do not work with `alertcondition()` calls, but they do support the generation of custom alerts via the `alert()` function. Along with this, each function that creates orders also comes with its own built-in alert functionality that does not require any additional code to implement. As such, any strategy that uses an order placement command can issue alerts upon order execution. The precise mechanics of such built-in strategy alerts are described in the Order Fill events section of the [Alerts](#) page in our User Manual.

When a strategy uses functions that create orders and the `alert()` function together, the alert creation dialogue provides a choice between the conditions that it will trigger upon: it can trigger on `alert()` events, order fill events, or both.

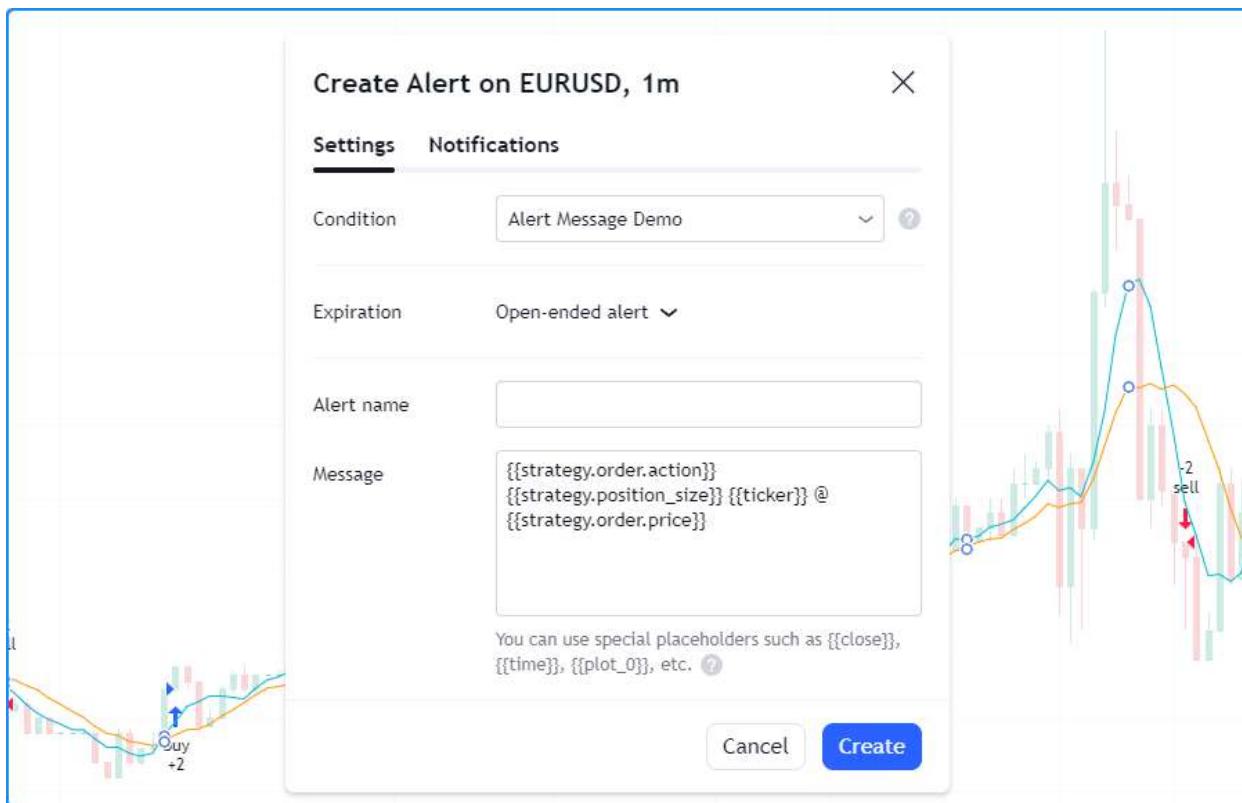
For many trading strategies, the latency between a triggered condition and a live trade can be a critical performance factor. By default, strategy scripts can only execute `alert()` function calls on the close of real-time bars, considering them to use `alert.freq_once_per_bar_close`, regardless of the `freq` argument in the call. Users can change the alert frequency by also including `calc_on_every_tick = true` in the `strategy()` call or selecting the “Recalculate on every tick” option in the “Properties” tab of the strategy settings before creating the alert. However, depending on the script, this may also adversely impact a strategy’s behavior, so exercise caution and be aware of the limitations when using this approach.

When sending alerts to a third party for strategy automation, we recommend using order fill alerts rather than the `alert()` function since they don’t suffer the same limitations; alerts from order fill events execute immediately, unaffected by a script’s `calc_on_every_tick` setting. Users can set the default message for order fill alerts via the `@strategy_alert_message` compiler annotation. The text provided with this annotation will populate the “Message” field for order fills in the alert creation dialogue.

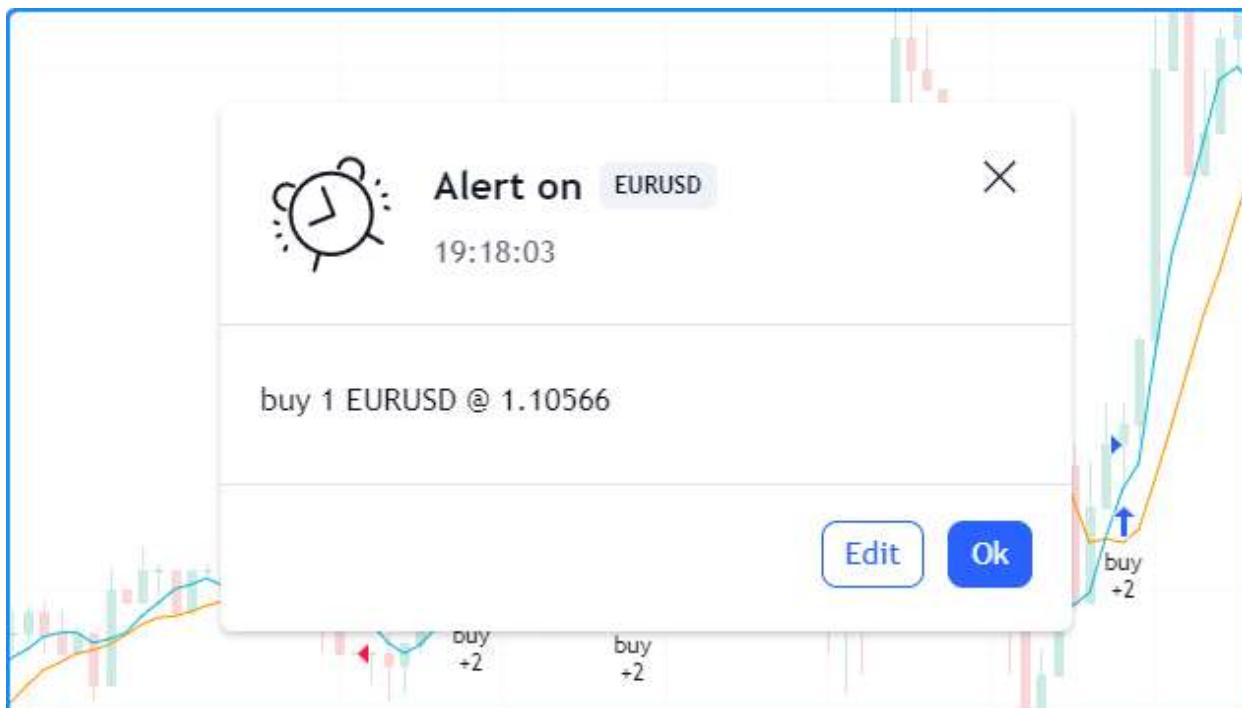
The following script shows a simple example of a default order fill alert message. Above the `strategy()` declaration statement, it uses `@strategy_alert_message` with *placeholders* for the trade action, position size, ticker, and fill price values in the message text:

```
1 // @version=5
2 // @strategy_alert_message {{strategy.order.action}} {{strategy.position_size}} {
3     ↪{{ticker}} @ {{strategy.order.price}}
4 strategy("Alert Message Demo", overlay = true)
5 float fastMa = ta.sma(close, 5)
6 float slowMa = ta.sma(close, 10)
7
8 if ta.crossover(fastMa, slowMa)
9     strategy.entry("buy", strategy.long)
10
11 if ta.crossunder(fastMa, slowMa)
12     strategy.entry("sell", strategy.short)
13
14 plot(fastMa, "Fast MA", color.aqua)
15 plot(slowMa, "Slow MA", color.orange)
```

This script will populate the alert creation dialogue with its default message when the user selects its name from the “Condition” dropdown tab:



Upon the alert trigger, the strategy will populate the placeholders in the alert message with their corresponding values. For example:



## 4.18.16 Notes on testing strategies

It's common for traders to test and tune their strategies in historical and real-time market conditions because many believe that analyzing the results may provide valuable insight into a strategy's characteristics, potential weaknesses, and possibly its future potential. However, traders should always be aware of the biases and limitations of simulated strategy results, especially when using the results to support live trading decisions. This section outlines some caveats associated with strategy validation and tuning and possible solutions to mitigate their effects.

---

**Note:** While testing strategies on existing data may give traders helpful information about a strategy's qualities, it's important to note that neither the past nor the present guarantees the future. Financial markets can change rapidly and unpredictably, which may cause a strategy to sustain uncontrollable losses. Additionally, simulated results may not fully account for other real-world factors that can impact trading performance. Therefore, we recommend that traders thoroughly understand the limitations and risks when evaluating backtests and forward tests and consider them "parts of the whole" in their validation processes rather than basing decisions solely on the results.

---

### Backtesting and forward testing

Backtesting is a technique that traders use to evaluate the historical performance of a trading strategy or model by simulating and analyzing its past results on historical market data; this technique assumes that analysis of a strategy's results on past data may provide insight into its strengths and weaknesses. When backtesting, many traders tweak the parameters of a strategy in an attempt to optimize its results. Analysis and optimization of historical results may help traders to gain a deeper understanding of a strategy. However, traders should always understand the risks and limitations when basing their decisions on optimized backtest results.

Parallel to backtesting, prudent trading system development often also involves incorporating real-time analysis as a tool for evaluating a trading system on a forward-looking basis. Forward testing aims to gauge the performance of a strategy in real-time, real-world market conditions, where factors such as trading costs, slippage, and liquidity can meaningfully affect its performance. Forward testing has the distinct advantage of not being affected by certain types of biases (e.g., lookahead bias or "future data leakage") but carries the disadvantage of being limited in the quantity of data to test. Therefore, it's not typically a standalone solution for strategy validation, but it can provide helpful insights into a strategy's performance in current market conditions.

Backtesting and forward testing are two sides of the same coin, as both approaches aim to validate the effectiveness of a strategy and identify its strengths and weaknesses. By combining backtesting and forward testing, traders may be able to compensate for some limitations and gain a clearer perspective on their strategy's performance. However, it's up to traders to sanitize their strategies and evaluation processes to ensure that insights align with reality as closely as possible.

### Lookahead bias

One typical issue in backtesting some strategies, namely ones that request alternate timeframe data, use repainting variables such as `timenow`, or alter calculation behavior for intrabar order fills, is the leakage of future data into the past during evaluation, which is known as lookahead bias. Not only is this bias a common cause of unrealistic strategy results since the future is never actually knowable beforehand, but it is also one of the typical causes of strategy repainting. Traders can often confirm this bias by forward testing their systems, as lookahead bias does not apply to real-time data where no known data exists beyond the current bar. Users can eliminate this bias in their strategies by ensuring that they don't use repainting variables that leak the future into the past, `request.*()` functions don't include `barmerge.lookahead_on` without offsetting the data series as described on [this](#) section of our page on [repainting](#), and they use realistic calculation behavior.

## Selection bias

Selection bias is a common issue that many traders experience when testing their strategies. It occurs when a trader only analyzes results on specific instruments or timeframes while ignoring others. This bias can result in a distorted perspective of the strategy's robustness, which may impact trading decisions and performance optimizations. Traders can reduce the effects of selection bias by evaluating their strategies on multiple, ideally diverse, symbols and timeframes, making it a point not to ignore poor performance results in their analysis or cherry-pick testing ranges.

## Overfitting

A common pitfall when optimizing a backtest is the potential for overfitting (“curve fitting”), which occurs when the strategy is tailored for specific data and fails to generalize well on new, unseen data. One widely-used approach to help reduce the potential for overfitting and promote better generalization is to split an instrument’s data into two or more parts to test the strategy outside the sample used for optimization, otherwise known as “in-sample” (IS) and “out-of-sample” (OOS) backtesting. In this approach, traders use the IS data for strategy optimization, while the OOS portion is used for testing and evaluating IS-optimized performance on new data without further optimization. While this and other, more robust approaches may provide a glimpse into how a strategy might fare after optimization, traders should exercise caution, as the future is inherently unknowable. No trading strategy can guarantee future performance, regardless of the data used for testing and optimization.



## 4.19 Tables

- *Introduction*
- *Creating tables*
- *Tips*

### 4.19.1 Introduction

Tables are objects that can be used to position information in specific and fixed locations in a script’s visual space. Contrary to all other plots or objects drawn in Pine Script™, tables are not anchored to specific bars; they *float* in a script’s space, whether in overlay or pane mode, in studies or strategies, independently of the chart bars being viewed or the zoom factor used.

Tables contain cells arranged in columns and rows, much like a spreadsheet. They are created and populated in two distinct steps:

1. A table’s structure and key attributes are defined using `table.new()`, which returns a table ID that acts like a pointer to the table, just like label, line, or array IDs do. The `table.new()` call will create the table object but does not display it.

- Once created, and for it to display, the table must be populated using one `table.cell()` call for each cell. Table cells can contain text, or not. This second step is when the width and height of cells are defined.

Most attributes of a previously created table can be changed using `table.set_*`() setter functions. Attributes of previously populated cells can be modified using `table.cell_set_*`() functions.

A table is positioned in an indicator's space by anchoring it to one of nine references: the four corners or midpoints, including the center. Tables are positioned by expanding the table from its anchor, so a table anchored to the `position.middle_right` reference will be drawn by expanding up, down and left from that anchor.

Two modes are available to determine the width/height of table cells:

- A default automatic mode calculates the width/height of cells in a column/row using the widest/highest text in them.
- An explicit mode allows programmers to define the width/height of cells using a percentage of the indicator's available x/y space.

Displayed table contents always represent the last state of the table, as it was drawn on the script's last execution, on the dataset's last bar. Contrary to values displayed in the Data Window or in indicator values, variable contents displayed in tables will thus not change as a script user moves his cursor over specific chart bars. For this reason, it is strongly recommended to always restrict execution of all `table.*()` calls to either the first or last bars of the dataset. Accordingly:

- Use the `var` keyword to declare tables.
- Enclose all other calls inside an `if barstate.islast` block.

**Multiple tables can be used in one script, as long as they are each anchored to a different position. Each table object is identified by its own ID. Limits on the quantity of cells in all tables are determined by the total number of cells used in one script.**

## 4.19.2 Creating tables

When creating a table using `table.new()`, three parameters are mandatory: the table's position and its number of columns and rows. Five other parameters are optional: the table's background color, the color and width of the table's outer frame, and the color and width of the borders around all cells, excluding the outer frame. All table attributes except its number of columns and rows can be modified using setter functions: `table.set_position()`, `table.set_bgcolor()`, `table.set_frame_color()`, `table.set_frame_width()`, `table.set_border_color()` and `table.set_border_width()`.

Tables can be deleted using `table.delete()`, and their content can be selectively removed using `table.clear()`.

When populating cells using `table.cell()`, you must supply an argument for four mandatory parameters: the table id the cell belongs to, its column and row index using indices that start at zero, and the text string the cell contains, which can be null. Seven other parameters are optional: the width and height of the cell, the text's attributes (color, horizontal and vertical alignment, size), and the cell's background color. All cell attributes can be modified using setter functions: `table.cell_set_text()`, `table.cell_set_width()`, `table.cell_set_height()`, `table.cell_set_text_color()`, `table.cell_set_text_halign()`, `table.cell_set_text_valign()`, `table.cell_set_text_size()` and `table.cell_set_bgcolor()`.

Keep in mind that each successive call to `table.cell()` redefines **all** the cell's properties, deleting any properties set by previous `table.cell()` calls on the same cell.

## Placing a single value in a fixed position

Let's create our first table, which will place the value of ATR in the upper-right corner of the chart. We first create a one-cell table, then populate that cell:

```

1 // @version=5
2 indicator("ATR", "", true)
3 // We use `var` to only initialize the table on the first bar.
4 var table atrDisplay = table.new(position.top_right, 1, 1)
5 // We call `ta.atr()` outside the `if` block so it executes on each bar.
6 myAtr = ta.atr(14)
7 if barstate.islast
8     // We only populate the table on the last bar.
9     table.cell(atrDisplay, 0, 0, str.tostring(myAtr))

```



Note that:

- We use the `var` keyword when creating the table with `table.new()`.
- We populate the cell inside an `if barstate.islast` block using `table.cell()`.
- When populating the cell, we do not specify the width or height. The width and height of our cell will thus adjust automatically to the text it contains.
- We call `ta.atr(14)` prior to entry in our `if` block so that it evaluates on each bar. Had we used `str.tostring(ta.atr(14))` inside the `if` block, the function would not have evaluated correctly because it would be called on the dataset's last bar without having calculated the necessary values from the previous bars.

Let's improve the usability and aesthetics of our script:

```

1 // @version=5
2 indicator("ATR", "", true)
3 atrPeriodInput = input.int(14, "ATR period", minval = 1, tooltip = "Using a period of 1 yields True Range.")
4
5 var table atrDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray,
6     frame_width = 2, frame_color = color.black)
7 myAtr = ta.atr(atrPeriodInput)
8 if barstate.islast
9     table.cell(atrDisplay, 0, 0, str.tostring(myAtr, format.mintick), text_color =
10        color.white)

```



Note that:

- We used `table.new()` to define a background color, a frame color and its width.
- When populating the cell with `table.cell()`, we set the text to display in white.
- We pass `format.mintick` as a second argument to the `str.tostring()` function to restrict the precision of ATR to the chart's tick precision.
- We now use an input to allow the script user to specify the period of ATR. The input also includes a tooltip, which the user can see when he hovers over the “i” icon in the script’s “Settings/Inputs” tab.

### Coloring the chart's background

This example uses a one-cell table to color the chart's background on the bull/bear state of RSI:

```

1 // @version=5
2 indicator("Chart background", "", true)
3 bullColorInput = input.color(color.green, 95), "Bull", inline = "1")
4 bearColorInput = input.color(color.red, 95), "Bear", inline = "1")
5 // ----- Function colors chart bg on RSI bull/bear state.
6 colorChartBg(bullColor, bearColor) =>
7     var table bgTable = table.new(position.middle_center, 1, 1)
8     float r = ta.rsi(close, 20)
9     color bgColor = r > 50 ? bullColor : r < 50 ? bearColor : na
10    if barstate.islast
11        table.cell(bgTable, 0, 0, width = 100, height = 100, bgcolor = bgColor)
12
13 colorChartBg(bullColorInput, bearColorInput)

```

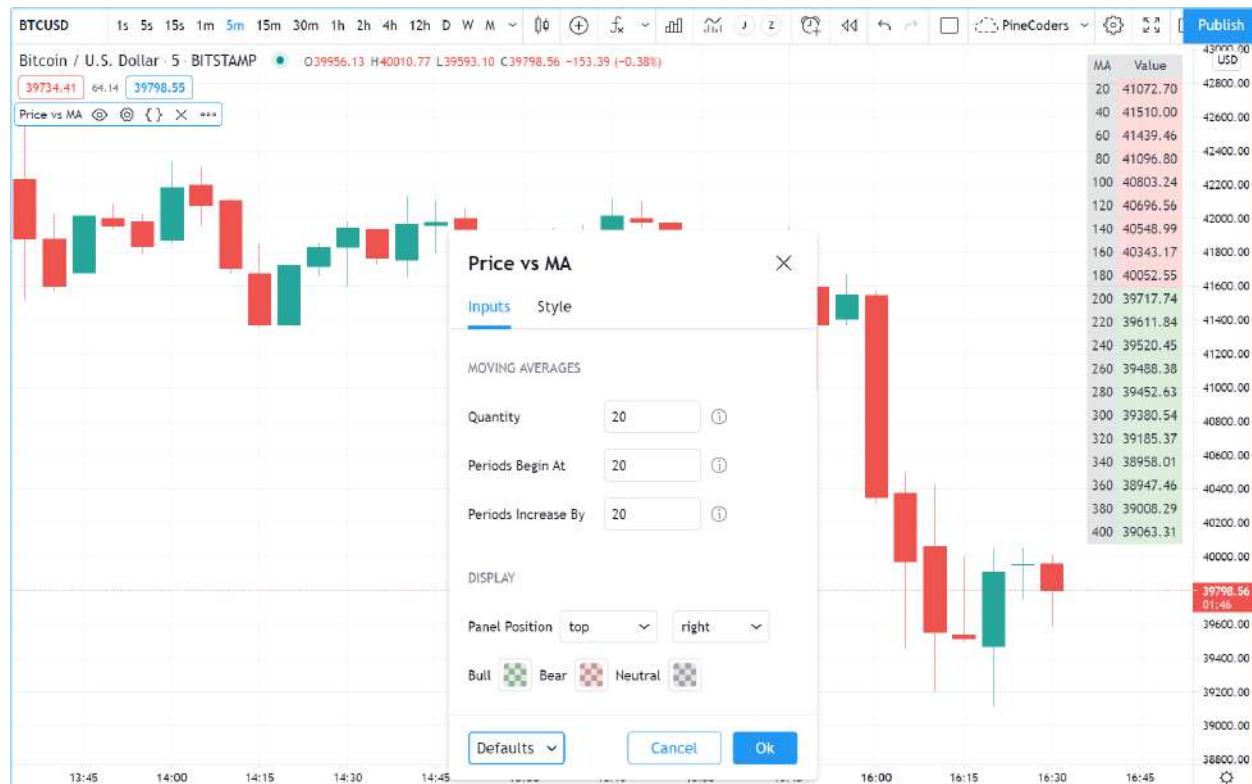
Note that:

- We provide users with inputs allowing them to specify the bull/bear colors to use for the background, and send those input colors as arguments to our `colorChartBg()` function.
- We create a new table only once, using the `var` keyword to declare the table.
- We use `table.cell()` on the last bar only, to specify the cell's properties. We make the cell the width and height of the indicator's space, so it covers the whole chart.

## Creating a display panel

Tables are ideal to create sophisticated display panels. Not only do they make it possible for display panels to always be visible in a constant position, they provide more flexible formatting because each cell's properties are controlled separately: background, text color, size and alignment, etc.

Here, we create a basic display panel showing a user-selected quantity of MAs values. We display their period in the first column, then their value with a green/red/gray background that varies with price's position with regards to each MA. When price is above/below the MA, the cell's background is colored with the bull/bear color. When the MA falls between the current bar's `open` and `close`, the cell's background is of the neutral color:



```

1 //@version=5
2 indicator("Price vs MA", "", true)
3
4 var string GP1 = "Moving averages"
5 int masQtyInput = input.int(20, "Quantity", minval = 1, maxval = 40, group = GP1, tooltip = "1-40")
6 int masStartInput = input.int(20, "Periods begin at", minval = 2, maxval = 200, group = GP1, tooltip = "2-200")
7 int masStepInput = input.int(20, "Periods increase by", minval = 1, maxval = 100, group = GP1, tooltip = "1-100")
8
9 var string GP2 = "Display"
10 string tableYposInput = input.string("top", "Panel position", inline = "11", options = ["top", "middle", "bottom"], group = GP2)
11 string tableXposInput = input.string("right", "", inline = "11", options = ["left", "center", "right"], group = GP2)
12 color bullColorInput = input.color(color.new(color.green, 30), "Bull", inline = "12", group = GP2)
13 color bearColorInput = input.color(color.new(color.red, 30), "Bear", inline = "12", group = GP2)

```

(continues on next page)

(continued from previous page)

```

14     ↵group = GP2)
color    neutColorInput = input.color(color.new(color.gray, 30), "Neutral", inline =
15     ↵"12", group = GP2)
16
17 var table panel = table.new(tableYposInput + "_" + tableXposInput, 2, masQtyInput + 1)
18 if barstate.islast
19     // Table header.
20     table.cell(panel, 0, 0, "MA", bgcolor = neutColorInput)
21     table.cell(panel, 1, 0, "Value", bgcolor = neutColorInput)
22
23 int period = masStartInput
24 for i = 1 to masQtyInput
25     // ----- Call MAs on each bar.
26     float ma = ta.sma(close, period)
27     // ----- Only execute table code on last bar.
28     if barstate.islast
29         // Period in left column.
30         table.cell(panel, 0, i, str.tostring(period), bgcolor = neutColorInput)
31         // If MA is between the open and close, use neutral color. If close is lower/
32     ↵higher than MA, use bull/bear color.
33         bgColor = close > ma ? open < ma ? neutColorInput : bullColorInput : open >
34     ↵ma ? neutColorInput : bearColorInput
35         // MA value in right column.
36         table.cell(panel, 1, i, str.tostring(ma, format.mintick), text_color = color.
37     ↵black, bgcolor = bgColor)
38         period += masStepInput

```

Note that:

- Users can select the table's position from the inputs, as well as the bull/bear/neutral colors to be used for the background of the right column's cells.
- The table's quantity of rows is determined using the number of MAs the user chooses to display. We add one row for the column headers.
- Even though we populate the table cells on the last bar only, we need to execute the calls to `ta.sma()` on every bar so they produce the correct results. The compiler warning that appears when you compile the code can be safely ignored.
- We separate our inputs in two sections using `group`, and join the relevant ones on the same line using `inline`. We supply tooltips to document the limits of certain fields using `tooltip`.

## Displaying a heatmap

Our next project is a heatmap, which will indicate the bull/bear relationship of the current price relative to its past values. To do so, we will use a table positioned at the bottom of the chart. We will display colors only, so our table will contain no text; we will simply color the background of its cells to produce our heatmap. The heatmap uses a user-selectable lookback period. It loops across that period to determine if price is above/below each bar in that past, and displays a progressively lighter intensity of the bull/bear color as we go further in the past:



```

1 // @version=5
2 indicator("Price vs Past", "", true)
3
4 var int MAX_LOOKBACK = 300
5
6 int      lookBackInput = input.int(150, minval = 1, maxval = MAX_LOOKBACK, step = 10)
7 color    bullColorInput = input.color(#00FF00ff, "Bull", inline = "11")
8 color    bearColorInput = input.color(#FF0080ff, "Bear", inline = "11")
9
10 // ----- Function draws a heatmap showing the position of the current `_src` relative
11 // to its past `_lookBack` values.
12 drawHeatmap(src, lookBack) =>
13     // float src      : evaluated price series.
14     // int   lookBack: number of past bars evaluated.
15     // Dependency: MAX_LOOKBACK
16
17     // Force historical buffer to a sufficient size.
18     max_bars_back(src, MAX_LOOKBACK)
19     // Only run table code on last bar.
20     if barstate.islast
21         var heatmap = table.new(position.bottom_center, lookBack, 1)
22         for i = 1 to lookBackInput
23             float transp = 100. * i / lookBack
24             if src > src[i]
25                 table.cell(heatmap, lookBack - i, 0, bgcolor = color.
26                 new(bullColorInput, transp))
27             else
28                 table.cell(heatmap, lookBack - i, 0, bgcolor = color.
29                 new(bearColorInput, transp))
30
31 drawHeatmap(high, lookBackInput)

```

Note that:

- We define a maximum lookback period as a `MAX_LOOKBACK` constant. This is an important value and we use it for two purposes: to specify the number of columns we will create in our one-row table, and to specify the lookback

period required for the `_src` argument in our function, so that we force Pine Script™ to create a historical buffer size that will allow us to refer to the required quantity of past values of `_src` in our `for` loop.

- We offer users the possibility of configuring the bull/bear colors in the inputs and we use `inline` to place the color selections on the same line.
- Inside our function, we enclose our table-creation code in an `if barstate.islast` construct so that it only runs on the last bar of the chart.
- The initialization of the table is done inside the `if` statement. Because of that, and the fact that it uses the `var` keyword, initialization only occurs the first time the script executes on a last bar. Note that this behavior is different from the usual `var` declarations in the script's global scope, where initialization occurs on the first bar of the dataset, at `bar_index` zero.
- We do not specify an argument to the `text` parameter in our `table.cell()` calls, so an empty string is used.
- We calculate our transparency in such a way that the intensity of the colors decreases as we go further in history.
- We use dynamic color generation to create different transparencies of our base colors as needed.
- Contrary to other objects displayed in Pine scripts, this heatmap's cells are not linked to chart bars. The configured lookback period determines how many table cells the heatmap contains, and the heatmap will not change as the chart is panned horizontally, or scaled.
- The maximum number of cells that can be displayed in the script's visual space will depend on your viewing device's resolution and the portion of the display used by your chart. Higher resolution screens and wider windows will allow more table cells to be displayed.

### 4.19.3 Tips

- When creating tables in strategy scripts, keep in mind that unless the strategy uses `calc_on_every_tick = true`, table code enclosed in `if barstate.islast` blocks will not execute on each realtime update, so the table will not display as you expect.
- Keep in mind that successive calls to `table.cell()` overwrite the cell's properties specified by previous `table.cell()` calls. Use the setter functions to modify a cell's properties.
- Remember to control the execution of your table code wisely by restricting it to the necessary bars only. This saves server resources and your charts will display faster, so everybody wins.



## 4.20 Text and shapes

- *Introduction*
- `'plotchar()'`
- `'plotshape()'`
- `'plotarrow()'`
- *Labels*

### 4.20.1 Introduction

You may display text or shapes using five different ways with Pine Script™:

- `plotchar()`
- `plotshape()`
- `plotarrow()`
- Labels created with `label.new()`
- Tables created with `table.new()` (see *Tables*)

Which one to use depends on your needs:

- Tables can display text in various relative positions on charts that will not move as users scroll or zoom the chart horizontally. Their content is not tethered to bars. In contrast, text displayed with `plotchar()`, `plotshape()` or `label.new()` is always tethered to a specific bar, so it will move with the bar's position on the chart. See the page on *Tables* for more information on them.
- Three function include are able to display pre-defined shapes: `plotshape()`, `plotarrow()` and Labels created with `label.new()`.
- `plotarrow()` cannot display text, only up or down arrows.
- `plotchar()` and `plotshape()` can display non-dynamic text on any bar or all bars of the chart.
- `plotchar()` can only display one character while `plotshape()` can display strings, including line breaks.
- `label.new()` can display a maximum of 500 labels on the chart. Its text **can** contain dynamic text, or “series strings”. Line breaks are also supported in label text.
- While `plotchar()` and `plotshape()` can display text at a fixed offset in the past or the future, which cannot change during the script's execution, each `label.new()` call can use a “series” offset that can be calculated on the fly.

These are a few things to keep in mind concerning Pine Script™ strings:

- Since the `text` parameter in both `plotchar()` and `plotshape()` require a “const string” argument, it cannot contain values such as prices that can only be known on the bar (“series string”).
- To include “series” values in text displayed using `label.new()`, they will first need to be converted to strings using `str.tostring()`.
- The concatenation operator for strings in Pine is `+`. It is used to join string components into one string, e.g., `msg = "Chart symbol: " + syminfo.tickerid` (where `syminfo.tickerid` is a built-in variable that returns the chart's exchange and symbol information in string format).
- Characters displayed by all these functions can be Unicode characters, which may include Unicode symbols. See this [Exploring Unicode](#) script to get an idea of what can be done with Unicode characters.

- The color or size of text can sometimes be controlled using function parameters, but no inline formatting (bold, italics, monospace, etc.) is possible.
- Text from Pine scripts always displays on the chart in the Trebuchet MS font, which is used in many TradingView texts, including this one.

This script displays text using the four methods available in Pine Script™:

```

1 //@version=5
2 indicator("Four displays of text", overlay = true)
3 plotchar(ta.rising(close, 5), "`plotchar()", "█", location.belowbar, color.lime,_
4     ↪size = size.small)
5 plotshape(ta.falling(close, 5), "`plotshape()", location = location.abovebar, color =_
6     ↪na, text = ".`plotshape()•`\n█", textcolor = color.fuchsia, size = size.huge)
7
8 if bar_index % 25 == 0
9     label.new(bar_index, na, ".LABEL.\nHigh = " + str.tostring(high, format.mintick)_
10      ↪+ "\n█", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black,_
11      ↪size = size.normal)
12
13 printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t,
14      ↪ 0, 0, txt, bgcolor = color.yellow)
15 printTable("•TABLE•\n" + str.tostring(bar_index + 1) + " bars\nin the dataset")
16

```



Note that:

- The method used to display each text string is shown with the text, except for the lime up arrows displayed using `plotchar()`, as it can only display one character.
- Label and table calls can be inserted in conditional structures to control when their are executed, whereas `plotchar()` and `plotshape()` cannot. Their conditional plotting must be controlled using their first argument, which is a “series bool” whose `true` or `false` value determines when the text is displayed.
- Numeric values displayed in the table and labels is first converted to a string using `str.tostring()`.
- We use the `+` operator to concatenate string components.
- `plotshape()` is designed to display a shape with accompanying text. Its `size` parameter controls the size of the shape, not of the text. We use `na` for its `color` argument so that the shape is not visible.
- Contrary to other texts, the table text will not move as you scroll or scale the chart.
- Some text strings contain the █ Unicode arrow (U+1F807).

- Some text strings contain the \n sequence that represents a new line.

## 4.20.2 `plotchar()`

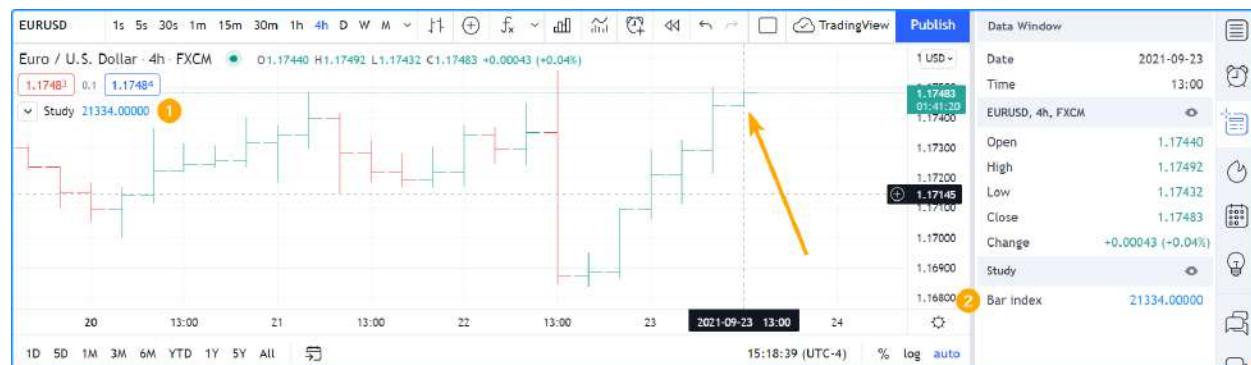
This function is useful to display a single character on bars. It has the following syntax:

```
plotchar(series, title, char, location, color, offset, text, textcolor, editable,
         size, show_last, display) → void
```

See the [Reference Manual](#) entry for `plotchar()` for details on its parameters.

As explained in the [When the script's scale must be preserved](#) section of our page on [Debugging](#), the function can be used to display and inspect values in the Data Window or in the indicator values displayed to the right of the script's name on the chart:

```
1 //@version=5
2 indicator("", "", true)
3 plotchar(bar_index, "Bar index", "", location.top)
```



Note that:

- The cursor is on the chart's last bar.
- The value of `bar_index` on **that** bar is displayed in indicator values (1) and in the Data Window (2).
- We use `location.top` because the default `location.abovebar` will put the price into play in the script's scale, which will often interfere with other plots.

`plotchar()` also works well to identify specific points on the chart or to validate that conditions are `true` when we expect them to be. This example displays an up arrow under bars where `close`, `high` and `volume` have all been rising for two bars:

```
1 //@version=5
2 indicator("", "", true)
3 bool longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.
4   rising(volume, 2))
4 plotchar(longSignal, "Long", "▲", location.belowbar, color = na(volume) ? color.gray :
      color.blue, size = size.tiny)
```



Note that:

- We use `(na(volume) or ta.rising(volume, 2))` so our script will work on symbols without `volume` data. If we did not make provisions for when there is no `volume` data, which is what `na(volume)` does by being `true` when there is no volume, the `longSignal` variable's value would never be `true` because `ta.rising(volume, 2)` yields `false` in those cases.
- We display the arrow in gray when there is no volume, to remind us that all three base conditions are not being met.
- Because `plotchar()` is now displaying a character on the chart, we use `size = size.tiny` to control its size.
- We have adapted the `location` argument to display the character under bars.

If you don't mind plotting only circles, you could also use `plot()` to achieve a similar effect:

```

1 //@version=5
2 indicator("", "", true)
3 longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.
4 ↪rising(volume, 2))
5 plot(longSignal ? low - ta.tr : na, "Long", color.blue, 2, plot.style_circles)

```

This method has the inconvenience that, since there is no relative positioning mechanism with `plot()` one must shift the circles down using something like `ta.tr` (the bar's "True Range"):



### 4.20.3 `plotshape()`

This function is useful to display pre-defined shapes and/or text on bars. It has the following syntax:

```
plotshape(series, title, style, location, color, offset, text, textcolor, editable,_
→size, show_last, display) → void
```

See the [Reference Manual entry for plotshape\(\)](#) for details on its parameters.

Let's use the function to achieve more or less the same result as with our second example of the previous section:

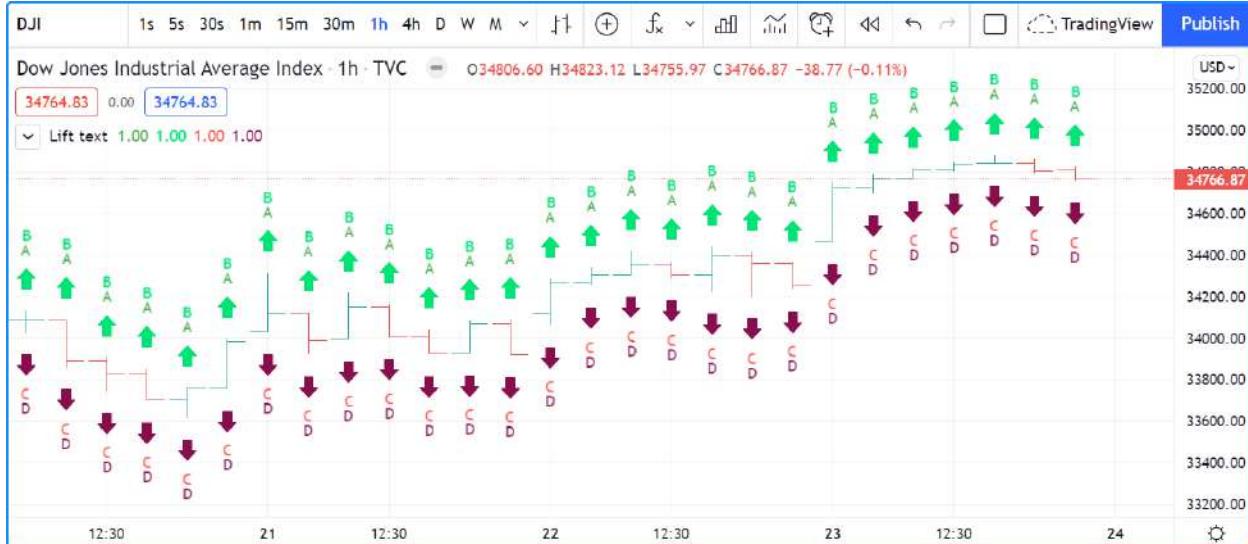
```
1 //@version=5
2 indicator("", "", true)
3 longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.
→rising(volume, 2))
4 plotshape(longSignal, "Long", shape.arrowup, location.belowbar)
```

Note that here, rather than using an arrow character, we are using the `shape.arrowup` argument for the `style` parameter.



It is possible to use different `plotshape()` calls to superimpose text on bars. You will need to use `\n` followed by a special non-printing character that doesn't get stripped out to preserve the newline's functionality. Here we're using a Unicode Zero-width space (`U+200E`). While you don't see it in the following code's strings, it is there and can be copy/pasted. The special Unicode character needs to be the **last** one in the string for text going up, and the **first** one when you are plotting under the bar and text is going down:

```
1 //@version=5
2 indicator("Lift text", "", true)
3 plotshape(true, "", shape.arrowup, location.abovebar, color.green, text = "A")
4 plotshape(true, "", shape.arrowup, location.abovebar, color.lime, text = "B\n\U200E")
5 plotshape(true, "", shape.arrowdown, location.belowbar, color.red, text = "C")
6 plotshape(true, "", shape.arrowdown, location.belowbar, color.maroon, text = "\nD")
```



The available shapes you can use with the `style` parameter are:

Argument	Shape	With Text	Argument	Shape	With Text
shape.xcross			shape.arrowup		
shape.cross			shape.arrowdown		
shape.circle			shape.square		
shape.triangleup			shape.diamond		
shape.triangledown			shape.labelup		
shape.flag			shape.labellow		

#### 4.20.4 `plotarrow()`

The `plotarrow` function displays up or down arrows of variable length, based on the relative value of the series used in the function's first argument. It has the following syntax:

```
plotarrow(series, title, colorup, colordown, offset, minheight, maxheight, editable,  
→ show_last, display) → void
```

See the [Reference Manual](#) entry for `plotarrow()` for details on its parameters.

The `series` parameter in `plotarrow()` is not a “series bool” as in `plotchar()` and `plotshape()`; it is a “series int/float” and there’s more to it than a simple `true` or `false` value determining when the arrows are plotted. This is the logic governing how the argument supplied to `series` affects the behavior of `plotarrow()`:

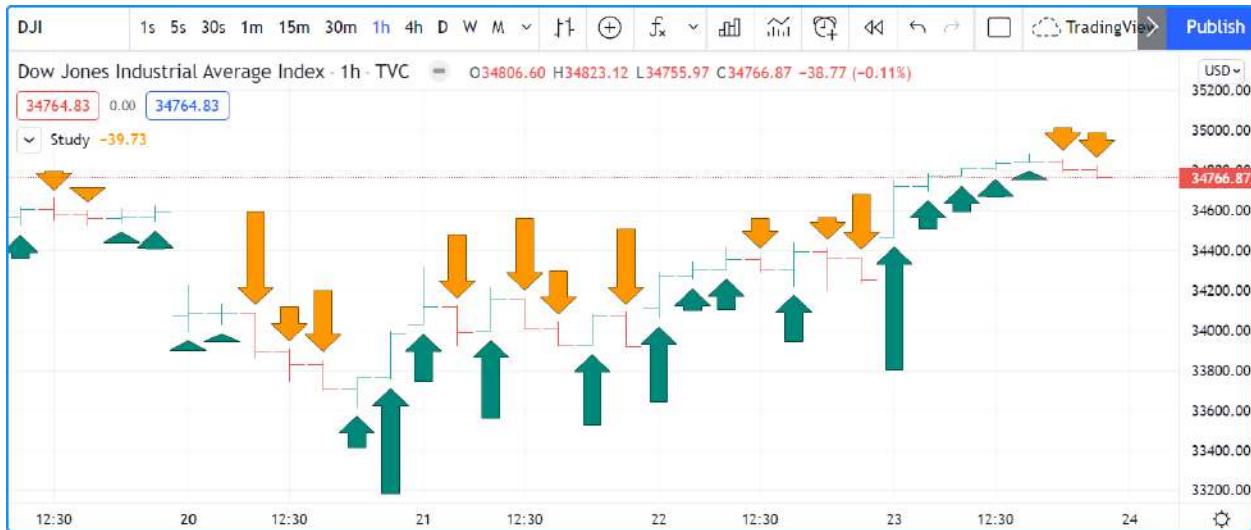
- `series > 0`: An up arrow is displayed, the length of which will be proportional to the relative value of the series on that bar in relation to other series values.

- `series < 0`: A down arrow is displayed, proportionally-sized using the same rules.
- `series == 0` or `na(series)`: No arrow is displayed.

The maximum and minimum possible sizes for the arrows (in pixels) can be controlled using the `minheight` and `maxheight` parameters.

Here is a simple script illustrating how `plotarrow()` works:

```
1 //@version=5
2 indicator("", "", true)
3 body = close - open
4 plotarrow(body, colorup = color.teal, colordown = color.orange)
```



Note how the height of arrows is proportional to the relative size of the bar bodies.

You can use any series to plot the arrows. Here we use the value of the “Chaikin Oscillator” to control the location and size of the arrows:

```
1 //@version=5
2 indicator("Chaikin Oscillator Arrows", overlay = true)
3 fastLengthInput = input.int(3, minval = 1)
4 slowLengthInput = input.int(10, minval = 1)
5 osc = ta.ema(ta.accdist, fastLengthInput) - ta.ema(ta.accdist, slowLengthInput)
6 plotarrow(osc)
```



Note that we display the actual “Chaikin Oscillator” in a pane below the chart, so you can see what values are used to determine the position and size of the arrows.

#### 4.20.5 Labels

Labels are only available in v4 and higher versions of Pine Script™. They work very differently than `plotchar()` and `plotshape()`.

Labels are objects, like [lines and boxes](#), or [tables](#). Like them, they are referred to using an ID, which acts like a pointer. Label IDs are of “label” type. As with other objects, labels IDs are “time series” and all the functions used to manage them accept “series” arguments, which makes them very flexible.

---

**Note:** On TradingView charts, a complete set of *Drawing Tools* allows users to create and modify drawings using mouse actions. While they may sometimes look similar to drawing objects created with Pine Script™ code, they are unrelated entities. Drawing objects created using Pine code cannot be modified with mouse actions, and hand-drawn drawings from the chart user interface are not visible from Pine scripts.

---

Labels are advantageous because:

- They allow “series” values to be converted to text and placed on charts. This means they are ideal to display values that cannot be known before time, such as price values, support and resistance levels, of any other values that your script calculates.
- Their positioning options are more flexible than those of the `plot*()` functions.
- They offer more display modes.
- Contrary to `plot*()` functions, label-handling functions can be inserted in conditional or loop structures, making it easier to control their behavior.
- You can add tooltips to labels.

One drawback to using labels versus `plotchar()` and `plotshape()` is that you can only draw a limited quantity of them on the chart. The default is ~50, but you can use the `max_labels_count` parameter in your `indicator()` or `strategy()` declaration statement to specify up to 500. Labels, like [lines and boxes](#), are managed using a garbage collection mechanism which deletes the oldest ones on the chart, such that only the most recently drawn labels are visible.

Your toolbox of built-ins to manage labels are all in the `label` namespace. They include:

- `label.new()` to create labels.
- `label.set_*` () functions to modify the properties of an existing label.
- `label.get_*` () functions to read the properties of an existing label.
- `label.delete()` to delete labels
- The `label.all` array which always contains the IDs of all the visible labels on the chart. The array's size will depend on the maximum label count for your script and how many of those you have drawn. `array.size(label.all)` will return the array's size.

### Creating and modifying labels

The `label.new()` function creates a new label. It has the following signature:

```
label.new(x, y, text, xloc, yloc, color, style, textcolor, size, textalign, tooltip)  
↳ series label
```

The *setter* functions allowing you to change a label's properties are:

- `label.set_x()`
- `label.set_y()`
- `label.set_xy()`
- `label.set_text()`
- `label.set_xloc()`
- `label.set_yloc()`
- `label.set_color()`
- `label.set_style()`
- `label.set_textcolor()`
- `label.set_size()`
- `label.set_textalign()`
- `label.set_tooltip()`

They all have a similar signature. The one for `label.set_color()` is:

```
label.set_color(id, color) → void
```

where:

- `id` is the ID of the label whose property is to be modified.
- The next parameter is the property of the label to modify. It depends on the setter function used. `label.set_xy()` changes two properties, so it has two such parameters.

This is how you can create labels in their simplest form:

```
1 // @version=5  
2 indicator("", "", true)  
3 label.new(bar_index, high)
```



Note that:

- The label is created with the parameters `x = bar_index` (the index of the current bar, `bar_index`) and `y = high` (the bar's `high` value).
- We do not supply an argument for the function's `text` parameter. Its default value being an empty string, no text is displayed.
- No logic controls our `label.new()` call, so labels are created on every bar.
- Only the last 54 labels are displayed because our `indicator()` call does not use the `max_labels_count` parameter to specify a value other than the ~50 default.
- Labels persist on bars until your script deletes them using `label.delete()`, or garbage collection removes them.

In the next example we display a label on the bar with the highest `high` value in the last 50 bars:

```

1 // @version=5
2 indicator("", "", true)
3
4 // Find the highest `high` in last 50 bars and its offset. Change it's sign so it is
5 // positive.
6 LOOKBACK = 50
7 hi = ta.highest(LOOKBACK)
8 highestBarOffset = -ta.highestbars(LOOKBACK)
9
10 // Create label on bar zero only.
11 var lbl = label.new(na, na, "", color = color.orange, style = label.style_label_lower_
12 // left)
13 // When a new high is found, move the label there and update its text and tooltip.
14 if ta.change(hi)
15     // Build label and tooltip strings.
16     labelText = "High: " + str.tostring(hi, format.mintick)
17     tooltipText = "Offset in bars: " + str.tostring(highestBarOffset) + "\nLow: " +
18 // str.tostring(low[highestBarOffset], format.mintick)
19     // Update the label's position, text and tooltip.
20     label.set_xy(lbl, bar_index[highestBarOffset], hi)
21     label.set_text(lbl, labelText)
22     label.set_tooltip(lbl, tooltipText)

```



Note that:

- We create the label on the first bar only by using the `var` keyword to declare the `lbl` variable that contains the label's ID. The `x`, `y` and `text` arguments in that `label.new()` call are irrelevant, as the label will be updated on further bars. We do, however, take care to use the `color` and `style` we want for the labels, so they don't need updating later.
- On every bar, we detect if a new high was found by testing for changes in the value of `hi`
- When a change in the high value occurs, we update our label with new information. To do this, we use three `label.set*` calls to change the label's relevant information. We refer to our label using the `lbl` variable, which contains our label's ID. The script is thus maintaining the same label throughout all bars, but moving it and updating its information when a new high is detected.

Here we create a label on each bar, but we set its properties conditionally, depending on the bar's polarity:

```

1 //@version=5
2 indicator("", "", true)
3 lbl = label.new(bar_index, na)
4 if close >= open
5     label.set_text(lbl, "green")
6     label.set_color(lbl, color.green)
7     label.set_yloc(lbl, yloc.belowbar)
8     label.set_style(lbl, label.style_label_up)
9 else
10    label.set_text(lbl, "red")
11    label.set_color(lbl, color.red)
12    label.set_yloc(lbl, yloc.abovebar)
13    label.set_style(lbl, label.style_label_down)

```



## Positioning labels

Labels are positioned on the chart according to `x` (bars) and `y` (price) coordinates. Five parameters affect this behavior: `x`, `y`, `xloc`, `yloc` and `style`:

### `x`

Is either a bar index or a time value. When a bar index is used, the value can be offset in the past or in the future (maximum of 500 bars in the future). Past or future offsets can also be calculated when using time values. The `x` value of an existing label can be modified using `label.set_x()` or `label.set_xy()`.

### `xloc`

Is either `xloc.bar_index` (the default) or `xloc.bar_time`. It determines which type of argument must be used with `x`. With `xloc.bar_index`, `x` must be an absolute bar index. With `xloc.bar_time`, `x` must be a UNIX time in milliseconds corresponding to the `time` value of a bar's `open`. The `xloc` value of an existing label can be modified using `label.set_xloc()`.

### `y`

Is the price level where the label is positioned. It is only taken into account with the default `yloc` value of `yloc.price`. If `yloc` is `yloc.abovebar` or `yloc.belowbar` then the `y` argument is ignored. The `y` value of an existing label can be modified using `label.set_y()` or `label.set_xy()`.

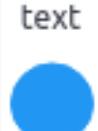
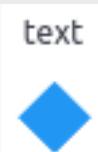
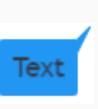
### `yloc`

Can be `yloc.price` (the default), `yloc.abovebar` or `yloc.belowbar`. The argument used for `y` is only taken into account with `yloc.price`. The `yloc` value of an existing label can be modified using `label.set_yloc()`.

### `style`

The argument used has an impact on the visual appearance of the label and on its position relative to the reference point determined by either the `y` value or the top/bottom of the bar when `yloc.abovebar` or `yloc.belowbar` are used. The `style` of an existing label can be modified using `label.set_style()`.

These are the available `style` arguments:

Argument	Label	Label with text	Argument	Label	Label with text
label.style_xcross			label.style_label_up		
label.style_cross			label.style_label_down		
label.style_flag			label.style_label_left		
label.style_circle			label.style_label_right		
label.style_square			label.style_label_lower_left		
label.style_diamond			label.style_label_lower_right		
label.style_triangleup			label.style_label_upper_left		
label.style_triangledown			label.style_label_upper_right		
label.style_arrowup			label.style_label_center		

When using `xloc.bar_time`, the `x` value must be a UNIX timestamp in milliseconds. See the page on [Time](#) for more information. The start time of the current bar can be obtained from the `time` built-in variable. The bar time of previous bars is `time[1]`, `time[2]` and so on. Time can also be set to an absolute value with the `timestamp` function. You may add or subtract periods of time to achieve relative time offset.

Let's position a label one day ago from the date on the last bar:

```

1 // @version=5
2 indicator("")
3 daysAgoInput = input.int(1, tooltip = "Use negative values to offset in the future")
4 if barstate.islast
5     MS_IN_ONE_DAY = 24 * 60 * 60 * 1000
6     oneDayAgo = time - (daysAgoInput * MS_IN_ONE_DAY)
7     label.new(oneDayAgo, high, xloc = xloc.bar_time, style = label.style_label_right)

```

Note that because of varying time gaps and missing bars when markets are closed, the positioning of the label may not always be exact. Time offsets of the sort tend to be more reliable on 24x7 markets.

You can also offset using a bar index for the `x` value, e.g.:

```

label.new(bar_index + 10, high)
label.new(bar_index - 10, high[10])
label.new(bar_index[10], high[10])

```

## Reading label properties

The following *getter* functions are available for labels:

- `label.get_x()`
- `label.get_y()`
- `label.get_text()`

They all have a similar signature. The one for `label.get_text()` is:

```
label.get_text(id) → series string
```

where `id` is the label whose text is to be retrieved.

## Cloning labels

The `label.copy()` function is used to clone labels. Its syntax is:

```
label.copy(id) → void
```

## Deleting labels

The `label.delete()` function is used to delete labels. Its syntax is:

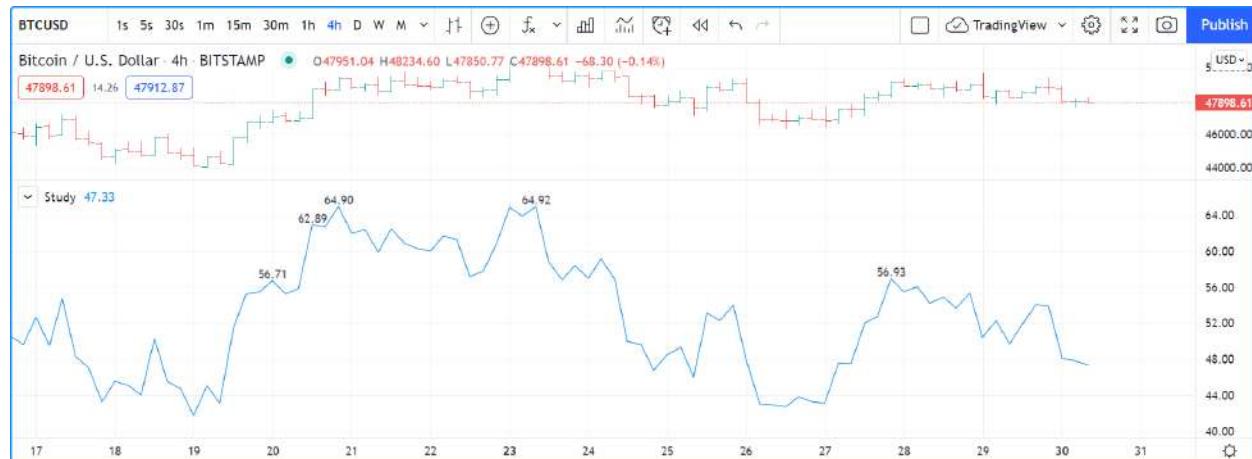
```
label.delete(id) → void
```

To keep only a user-defined quantity of labels on the chart, one could use code like this:

```

1 // @version=5
2 MAX_LABELS = 500
3 indicator("", max_labels_count = MAX_LABELS)
4 qtyLabelsInput = input.int(5, "Labels to keep", minval = 0, maxval = MAX_LABELS)
5 myRSI = ta.rsi(close, 20)
6 if myRSI > ta.highest(myRSI, 20) [1]
7     label.new(bar_index, myRSI, str.tostring(myRSI, "#.00"), style = label.style_none)
8     if array.size(label.all) > qtyLabelsInput
9         label.delete(array.get(label.all, 0))
10 plot(myRSI)

```



Note that:

- We define a `MAX_LABELS` constant to hold the maximum quantity of labels a script can accommodate. We use that value to set the `max_labels_count` parameter's value in our `indicator()` call, and also as the `maxval` value in our `input.int()` call to cap the user value.
- We create a new label when our RSI breaches its highest value of the last 20 bars. Note the offset of `[1]` we use in `if myRSI > ta.highest(myRSI, 20) [1]`. This is necessary. Without it, the value returned by `ta.highest()` would always include the current value of `myRSI`, so `myRSI` would never be higher than the function's return value.
- After that, we delete the oldest label in the `label.all` array that is automatically maintained by the Pine Script™ runtime and contains the ID of all the visible labels drawn by our script. We use the `array.get()` function to retrieve the array element at index zero (the oldest visible label ID). We then use `label.delete()` to delete the label linked with that ID.

Note that if one wants to position a label on the last bar only, it is unnecessary and inefficient to create and delete the label as the script executes on all bars, so that only the last label remains:

```

1 // INEFFICIENT!
2 // @version=5
3 indicator("", "", true)
4 lbl = label.new(bar_index, high, str.tostring(high, format.mintick))
5 label.delete(lbl[1])

```

This is the efficient way to realize the same task:

```

1 // @version=5
2 indicator("", "", true)
3 if barstate.islast
4     // Create the label once, the first time the block executes on the last bar.

```

(continues on next page)

(continued from previous page)

```

5  var lbl = label.new(na, na)
6  // On all iterations of the script on the last bar, update the label's
7  // information.
8  label.set_xy(lbl, bar_index, high)
9  label.set_text(lbl, str.tostring(high, format.mintick))

```

## Realtime behavior

Labels are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes in the realtime bar. See the page on Pine Script™'s Execution model.

This script demonstrates the effect of rollback when running in the realtime bar:

```

1 // @version=5
2 indicator("", "", true)
3 label.new(bar_index, high)

```

On realtime bars, `label.new()` creates a new label on every script update, but because of the rollback process, the label created on the previous update on the same bar is deleted. Only the last label created before the realtime bar's close will be committed, and thus persist.



## 4.21 Time

- *Introduction*
- *Time variables*
- *Time functions*
- *Formatting dates and time*

## 4.21.1 Introduction

### Four references

Four different references come into play when using date and time values in Pine Script™:

1. **UTC time zone:** The native format for time values in Pine Script™ is the **Unix time in milliseconds**. Unix time is the time elapsed since the **Unix Epoch on January 1st, 1970**. See here for the current [Unix time in seconds](#) and here for more information on [Unix Time](#). A value for the Unix time is called a *timestamp*. Unix timestamps are always expressed in the UTC (or “GMT”, or “GMT+0”) time zone. They are measured from a fixed reference, i.e., the Unix Epoch, and do not vary with time zones. Some built-ins use the UTC time zone as a reference.
2. **Exchange time zone:** A second time-related key reference for traders is the time zone of the exchange where an instrument is traded. Some built-ins like `hour` return values in the exchange’s time zone by default.
3. `timezone` parameter: Some functions that normally return values in the exchange’s time zone, such as `hour()` include a `timezone` parameter that allows you to adapt the function’s result to another time zone. Other functions like `time()` include both `session` and `timezone` parameters. In those cases, the `timezone` argument applies to how the `session` argument is interpreted — not to the time value returned by the function.
4. **Chart’s time zone:** This is the time zone chosen by the user from the chart using the “Chart Settings/Symbol/Time Zone” field. This setting only affects the display of dates and times on the chart. It does not affect the behavior of Pine scripts, and they have no visibility over this setting.

When discussing variables or functions, we will note if they return dates or times in UTC or exchange time zone. Scripts do not have visibility on the user’s time zone setting on his chart.

### Time built-ins

Pine Script™ has built-in **variables** to:

- Get timestamp information from the current bar (UTC time zone): `time` and `time_close`
- Get timestamp information for the beginning of the current trading day (UTC time zone): `time_tradingday`
- Get the current time in one-second increments (UTC time zone): `timenow`
- Retrieve calendar and time values from the bar (exchange time zone): `year`, `month`, `weekofyear`, `dayofmonth`, `dayofweek`, `hour`, `minute` and `second`
- Return the time zone of the exchange of the chart’s symbol with `syminfo.timezone`

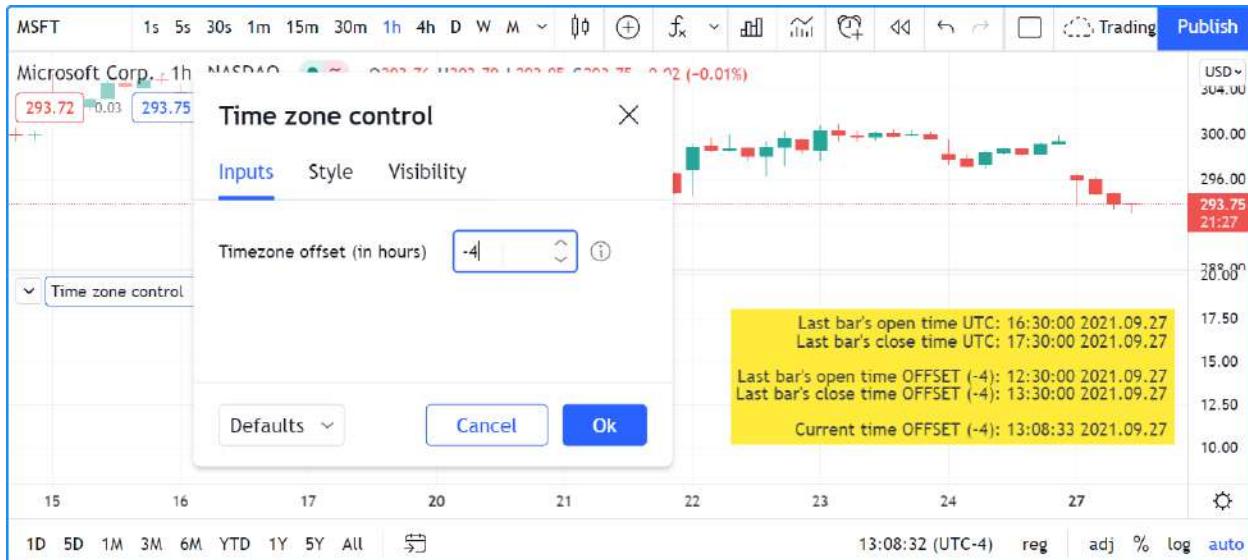
There are also built-in **functions** that can:

- Return timestamps of bars from other timeframes with `time()` and `time_close()`, without the need for a `request.security()` call
- Retrieve calendar and time values from any timestamp, which can be offset with a time zone: `year()`, `month()`, `weekofyear()`, `dayofmonth()`, `dayofweek()`, `hour()`, `minute()` and `second()`
- Create a timestamp using `timestamp()`
- Convert a timestamp to a formatted date/time string for display, using `str.format()`
- Input data and time values. See the section on [Inputs](#).
- Work with [session information](#).

## Time zones

TradingViewers can change the time zone used to display bar times on their charts. Pine scripts have no visibility over this setting. While there is a `syminfo.timezone` variable to return the time zone of the exchange where the chart's instrument is traded, there is **no** `chart.timezone` equivalent.

When displaying times on the chart, this shows one way of providing users a way of adjusting your script's time values to those of their chart. This way, your displayed times can match the time zone used by traders on their chart:



```

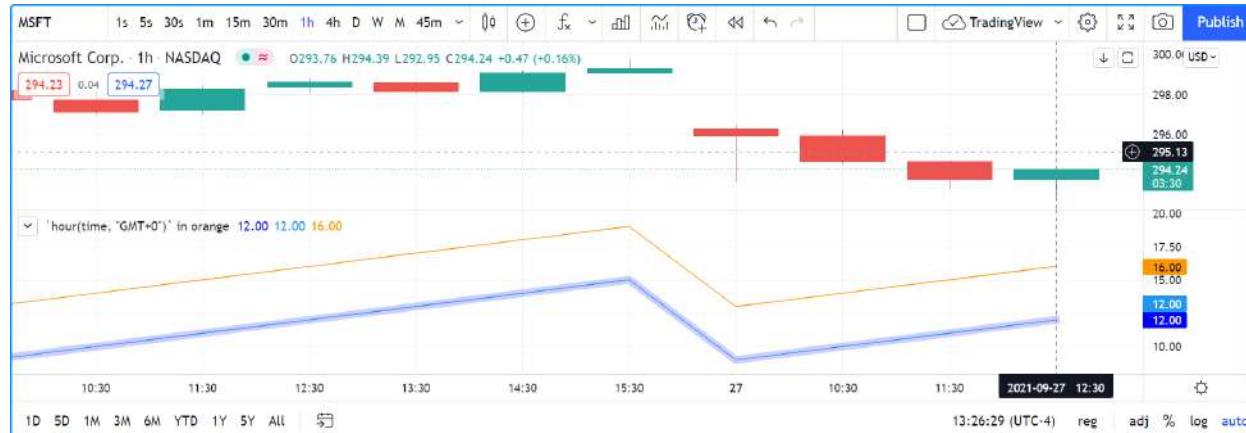
1 //@version=5
2 indicator("Time zone control")
3 MS_IN_1H = 1000 * 60 * 60
4 TOOLTIP01 = "Enter your time zone's offset (+ or -), including a decimal fraction if needed."
5 hoursOffsetInput = input.float(0.0, "Timezone offset (in hours)", minval = -12.0, maxval = 14.0, step = 0.5, tooltip = TOOLTIP01)
6
7 printTable(txt) =>
8     var table t = table.new(position.middle_right, 1, 1)
9     table.cell(t, 0, 0, txt, text_align_right, bgcolor = color.yellow)
10
11 msOffsetInput = hoursOffsetInput * MS_IN_1H
12 printTable(
13     str.format("Last bar's open time UTC: {0,date,HH:mm:ss yyyy.MM.dd}", time) +
14     str.format("\nLast bar's close time UTC: {0,date,HH:mm:ss yyyy.MM.dd}", time_close) +
15     str.format("\n\nLast bar's open time EXCHANGE: {0,date,HH:mm:ss yyyy.MM.dd}", time(timeframe.period, syminfo.session, syminfo.timezone)) +
16     str.format("\nLast bar's close time EXCHANGE: {0,date,HH:mm:ss yyyy.MM.dd}", time_close(timeframe.period, syminfo.session, syminfo.timezone)) +
17     str.format("\n\nLast bar's open time OFFSET ({0}): {1,date,HH:mm:ss yyyy.MM.dd}", time + hoursOffsetInput, time + msOffsetInput) +
18     str.format("\nLast bar's close time OFFSET ({0}): {1,date,HH:mm:ss yyyy.MM.dd}", time_close + hoursOffsetInput, time_close + msOffsetInput) +
19     str.format("\n\nCurrent time OFFSET ({0}): {1,date,HH:mm:ss yyyy.MM.dd}", timenow + hoursOffsetInput, timenow + msOffsetInput))

```

Note that:

- We convert the user offset expressed in hours to milliseconds with `msOffsetInput`. We then add that offset to a timestamp in UTC format before converting it to display format, e.g., `time + msOffsetInput` and `timenow + msOffsetInput`.
- We use a tooltip to provide instructions to users.
- We provide `minval` and `maxval` values to protect the input field, and a `step` value of 0.5 so that when they use the field's up/down arrows, they can intuitively figure out that fractions can be used.
- The `str.format()` function formats our time values, namely the last bar's time and the current time.

Some functions that normally return values in the exchange's time zone provide means to adapt their result to another time zone through the `timezone` parameter. This script illustrates how to do this with `hour()`:



```

1 //@version=5
2 indicator(`hour(time, "GMT+0")` in orange)
3 color BLUE_LIGHT = #0000FF30
4 plot(hour, "", BLUE_LIGHT, 8)
5 plot(hour(time, syminfo.timezone))
6 plot(hour(time, "GMT+0"), "UTC", color.orange)

```

Note that:

- The `hour` variable and the `hour()` function normally returns a value in the exchange's time zone. Accordingly, plots in blue for both `hour` and `hour(time, syminfo.timezone)` overlap. Using the function form with `syminfo.timezone` is thus redundant if the exchange's hour is required.
- The orange line plotting `hour(time, "GMT+0")`, however, returns the bar's hour at UTC, or "GMT+0" time, which in this case is four hours less than the exchange's time, since MSFT trades on the NASDAQ whose time zone is UTC-4.

### Time zone strings

The argument used for the `timezone` parameter in functions such as `time()`, `timestamp()`, `hour()`, etc., can be in different formats, which you can find in the [IANA time zone database name](#) reference page. Contents from the "TZ database name", "UTC offset ±hh:mm" and "UTC DST offset ±hh:mm" columns of that page's table can be used.

To express an offset of +5.5 hours from UTC, these strings found in the reference page are all equivalent:

- "GMT+05:30"
- "Asia/Calcutta"
- "Asia/Colombo"

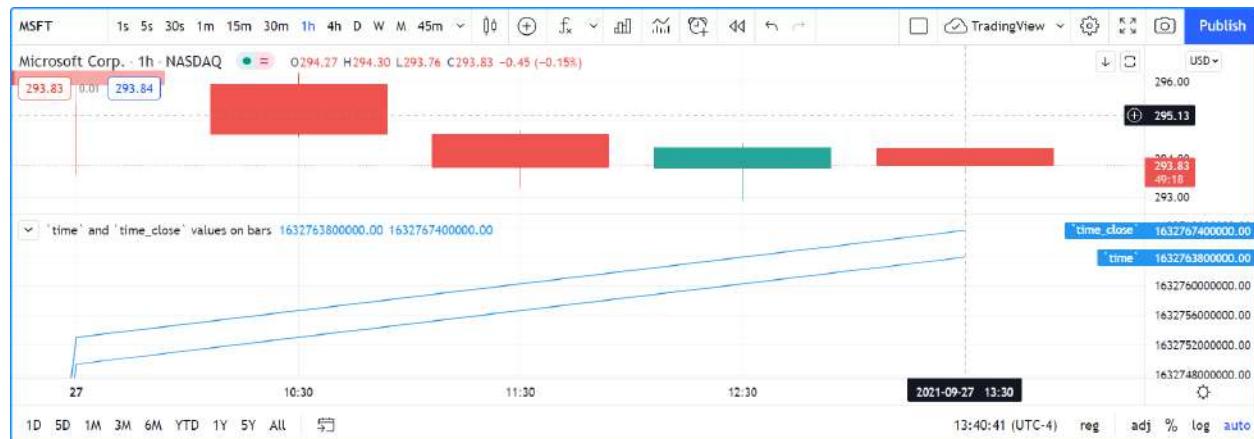
- "Asia/Kolkata"

Non-fractional offsets can be expressed in the "GMT+5" form. "GMT+5 . 5" is not allowed.

## 4.21.2 Time variables

### 'time` and 'time\_close`

Let's start by plotting `time` and `time_close`, the Unix timestamp in milliseconds of the bar's opening and closing time:



```

1 //@version=5
2 indicator("`time` and `time_close` values on bars")
3 plot(time, "`time`")
4 plot(time_close, "`time_close`")

```

Note that:

- The `time` and `time_close` variables returns a timestamp in **UNIX time**, which is independent of the timezone selected by the user on his chart. In this case, the **chart's** time zone setting is the exchange time zone, so whatever symbol is on the chart, its exchange time zone will be used to display the date and time values on the chart's cursor. The NASDAQ's time zone is UTC-4, but this only affects the chart's display of date/time values; it does not impact the values plotted by the script.
- The last `time` value for the plot shown in the scale is the number of milliseconds elapsed from 00:00:00 UTC, 1 January, 1970, until the bar's opening time. It corresponds to 17:30 on the 27th of September 2021. However, because the chart uses the UTC-4 time zone (the NASDAQ's time zone), it displays the 13:30 time, four hours earlier than UTC time.
- The difference between the two values on the last bar is the number of milliseconds in one hour ( $1000 * 60 * 60 = 3,600,000$ ) because we are on a 1H chart.

### `time\_tradingday`

`time_tradingday` is useful when a symbol trades on overnight sessions that start and close on different calendar days. For example, this happens in forex markets where a session can open Sunday at 17:00 and close Monday at 17:00.

The variable returns the time of the beginning of the trading day in **UNIX time** when used at timeframes of 1D and less. When used on timeframes higher than 1D, it returns the starting time of the last trading day in the bar (e.g., at 1W, it will return the starting time of the last trading day of the week).

### `timenow`

`timenow` returns the current time in **UNIX time**. It works in realtime, but also when a script executes on historical bars. In realtime, your scripts will only perceive changes when they execute on feed updates. When no updates occur, the script is idle, so it cannot update its display. See the page on Pine Script™'s *execution model* for more information.

This script uses the values of `timenow` and `time_close` to calculate a realtime countdown for intraday bars. Contrary to the countdown on the chart, this one will only update when a feed update causes the script to execute another iteration:

```

1 // @version=5
2 indicator("", "", true)
3
4 printTable(txt) =>
5     var table t = table.new(position.middle_right, 1, 1)
6     table.cell(t, 0, 0, txt, text_align_right, bgcolor = color.yellow)
7
8 printTable(str.format("{0,time,HH:mm:ss.SSS}", time_close - timenow))

```

### Calendar dates and times

Calendar date and time variables such as `year`, `month`, `weekofyear`, `dayofmonth`, `dayofweek`, `hour`, `minute` and `second` can be useful to test for specific dates or times, and as arguments to `timestamp()`.

When testing for specific dates or times, ones needs to account for the possibility that the script will be executing on timeframes where the tested condition cannot be detected, or for cases where a bar with the specific requirement will not exist. Suppose, for example, we wanted to detect the first trading day of the month. This script shows how using only `dayofmonth` will not work when a weekly chart is used or when no trading occurs on the 1st of the month:



```

1 //@version=5
2 indicator("", "", true)
3 firstDayIncorrect = dayofmonth == 1
4 firstDay = ta.change(time("M"))
5 plotchar(firstDayIncorrect, "firstDayIncorrect", ".", location.top, size = size.small)
6 bgcolor(firstDay ? color.silver : na)

```

Note that:

- Using `ta.change(time("M"))` is more robust as it works on all months (#1 and #2), displayed as the silver background, whereas the blue dot detected using `dayofmonth == 1` does not work (#1) when the first trading day of September occurs on the 2nd.
- The `dayofmonth == 1` condition will be `true` on all intrabars of the first day of the month, but `ta.change(time("M"))` will only be `true` on the first.

If you wanted your script to only display for years 2020 and later, you could use:

```

1 //@version=5
2 indicator("", "", true)
3 plot(year >= 2020 ? close : na, linewidth = 3)

```

### `'syminfo.timezone()'`

`syminfo.timezone` returns the time zone of the chart symbol's exchange. It can be helpful when a `timezone` parameter is available in a function, and you want to mention that you are using the exchange's timezone explicitly. It is usually redundant because when no argument is supplied to `timezone`, the exchange's time zone is assumed.

## 4.21.3 Time functions

### `'time()'` and `'time_close()'`

The `time()` and `time_close()` functions have the following signature:

```

time(timeframe, session, timezone) → series int
time_close(timeframe, session, timezone) → series int

```

They accept three arguments:

#### **timeframe**

A string in `timeframe.period` format.

#### **session**

An optional string in session specification format: "`hhmm-hhmm[:days]`", where the `[:days]` part is optional. See the page on [sessions](#) for more information.

#### **timezone**

An optional value that qualifies the argument for `session` when one is used.

See the `time()` and `time_close()` entries in the Reference Manual for more information.

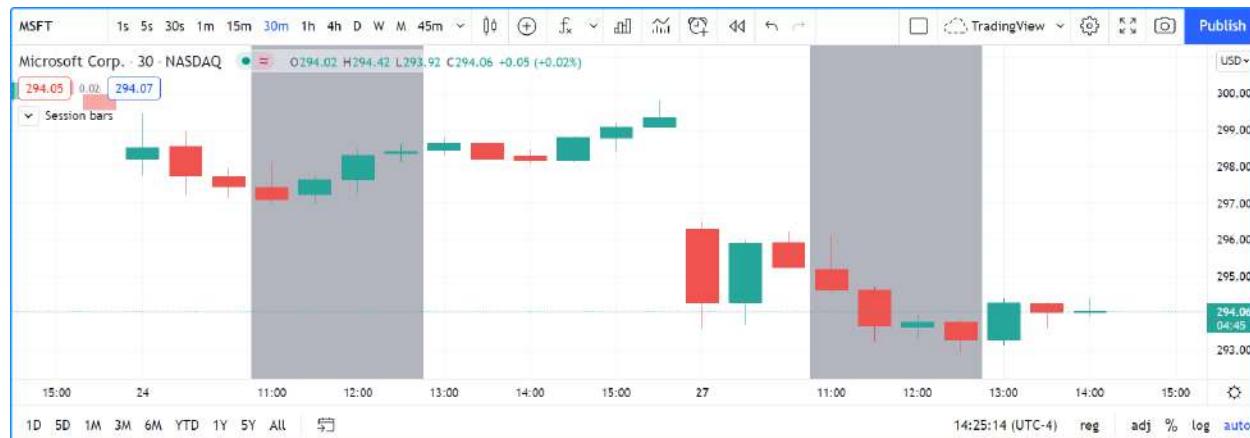
The `time()` function is most often used to:

1. Test if a bar is in a specific time period, which will require using the `session` parameter. In those cases, `timeframe.period`, i.e., the chart's timeframe, will often be used for the first parameter. When using the function this way, we rely on the fact that it will return `na` when the bar is not part of the period specified in the `session` argument.

2. Detecting changes in higher timeframes than the chart's by using the higher timeframe for the `timeframe` argument. When using the function for this purpose, we are looking for changes in the returned value, which means the higher timeframe bar has changed. This will usually require using `ta.change()` to test, e.g., `ta.change(time("D"))` will return the change in time when a new higher timeframe bar comes in, so the expression's result will cast to a "bool" value when used in a conditional expression. The "bool" result will be `true` when there is a change and `false` when there is no change.

### Testing for sessions

Let's look at an example of the first case where we want to determine if a bar's starting time is part of a period between 11:00 and 13:00:



```

1 // @version=5
2 indicator("Session bars", "", true)
3 inSession = not na(time(timeframe.period, "1100-1300"))
4 bgcolor(inSession ? color.silver : na)

```

Note that:

- We use `time(timeframe.period, "1100-1300")`, which says: "Check the chart's timeframe if the current bar's opening time is between 11:00 and 13:00 inclusively". The function returns its opening time if the bar is in the session. If it is **not**, the function returns `na`.
- We are interested in identifying the instances when `time()` does not return `na` because that means the bar is in the session, so we test for `not na(...)`. We do not use the actual return value of `time()` when it is not `na`; we are only interested in whether it returns `na` or not.

### Testing for changes in higher timeframes

It is often helpful to detect changes in a higher timeframe. For example, you may want to detect trading day changes while on intraday charts. For these cases, you can use the fact that `time("D")` returns the opening time of the 1D bar, even if the chart is at an intraday timeframe such as 1H:



```

1 // @version=5
2 indicator("", "", true)
3 bool newDay = ta.change(time("D"))
4 bgcolor(newDay ? color.silver : na)
5
6 newExchangeDay = ta.change(dayofmonth)
7 plotchar(newExchangeDay, "newExchangeDay", "▣", location.top, size = size.small)

```

Note that:

- The `newDay` variable detects changes in the opening time of 1D bars, so it follows the conventions for the chart's symbol, which uses overnight sessions of 17:00 to 17:00. It changes values when a new session comes in.
- Because `newExchangeDay` detects change in `dayofmonth` in the calendar day, it changes when the day changes on the chart.
- The two change detection methods only coincide on the chart when there are days without trading. On Sundays here, for example, both detection methods will detect a change because the calendar day changes from the last trading day (Friday) to the first calendar day of the new week, Sunday, which is when Monday's overnight session begins at 17:00.

## Calendar dates and times

Calendar date and time functions such as `year()`, `month()`, `weekofyear()`, `dayofmonth()`, `dayofweek()`, `hour()`, `minute()` and `second()` can be useful to test for specific dates or times. They all have signatures similar to the ones shown here for `dayofmonth()`:

```

dayofmonth(time) → series int
dayofmonth(time, timezone) → series int

```

This will plot the day of the opening of the bar where the January 1st, 2021 at 00:00 time falls between its `time` and `time_close` values:

```

1 // @version=5
2 indicator("")
3 exchangeDay = dayofmonth(timestamp("2021-01-01"))
4 plot(exchangeDay)

```

The value will be the 31st or the 1st, depending on the calendar day of when the session opens on the chart's symbol. The date for symbols traded 24x7 at exchanges using the UTC time zone will be the 1st. For symbols trading on exchanges at UTC-4, the date will be the 31st.

## `timestamp()`

The `timestamp()` function has a few different signatures:

```
timestamp(year, month, day, hour, minute, second) → simple/series int
timestamp(timezone, year, month, day, hour, minute, second) → simple/series int
timestamp(dateString) → const int
```

The only difference between the first two is the `timezone` parameter. Its default value is `syminfo.timezone`. See the [Time zone strings](#) section of this page for valid values.

The third form is used as a `defval` value in `input.time()`. See the `timestamp()` entry in the Reference Manual for more information.

`timestamp()` is useful to generate a timestamp for a specific date. To generate a timestamp for Jan 1, 2021, use either one of these methods:

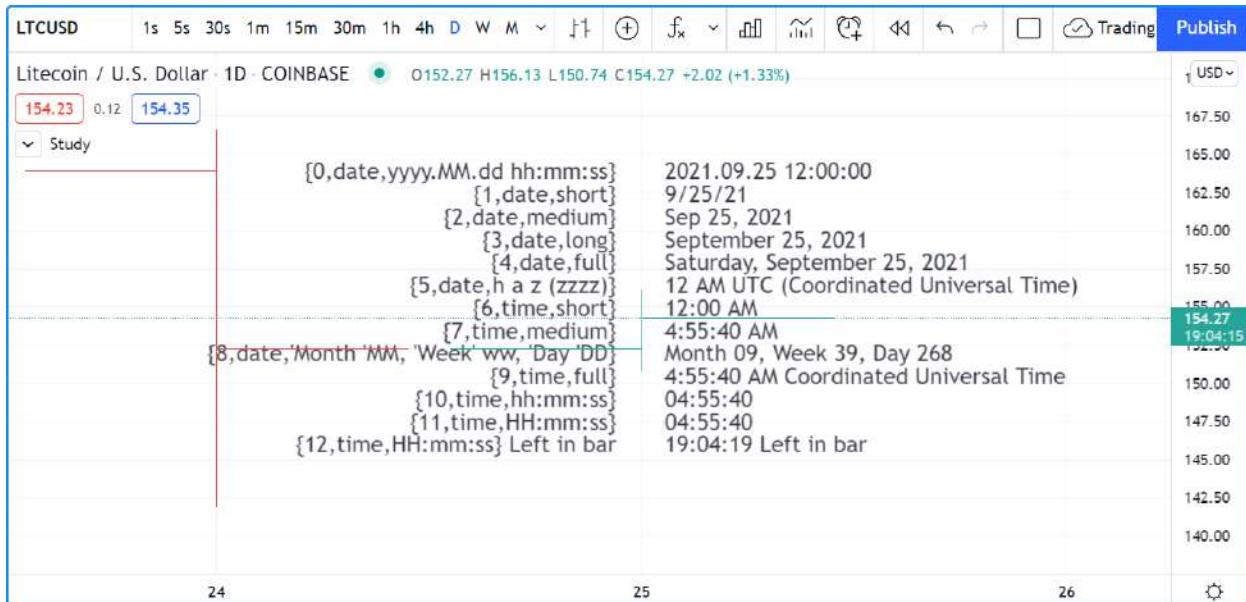
```
1 // @version=5
2 indicator("")
3 yearBeginning1 = timestamp("2021-01-01")
4 yearBeginning2 = timestamp(2021, 1, 1, 0, 0)
5 printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t,
   ↪ 0, 0, txt, bgcolor = color.yellow)
6 printTable(str.format("yearBeginning1: {0,date,yyyy.MM.dd hh:mm}\nyearBeginning2: {1,
   ↪ date,yyyy.MM.dd hh:mm}", yearBeginning1, yearBeginning2))
```

You can use offsets in `timestamp()` arguments. Here, we subtract 2 from the value supplied for its `day` parameter to get the date/time from the chart's last bar two days ago. Note that because of different bar alignments on various instruments, the bar identified on the chart may not always be exactly 48 hours away, although the function's return value is correct:

```
1 // @version=5
2 indicator("")
3 twoDaysAgo = timestamp(year, month, dayofmonth - 2, hour, minute)
4 printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t,
   ↪ 0, 0, txt, bgcolor = color.yellow)
5 printTable(str.format("{0,date,yyyy.MM.dd hh:mm}", twoDaysAgo))
```

### 4.21.4 Formatting dates and time

Timestamps can be formatted using `str.format()`. These are examples of various formats:



```

1 //@version=5
2 indicator("", "", true)
3
4 print(txt, styl) =>
5     var alignment = styl == label.style_label_right ? text.align_right : text.align_
6         ←left
7     var lbl = label.new(na, na, "", xloc.bar_index, yloc.price, color(na), styl, ←
8         color.black, size.large, alignment)
9     if barstate.islast
10        label.set_xy(lbl, bar_index, hl2[1])
11        label.set_text(lbl, txt)
12
13 var string format =
14     "{0,date,yyyy.MM.dd hh:mm:ss}\n" +
15     "{1,date,short}\n" +
16     "{2,date,medium}\n" +
17     "{3,date,long}\n" +
18     "{4,date,full}\n" +
19     "{5,date,h a z (zzzz)}\n" +
20     "{6,time,short}\n" +
21     "{7,time,medium}\n" +
22     "{8,date,'Month' 'MM', 'Week' 'ww', 'Day' 'DD'}\n" +
23     "{9,time,full}\n" +
24     "{10,time,hh:mm:ss}\n" +
25     "{11,time,HH:mm:ss}\n" +
26     "{12,time,HH:mm:ss} Left in bar\n"
27
28 print(format, label.style_label_right)
29 print(str.format(format,
30     time, time, time, time, time, time,
31     timenow, timenow, timenow, timenow,
32     timenow - time, time_close - timenow), label.style_label_left)

```





## 4.22 Timeframes

- *Introduction*
- *Timeframe string specifications*
- *Comparing timeframes*

### 4.22.1 Introduction

The *timeframe* of a chart is sometimes also referred to as its *interval* or *resolution*. It is the unit of time represented by one bar on the chart. All standard chart types use a timeframe: “Bars”, “Candles”, “Hollow Candles”, “Line”, “Area” and “Baseline”. One non-standard chart type also uses timeframes: “Heikin Ashi”.

Programmers interested in accessing data from multiple timeframes will need to become familiar with how timeframes are expressed in Pine Script™, and how to use them.

**Timeframe strings** come into play in different contexts:

- They must be used in `request.security()` when requesting data from another symbol and/or timeframe. See the page on *Other timeframes and data* to explore the use of `request.security()`.
- They can be used as an argument to `time()` and `time_close()` functions, to return the time of a higher timeframe bar. This, in turn, can be used to detect changes in higher timeframes from the chart’s timeframe without using `request.security()`. See the *Testing for changes in higher timeframes* section to see how to do this.
- The `input.timeframe()` function provides a way to allow script users to define a timeframe through a script’s “Inputs” tab (see the *Timeframe input* section for more information).
- The `indicator()` declaration statement has an optional `timeframe` parameter that can be used to provide multi-timeframe capabilities to simple scripts without using `request.security()`.
- Many built-in variables provide information on the timeframe used by the chart the script is running on. See the *Chart timeframe* section for more information on them, including `timeframe.period` which returns a string in Pine Script™’s timeframe specification format.

## 4.22.2 Timeframe string specifications

Timeframe strings follow these rules:

- They are composed of the multiplier and the timeframe unit, e.g., “1S”, “30” (30 minutes), “1D” (one day), “3M” (three months).
- The unit is represented by a single letter, with no letter used for minutes: “S” for seconds, “D” for days, “W” for weeks and “M” for months.
- When no multiplier is used, 1 is assumed: “S” is equivalent to “1S”, “D” to “1D”, etc. If only “1” is used, it is interpreted as “1min”, since no unit letter identifier is used for minutes.
- There is no “hour” unit; “1H” is **not** valid. The correct format for one hour is “60” (remember no unit letter is specified for minutes).
- The valid multipliers vary for each timeframe unit:
  - For seconds, only the discrete 1, 5, 10, 15 and 30 multipliers are valid.
  - For minutes, 1 to 1440.
  - For days, 1 to 365.
  - For weeks, 1 to 52.
  - For months, 1 to 12.

## 4.22.3 Comparing timeframes

It can be useful to compare different timeframe strings to determine, for example, if the timeframe used on the chart is lower than the higher timeframes used in the script.

Converting timeframe strings to a representation in fractional minutes provides a way to compare them using a universal unit. This script uses the `timeframe.in_seconds()` function to convert a timeframe into float seconds and then converts the result into minutes:

```

1 //@version=5
2 indicator("Timeframe in minutes example", "", true)
3 string tfInput = input.timeframe(defval = "", title = "Input TF")
4
5 float chartTFInMinutes = timeframe.in_seconds() / 60
6 float inputTFInMinutes = timeframe.in_seconds(tfInput) / 60
7
8 var table t = table.new(position.top_right, 1, 1)
9 string txt = "Chart TF: " + str.tostring(chartTFInMinutes, "#.##### minutes") +
10 "\nInput TF: " + str.tostring(inputTFInMinutes, "#.##### minutes")
11 if barstate.isfirst
12     table.cell(t, 0, 0, txt, bgcolor = color.yellow)
13 else if barstate.islast
14     table.cell_set_text(t, 0, 0, txt)
15
16 if chartTFInMinutes > inputTFInMinutes
17     runtime.error("The chart's timeframe must not be higher than the input's
18     ↪timeframe.")

```

Note that:

- We use the built-in `timeframe.in_seconds()` function to convert the chart and the `input.timeframe()` function into seconds, then divide by 60 to convert into minutes.

- We use two calls to the `timeframe.in_seconds()` function in the initialization of the `chartTFInMinutes` and `inputTFInMinutes` variables. In the first instance, we do not supply an argument for its `timeframe` parameter, so the function returns the chart's timeframe in seconds. In the second call, we supply the timeframe selected by the script's user through the call to `input.timeframe()`.
- Next, we validate the timeframes to ensure that the input timeframe is equal to or higher than the chart's timeframe. If it is not, we generate a runtime error.
- We finally print the two timeframe values converted to minutes.



## WRITING SCRIPTS



### 5.1 Style guide

- *Introduction*
- *Naming Conventions*
- *Script organization*
- *Spacing*
- *Line wrapping*
- *Vertical alignment*
- *Explicit typing*

#### 5.1.1 Introduction

This style guide provides recommendations on how to name variables and organize your Pine scripts in a standard way that works well. Scripts that follow our best practices will be easier to read, understand and maintain.

You can see scripts using these guidelines published from the [TradingView](#) and [PineCoders](#) accounts on the platform.

## 5.1.2 Naming Conventions

We recommend the use of:

- camelCase for all identifiers, i.e., variable or function names: ma, maFast, maLengthInput, maColor, roundedOHLC(), pivotHi().
- All caps SNAKE\_CASE for constants: BULL\_COLOR, BEAR\_COLOR, MAX\_LOOKBACK.
- The use of qualifying suffixes when it provides valuable clues about the type or provenance of a variable: maShowInput, bearColor, bearColorInput, volumesArray, maPlotID, resultsTable, levelsColorArray.

## 5.1.3 Script organization

The Pine Script™ compiler is quite forgiving of the positioning of specific statements or the version *compiler annotation* in the script. While other arrangements are syntactically correct, this is how we recommend organizing scripts:

```
<license>
<version>
<declaration_statement>
<import_statements>
<constant_declarations>
<inputs>
<function_declarations>
<calculations>
<strategy_calls>
<visuals>
<alerts>
```

### <license>

If you publish your open-source scripts publicly on TradingView (scripts can also be published privately), your open-source code is by default protected by the Mozilla license. You may choose any other license you prefer.

The reuse of code from those scripts is governed by our [House Rules on Script Publishing](#) which preempt the author's license.

The standard license comments appearing at the beginning of scripts are:

```
// This source code is subject to the terms of the Mozilla Public License 2.0 at_
https://mozilla.org/MPL/2.0/
// © username
```

### <version>

This is the *compiler annotation* defining the version of Pine Script™ the script will use. If none is present, v1 is used. For v5, use:

```
//@version=5
```

## <declaration\_statement>

This is the mandatory declaration statement which defines the type of your script. It must be a call to either `indicator()`, `strategy()`, or `library()`.

## <import\_statements>

If your script uses one or more Pine Script™ *libraries*, your `import` statements belong here.

## <constant\_declarations>

Scripts can declare variables qualified as “const”, i.e., ones referencing a constant value.

We refer to variables as “constants” when they meet these criteria:

- Their declaration uses the optional `const` keyword (see our User Manual’s section on *type qualifiers* for more information).
- They are initialized using a literal (e.g., `100` or `"AAPL"`) or a built-in qualified as “const” (e.g., `color.green`).
- Their value does not change during the script’s execution.

We use SNAKE\_CASE to name these variables and group their declaration near the top of the script. For example:

```
// ----- Constants
int    MS_IN_MIN     = 60 * 1000
int    MS_IN_HOUR    = MS_IN_MIN    * 60
int    MS_IN_DAY     = MS_IN_HOUR * 24

color   GRAY         = #808080ff
color   LIME         = #00FF00ff
color   MAROON       = #800000ff
color   ORANGE       = #FF8000ff
color   PINK          = #FF0080ff
color   TEAL          = #008080ff
color   BG_DIV        = color.new(ORANGE, 90)
color   BG_RESETS    = color.new(GRAY, 90)

string  RST1         = "No reset; cumulate since the beginning of the chart"
string  RST2         = "On a stepped higher timeframe (HTF)"
string  RST3         = "On a fixed HTF"
string  RST4         = "At a fixed time"
string  RST5         = "At the beginning of the regular session"
string  RST6         = "At the first visible chart bar"
string  RST7         = "Fixed rolling period"

string  LTF1         = "Least precise, covering many chart bars"
string  LTF2         = "Less precise, covering some chart bars"
string  LTF3         = "More precise, covering less chart bars"
string  LTF4         = "Most precise, 1min intrabars"

string  TT_TOTVOL    = "The 'Bodies' value is the transparency of the total volume"
                     "candle bodies. Zero is opaque, 100 is transparent."
string  TT_RST-HTF   = "This value is used when '" + RST3 + "' is selected."
string  TT_RST-TIME  = "These values are used when '" + RST4 + "' is selected.
                     A reset will occur when the time is greater or equal to the bar's open time, and"
```

(continues on next page)

(continued from previous page)

```
↳ less than its close time.\nHour: 0-23\nMinute: 0-59"
string TT_RST_PERIOD = "This value is used when '" + RST7 +"' is selected."
```

In this example:

- The RST\* and LTF\* constants will be used as tuple elements in the options argument of `input.*()` calls.
- The TT\_\* constants will be used as tooltip arguments in `input.*()` calls. Note how we use a line continuation for long string literals.
- We do not use `var` to initialize constants. The Pine Script™ runtime is optimized to handle declarations on each bar, but using `var` to initialize a variable only the first time it is declared incurs a minor penalty on script performance because of the maintenance that `var` variables require on further bars.

Note that:

- Literals used in more than one place in a script should always be declared as a constant. Using the constant rather than the literal makes it more readable if it is given a meaningful name, and the practice makes code easier to maintain. Even though the quantity of milliseconds in a day is unlikely to change in the future, `MS_IN_DAY` is more meaningful than `1000 * 60 * 60 * 24`.
- Constants only used in the local block of a function or `if`, `while`, etc., statement for example, can be declared in that local block.

### <inputs>

It is **much** easier to read scripts when all their inputs are in the same code section. Placing that section at the beginning of the script also reflects how they are processed at runtime, i.e., before the rest of the script is executed.

Suffixing input variable names with `input` makes them more readily identifiable when they are used later in the script: `maLengthInput`, `bearColorInput`, `showAvgInput`, etc.

```
// ----- Inputs
string resetInput           = input.string(RST2,          "CVD Resets",
    ↳      inline = "00", options = [RST1, RST2, RST3, RST4, RST5, RST6, RST7])
string fixedTfInput         = input.timeframe("D",        " Fixed HTF: ",
    ↳      tooltip = TT_RST_HTF)
int    hourInput             = input.int(9,              " Fixed time hour: ",
    ↳      inline = "01", minval = 0, maxval = 23)
int    minuteInput           = input.int(30,             "minute",
    ↳      inline = "01", minval = 0, maxval = 59, tooltip = TT_RST_TIME)
int    fixedPeriodInput      = input.int(20,             " Fixed period: ",
    ↳      inline = "02", minval = 1, tooltip = TT_RST_PERIOD)
string ltfModeInput          = input.string(LTF3,          "Intrabar precision",
    ↳      inline = "03", options = [LTF1, LTF2, LTF3, LTF4])
```

## <function\_declarations>

All user-defined functions must be defined in the script's global scope; nested function definitions are not allowed in Pine Script™.

Optimal function design should minimize the use of global variables in the function's scope, as they undermine function portability. When it can't be avoided, those functions must follow the global variable declarations in the code, which entails they can't always be placed in the <function\_declarations> section. Such dependencies on global variables should ideally be documented in the function's comments.

It will also help readers if you document the function's objective, parameters and result. The same syntax used in *libraries* can be used to document your functions. This can make it easier to port your functions to a library should you ever decide to do so:

```

1 // @version=5
2 indicator("<function_declarations>", "", true)
3
4 string SIZE_LARGE = "Large"
5 string SIZE_NORMAL = "Normal"
6 string SIZE_SMALL = "Small"
7
8 string sizeInput = input.string(SIZE_NORMAL, "Size", options = [SIZE_LARGE, SIZE_
9   ↪NORMAL, SIZE_SMALL])
10
11 // @function      Used to produce an argument for the `size` parameter in built-in_
12 //   ↪functions.
13 // @param userSize (simple string) User-selected size.
14 // @returns        One of the `size.*` built-in constants.
15 // Dependencies: SIZE_LARGE, SIZE_NORMAL, SIZE_SMALL
16 getSize(simple string userSize) =>
17     result =
18     switch userSize
19         SIZE_LARGE  => size.large
20         SIZE_NORMAL => size.normal
21         SIZE_SMALL   => size.small
22         => size.auto
23
24 if ta.rising(close, 3)
25     label.new(bar_index, na, yloc = yloc.abovebar, style = label.style_arrowup, size_
26   ↪= getSize(sizeInput))

```

## <calculations>

This is where the script's core calculations and logic should be placed. Code can be easier to read when variable declarations are placed near the code segment using the variables. Some programmers prefer to place all their non-constant variable declarations at the beginning of this section, which is not always possible for all variables, as some may require some calculations to have been executed before their declaration.

### <strategy\_calls>

Strategies are easier to read when strategy calls are grouped in the same section of the script.

### <visuals>

This section should ideally include all the statements producing the script's visuals, whether they be plots, drawings, background colors, candle-plotting, etc. See the Pine Script™ User Manual's section on [here](#) for more information on how the relative depth of visuals is determined.

### <alerts>

Alert code will usually require the script's calculations to have executed before it, so it makes sense to put it at the end of the script.

## 5.1.4 Spacing

A space should be used on both sides of all operators, except unary operators (-1). A space is also recommended after all commas and when using named function arguments, as in `plot(series = close)`:

```
int a = close > open ? 1 : -1
var int newLen = 2
newLen := min(20, newlen + 1)
float a = -b
float c = d > e ? d - e : d
int index = bar_index % 2 == 0 ? 1 : 2
plot(close, color = color.red)
```

## 5.1.5 Line wrapping

Line wrapping can make long lines easier to read. Line wraps are defined by using an indentation level that is not a multiple of four, as four spaces or a tab are used to define local blocks. Here we use two spaces:

```
plot(
    series = close,
    title = "Close",
    color = color.blue,
    show_last = 10
)
```

## 5.1.6 Vertical alignment

Vertical alignment using tabs or spaces can be useful in code sections containing many similar lines such as constant declarations or inputs. They can make mass edits much easier using the Pine Editor's multi-cursor feature (`ctrl + alt + ⌘/⌘`):

```
// Colors used as defaults in inputs.
color COLOR_AQUA = #0080FFff
color COLOR_BLACK = #000000ff
color COLOR_BLUE = #013BCAff
```

(continues on next page)

(continued from previous page)

```
color COLOR_CORAL = #FF8080ff
color COLOR_GOLD  = #CCCC00ff
```

### 5.1.7 Explicit typing

Including the type of variables when declaring them is not required. However, it helps make scripts easier to read, navigate, and understand. It can help clarify the expected types at each point in a script's execution and distinguish a variable's declaration (using `=`) from its reassessments (using `:=`). Using explicit typing can also make scripts easier to *debug*.



## 5.2 Debugging

- *Introduction*
- *The lay of the land*
- *Numeric values*
- *Conditions*
- *Strings*
- *Pine Logs*
- *Debugging functions*
- *Debugging loops*
- *Tips*

### 5.2.1 Introduction

TradingView's close integration between the Pine Editor and the chart interface facilitates efficient, interactive debugging of Pine Script™ code, as scripts can produce dynamic results in multiple locations, on and off the chart. Programmers can utilize such results to refine their script's behaviors and ensure everything works as expected.

When a programmer understands the appropriate techniques for inspecting the variety of behaviors one may encounter while writing a script, they can quickly and thoroughly identify and resolve potential problems in their code, which allows for a more seamless overall coding experience. This page demonstrates some of the handiest ways to debug code when working with Pine Script™.

**Note:** Before venturing further on this page, we recommend familiarizing yourself with Pine's *Execution model* and *Type system*, as it's crucial to understand these details when debugging in the Pine Script™ environment.

### 5.2.2 The lay of the land

Pine scripts can output their results in multiple different ways, any of which programmers can utilize for debugging.

The `plot()` functions can display results in a chart pane, the script's status line, the price (y-axis) scale, and the Data Window, providing simple, convenient ways to debug numeric and conditional values:



```

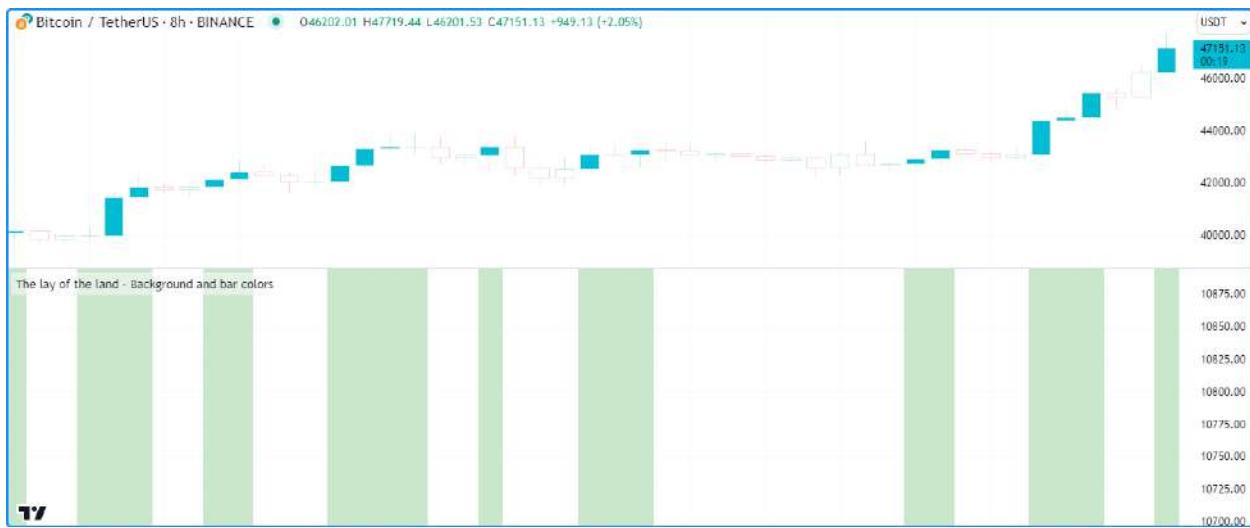
1 // @version=5
2 indicator("The lay of the land - Plots")
3
4 // Plot the `bar_index` in all available locations.
5 plot(bar_index, "bar_index", color.teal, 3)

```

#### Note that:

- A script's status line outputs will only show when enabling the “Values” checkbox within the “Indicators” section of the chart's “Status line” settings.
- Price scales will only show plot values or names when enabling the options from the “Indicators and financials” dropdown in the chart's “Scales and lines” settings.

The `bcolor()` function displays colors in the script pane's background, and the `barcolor()` function changes the colors of the main chart's bars or candles. Both of these functions provide a simple way to visualize conditions:

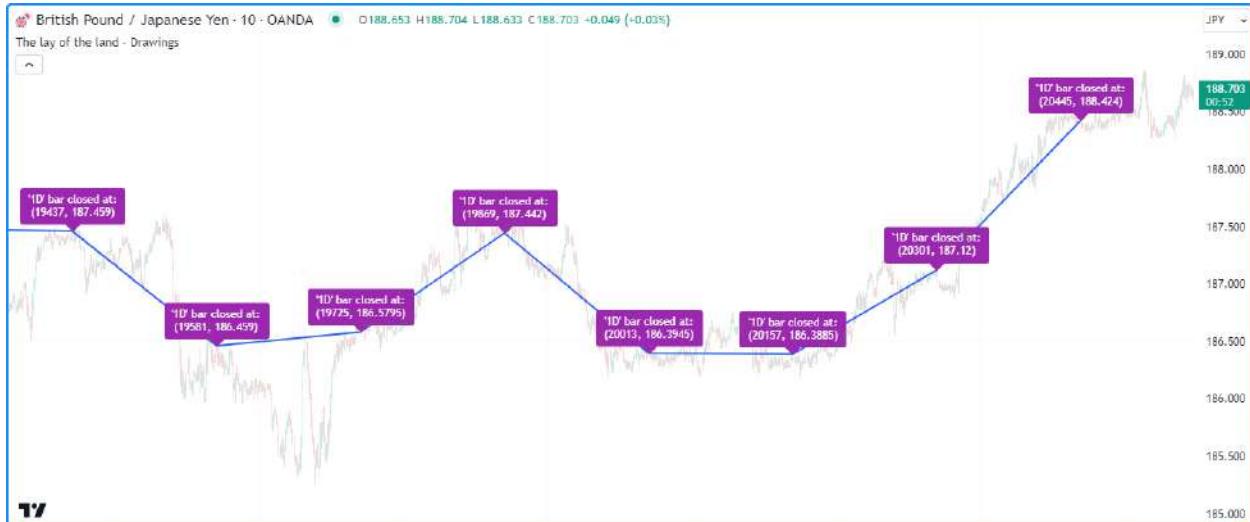


```

1 // @version=5
2 indicator("The lay of the land - Background and bar colors")
3
4 //@variable Is `true` if the `close` is rising over 2 bars.
5 bool risingPrice = ta.rising(close, 2)
6
7 // Highlight the chart background and color the main chart bars based on
8 // `risingPrice`.
9 bgcolor(risingPrice ? color.new(color.green, 70) : na, title= "`risingPrice` highlight
10 ")
11 barcolor(risingPrice ? color.aqua : chart.bg_color, title = "`risingPrice` bar color")

```

Pine's *drawing types* (*line*, *box*, *polyline*, *label*, and *table*) produce drawings in the script's pane. While they don't return results in other locations, such as the status line or Data Window, they provide alternative, flexible solutions for inspecting numeric values, conditions, and strings directly on the chart:



```

1 // @version=5
2 indicator("The lay of the land - Drawings", overlay = true)
3
4 //@variable Is `true` when the time changes on the "1D" timeframe.

```

(continues on next page)

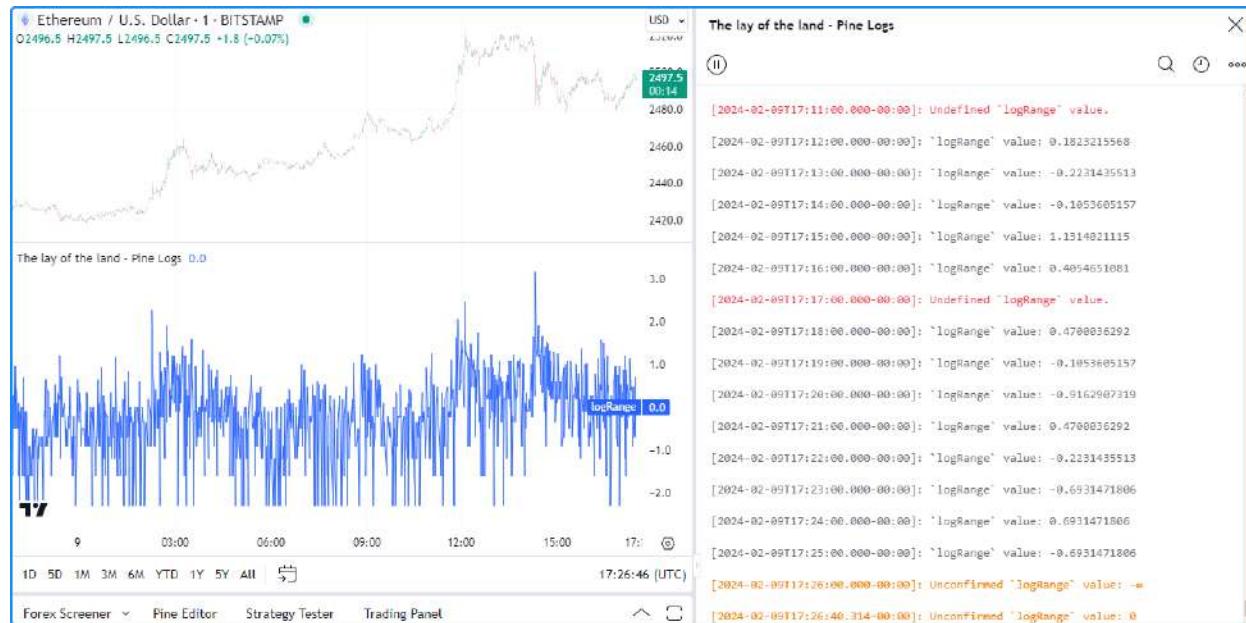
(continued from previous page)

```

5 bool newDailyBar = timeframe.change("1D")
6 // @variable The previous bar's `bar_index` from when `newDailyBar` last occurred.
7 int closedIndex = ta.valuewhen(newDailyBar, bar_index - 1, 0)
8 // @variable The previous bar's `close` from when `newDailyBar` last occurred.
9 float closedPrice = ta.valuewhen(newDailyBar, close[1], 0)
10
11 if newDailyBar
12     // @variable Draws a line from the previous `closedIndex` and `closedPrice` to the
13     // current values.
14     line debugLine = line.new(closedIndex[1], closedPrice[1], closedIndex,_
15     closedPrice, width = 2)
16     // @variable Variable info to display in a label.
17     string debugText = "'1D' bar closed at: \n(" + str.tostring(closedIndex) + ", " +_
18     str.tostring(closedPrice) + ")"
19     // @variable Draws a label at the current `closedIndex` and `closedPrice`.
20     label.new(closedIndex, closedPrice, debugText, color = color.purple, textcolor =_
21     color.white)

```

The `log.*()` functions produce *Pine Logs* results. Every time a script calls any of these functions, the script logs a message in the *Pine Logs* pane, along with a timestamp and navigation options to identify the specific times, chart bars, and lines of code that triggered a log:



```

1 // @version=5
2 indicator("The lay of the land - Pine Logs")
3
4 // @variable The natural logarithm of the current `high - low` range.
5 float logRange = math.log(high - low)
6
7 // Plot the `logRange`.
8 plot(logRange, "logRange")
9
10 if barstate.isconfirmed
11     // Generate an "error" or "info" message on the confirmed bar, depending on
12     // whether `logRange` is defined.

```

(continues on next page)

(continued from previous page)

```

12  switch
13      na(logRange) => log.error("Undefined `logRange` value.")
14      =>          log.info(`logRange` value: " + str.tostring(logRange))
15 else
16     // Generate a "warning" message for unconfirmed values.
17     log.warning("Unconfirmed `logRange` value: " + str.tostring(logRange))

```

One can apply any of the above, or a combination, to establish debugging routines to fit their needs and preferences, depending on the data types and structures they're working with. See the sections below for detailed explanations of various debugging techniques.

### 5.2.3 Numeric values

When creating code in Pine Script™, working with numbers is inevitable. Therefore, to ensure a script works as intended, it's crucial to understand how to inspect the numeric (*int* and *float*) values it receives and calculates.

---

**Note:** This section discusses fundamental *chart-based* approaches for debugging numbers. Scripts can also convert numbers to *strings*, allowing one to inspect numbers using string-related techniques. For more information, see the [Strings](#) and [Pine Logs](#) sections.

---

#### Plotting numbers

One of the most straightforward ways to inspect a script's numeric values is to use `plot*` () functions, which can display results graphically on the chart and show formatted numbers in the script's status line, the price scale, and the Data Window. The locations where a `plot*` () function displays its results depend on the `display` parameter. By default, its value is `display.all`.

---

**Note:** Only a script's *global scope* can contain `plot*` () calls, meaning these functions can only accept global variables and literals. They cannot use variables declared from the local scopes of *loops*, *conditional structures*, or *user-defined functions* and *methods*.

---

The following example uses the `plot()` function to display the 1-bar change in the value of the built-in `time` variable measured in chart timeframes (e.g., a plotted value of 1 on the “1D” chart means there is a one-day difference between the opening times of the current and previous bars). Inspecting this series can help to identify time gaps in a chart's data, which is helpful information when designing time-based indicators.

Since we have not specified a `display` argument, the function uses `display.all`, meaning it will show data in *all* possible locations, as we see below:



```

1 // @version=5
2 indicator("Plotting numbers demo", "Time changes")
3
4 // @variable The one-bar change in the chart symbol's `time` value, measured in units
5 // of the chart timeframe.
6 float timeChange = ta.change(time) / (1000.0 * timeframe.in_seconds())
7
8 // Display the `timeChange` in all possible locations.
9 plot(timeChange, "Time difference (in chart bar units)", color.purple, 3)

```

### Note that:

- The numbers displayed in the script's status line and the Data Window reflect the plotted values at the location of the chart's cursor. These areas will show the latest bar's value when the mouse pointer isn't on the chart.
- The number in the price scale reflects the latest available value on the visible chart.

### Without affecting the scale

When debugging multiple numeric values in a script, programmers may wish to inspect them without interfering with the price scales or cluttering the visual outputs in the chart's pane, as distorted scales and overlapping plots may make it harder to evaluate the results.

A simple way to inspect numbers without adding more visuals to the chart's pane is to change the `display` values in the script's `plot*` calls to other `display.*` variables or expressions using them.

Let's look at a practical example. Here, we've drafted the following script that calculates a custom-weighted moving average by dividing the `sum` of `weight * close` values by the `sum` of the `weight` series:



```

1 //@version=5
2 indicator("Plotting without affecting the scale demo", "Weighted Average", true)
3
4 //@variable The number of bars in the average.
5 int lengthInput = input.int(20, "Length", 1)
6
7 //@variable The weight applied to the price on each bar.
8 float weight = math.pow(close - open, 2)
9
10 //@variable The numerator of the average.
11 float numerator = math.sum(weight * close, lengthInput)
12 //@variable The denominator of the average.
13 float denominator = math.sum(weight, lengthInput)
14
15 //@variable The `lengthInput`-bar weighted average.
16 float average = numerator / denominator
17
18 // Plot the `average`.
19 plot(average, "Weighted Average", linewidth = 3)

```

Suppose we'd like to inspect the variables used in the average calculation to understand and fine-tune the result. If we were to use `plot()` to display the script's `weight`, `numerator`, and `denominator` in all locations, we can no longer easily identify our `average` line on the chart since each variable has a radically different scale:



```

1 //@version=5
2 indicator("Plotting without affecting the scale demo", "Weighted Average", true)
3
4 //@variable The number of bars in the average.
5 int lengthInput = input.int(20, "Length", 1)
6
7 //@variable The weight applied to the price on each bar.
8 float weight = math.pow(close - open, 2)
9
10 //@variable The numerator of the average.
11 float numerator = math.sum(close * weight, lengthInput)
12 //@variable The denominator of the average.
13 float denominator = math.sum(weight, lengthInput)
14
15 //@variable The `lengthInput`-bar weighted average.
16 float average = numerator / denominator
17
18 // Plot the `average`.
19 plot(average, "Weighted Average", linewidth = 3)
20
21 // Create debug plots for the `weight`, `numerator`, and `denominator`.
22 plot(weight, "weight", color.purple)
23 plot(numerator, "numerator", color.teal)
24 plot(denominator, "denominator", color.maroon)

```

While we could hide individual plots from the “Style” tab of the script’s settings, doing so also prevents us from inspecting the results in any other location. To simultaneously view the variables’ values and preserve the scale of our chart, we can change the `display` values in our debug plots.

The version below includes a `debugLocations` variable in the `plot()` calls with a value of `display.all - display.pane` to specify that all locations *except* the chart pane will show the results. Now we can inspect the calculation’s values without the extra clutter:



```

1 //@version=5
2 indicator("Plotting without affecting the scale demo", "Weighted Average", true)
3
4 //@variable The number of bars in the average.
5 int lengthInput = input.int(20, "Length", 1)
6
7 //@variable The weight applied to the price on each bar.
8 float weight = math.pow(close - open, 2)
9
10 //@variable The numerator of the average.
11 float numerator = math.sum(close * weight, lengthInput)
12 //@variable The denominator of the average.
13 float denominator = math.sum(weight, lengthInput)
14
15 //@variable The `lengthInput`-bar weighted average.
16 float average = numerator / denominator
17
18 // Plot the `average`.
19 plot(average, "Weighted Average", linewidth = 3)
20
21 //@variable The display locations of all debug plots.
22 debugLocations = display.all - display.pane
23 // Create debug plots for the `weight`, `numerator`, and `denominator`.
24 plot(weight, "weight", color.purple, display = debugLocations)
25 plot(numerator, "numerator", color.teal, display = debugLocations)
26 plot(denominator, "denominator", color.maroon, display = debugLocations)

```

## From local scopes

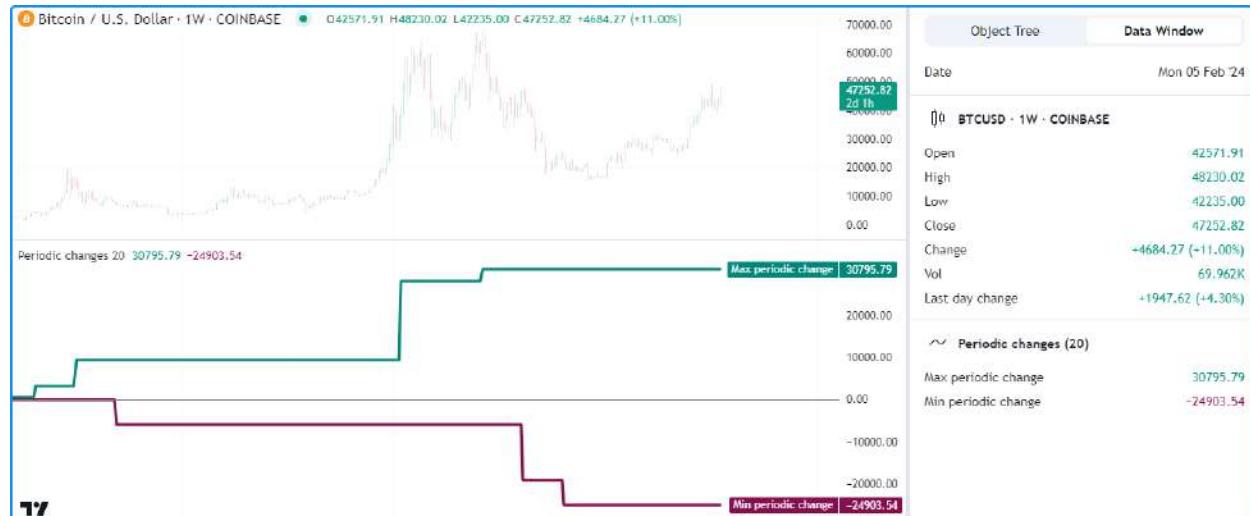
A script's *local scopes* are sections of indented code within *conditional structures*, *loops*, *functions*, and *methods*. When working with variables declared within these scopes, using the `plot*()` functions to display their values directly *will not work*, as plots only work with literals and *global* variables.

To display a local variable's values using plots, one can assign its results to a global variable and pass that variable to the `plot*()` call.

**Note:** The approach described below works for local variables declared within *conditional structures* and *loops*. Employ-

ing a similar process for *functions* and *methods* requires *collections*, *user-defined types*, or other built-in reference types. See the *Debugging functions* section for more information.

For example, this script calculates the all-time maximum and minimum change in the `close` price over a `lengthInput` period. It uses an `if` structure to declare a local `change` variable and update the global `maxChange` and `minChange` once every `lengthInput` bars:



```

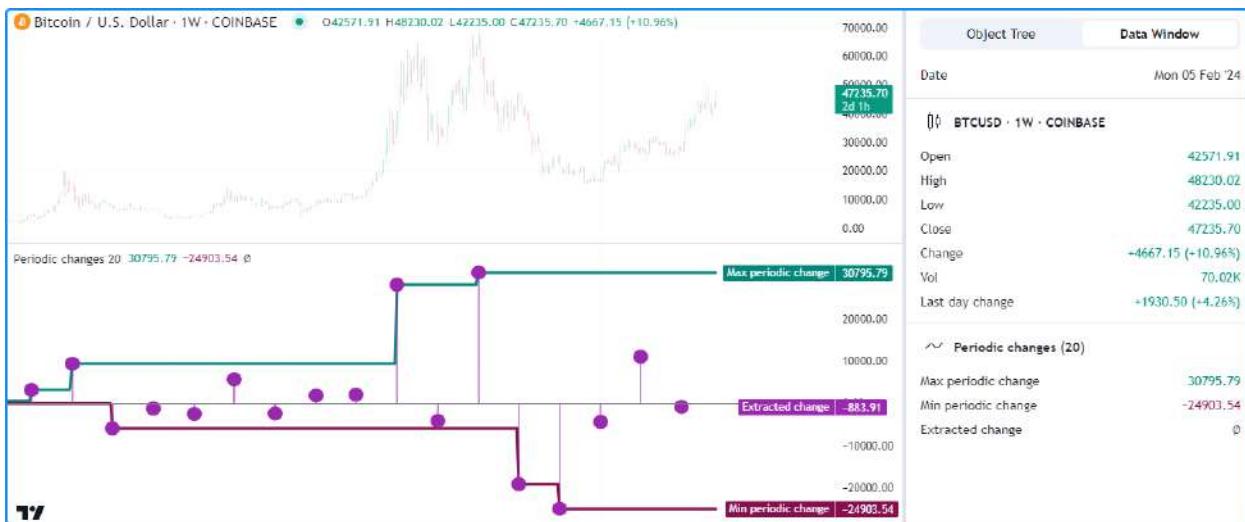
1 //@version=5
2 indicator("Plotting numbers from local scopes demo", "Periodic changes")
3
4 //@variable The number of chart bars in each period.
5 int lengthInput = input.int(20, "Period length", 1)
6
7 //@variable The maximum `close` change over each `lengthInput` period on the chart.
8 var float maxChange = na
9 //@variable The minimum `close` change over each `lengthInput` period on the chart.
10 var float minChange = na
11
12 //@variable Is `true` once every `lengthInput` bars.
13 bool periodClose = bar_index % lengthInput == 0
14
15 if periodClose
16     //@variable The change in `close` prices over `lengthInput` bars.
17     float change = close - close[lengthInput]
18     // Update the global `maxChange` and `minChange`.
19     maxChange := math.max(nz(maxChange, change), change)
20     minChange := math.min(nz(minChange, change), change)
21
22 // Plot the `maxChange` and `minChange`.
23 plot(maxChange, "Max periodic change", color.teal, 3)
24 plot(minChange, "Min periodic change", color.maroon, 3)
25 hline(0.0, color = color.gray, linestyle = hline.style_solid)

```

Suppose we want to inspect the history of the `change` variable using a `plot`. While we cannot plot the variable directly since the script declares it in a local scope, we can assign its value to another *global* variable for use in a `plot()` function.

Below, we've added a `debugChange` variable with an initial value of `na` to the global scope, and the script reassigns its value within the `if` structure using the local `change` variable. Now, we can use `plot()` with the `debugChange` variable

to view the history of available change values:



```

1 // @version=5
2 indicator("Plotting numbers from local scopes demo", "Periodic changes")
3
4 //@variable The number of chart bars in each period.
5 int lengthInput = input.int(20, "Period length", 1)
6
7 //@variable The maximum `close` change over each `lengthInput` period on the chart.
8 var float maxChange = na
9 //@variable The minimum `close` change over each `lengthInput` period on the chart.
10 var float minChange = na
11
12 //@variable Is `true` once every `lengthInput` bars.
13 bool periodClose = bar_index % lengthInput == 0
14
15 //@variable Tracks the history of the local `change` variable.
16 float debugChange = na
17
18 if periodClose
19     //@variable The change in `close` prices over `lengthInput` bars.
20     float change = close - close[lengthInput]
21     // Update the global `maxChange` and `minChange`.
22     maxChange := math.max(nz(maxChange, change), change)
23     minChange := math.min(nz(minChange, change), change)
24     // Assign the `change` value to the `debugChange` variable.
25     debugChange := change
26
27 // Plot the `maxChange` and `minChange`.
28 plot(maxChange, "Max periodic change", color.teal, 3)
29 plot(minChange, "Min periodic change", color.maroon, 3)
30 hline(0.0, color = color.gray, linestyle = hline.style_solid)
31
32 // Create a debug plot to visualize the `change` history.
33 plot(debugChange, "Extracted change", color.purple, 15, plot.style_areabr)

```

#### Note that:

- The script uses `plot.style_areabr` in the debug plot, which doesn't bridge over `na` values as the default style does.

- When the rightmost visible bar's plotted value is `na` the number in the price scale represents the latest *non-na* value before that bar, if one exists.

### With drawings

An alternative approach to graphically inspecting the history of a script's numeric values is to use Pine's *drawing types*, including *lines*, *boxes*, *polylines*, and *labels*.

While Pine drawings don't display results anywhere other than the chart pane, scripts can create them from within *local scopes*, including the scopes of *functions* and *methods* (see the *Debugging functions* section to learn more). Additionally, scripts can position drawings at *any* available chart location, irrespective of the current `bar_index`.

For example, let's revisit the "Periodic changes" script from the *previous section*. Suppose we'd like to inspect the history of the local `change` variable *without* using a plot. In this case, we can avoid declaring a separate global variable and instead create drawing objects directly from the `if` structure's local scope.

The script below is a modification of the previous script that uses *boxes* to visualize the `change` variable's behavior. Inside the scope of the `if` structure, it calls `box.new()` to create a `box` that spans from the bar `lengthInput` bars ago to the current `bar_index`:



```

1 // @version=5
2 indicator("Drawing numbers from local scopes demo", "Periodic changes", max_boxes_
3   ↪ count = 500)
4
5 // @variable The number of chart bars in each period.
6 int lengthInput = input.int(20, "Period length", 1)
7
8 var float maxChange = na
9 // @variable The minimum `close` change over each `lengthInput` period on the chart.
10 var float minChange = na
11
12 // @variable Is `true` once every `lengthInput` bars.
13 bool periodClose = bar_index % lengthInput == 0
14
15 if periodClose
16   // @variable The change in `close` prices over `lengthInput` bars.
17   float change = close - close[lengthInput]
18   // Update the global `maxChange` and `minChange`.

```

(continues on next page)

(continued from previous page)

```

19 maxChange := math.max(nz(maxChange, change), change)
20 minChange := math.min(nz(minChange, change), change)
// @variable Draws a box on the chart to visualize the `change` value.
21 box debugBox = box.new(
22     bar_index - lengthInput, math.max(change, 0.0), bar_index, math.min(change,_
23     ↪ 0.0),
24     color.purple, bgcolor = color.new(color.purple, 80), text = str.
25     ↪ tostring(change)
26 )
27 // Plot the `maxChange` and `minChange`.
28 plot(maxChange, "Max periodic change", color.teal, 3)
29 plot(minChange, "Min periodic change", color.maroon, 3)
30 hline(0.0, color = color.gray, linestyle = hline.style_solid)

```

**Note that:**

- The script includes `max_boxes_count = 500` in the `indicator()` function, which allows it to show up to 500 `boxes` on the chart.
- We used `math.max(change, 0.0)` and `math.min(change, 0.0)` in the `box.new()` function as the `top` and `bottom` values.
- The `box.new()` call includes `str.tostring(change)` as its `text` argument to display a “string” representation of the `change` variable’s “float” value in each `box` drawing. See [this](#) portion of the [Strings](#) section below to learn more about representing data with strings.

For more information about using `boxes` and other related *drawing types*, see our User Manual’s [Lines and boxes](#) page.

## 5.2.4 Conditions

Many scripts one will create in Pine involve declaring and evaluating *conditions* to dictate specific script actions, such as triggering different calculation patterns, visuals, signals, alerts, strategy orders, etc. As such, it’s imperative to understand how to inspect the conditions a script uses to ensure proper execution.

---

**Note:** This section discusses debugging techniques based on chart visuals. To learn about *logging* conditions, see the [Pine Logs](#) section below.

---

### As numbers

One possible way to debug a script’s conditions is to define *numeric values* based on them, which allows programmers to inspect them using numeric approaches, such as those outlined in the [previous section](#).

Let’s look at a simple example. This script calculates the ratio between the `ohlc4` price and the `lengthInput`-bar `moving average`. It assigns a condition to the `priceAbove` variable that returns `true` whenever the value of the ratio exceeds 1 (i.e., the price is above the average).

To inspect the occurrences of the condition, we created a `debugValue` variable assigned to the result of an expression that uses the ternary `?:` operator to return 1 when `priceAbove` is `true` and 0 otherwise. The script plots the variable’s value in all available locations:



```

1 //@version=5
2 indicator("Conditions as numbers demo", "MA signal")
3
4 //@variable The number of bars in the moving average calculation.
5 int lengthInput = input.int(20, "Length", 1)
6
7 //@variable The ratio of the `ohlc4` price to its `lengthInput`-bar moving average.
8 float ratio = ohlc4 / ta.sma(ohlc4, lengthInput)
9
10 //@variable The condition to inspect. Is `true` when `ohlc4` is above its moving_
11 //→average, `false` otherwise.
12 bool priceAbove = ratio > 1.0
13 //@variable Returns 1 when the `priceAbove` condition is `true`, 0 otherwise.
14 int debugValue = priceAbove ? 1 : 0
15
16 // Plot the `debugValue`.
17 plot(debugValue, "Conditional number", color.teal, 3)

```

### Note that:

- Representing “bool” values using numbers also allows scripts to display conditional shapes or characters at specific y-axis locations with `plotshape()` and `plotchar()`, and it facilitates conditional debugging with `plotarrow()`. See the [next section](#) to learn more.

### Plotting conditional shapes

The `plotshape()` and `plotchar()` functions provide utility for debugging conditions, as they can plot shapes or characters at absolute or relative chart locations whenever they contain a `true` or `non-na` series argument.

These functions can also display *numeric* representations of the `series` in the script’s status line and the Data Window, meaning they’re also helpful for debugging `numbers`. We show a simple, practical way to debug numbers with these functions in the [Tips](#) section.

The chart locations of the plots depend on the `location` parameter, which is `location.abovebar` by default.

---

**Note:** When using `location.abovebar` or `location.belowbar`, the function positions the shapes/characters relative to the *main chart* prices. If the script plots its values in a separate chart pane, we recommend debugging with other `location` options to avoid affecting the pane’s scale.

Let's inspect a condition using these functions. The following script calculates an RSI with a `lengthInput` length and a `crossBelow` variable whose value is the result of a condition that returns `true` when the RSI crosses below 30. It calls `plotshape()` to display a circle near the top of the pane each time the condition occurs:



```

1 //@version=5
2 indicator("Conditional shapes demo", "RSI cross under 30")
3
4 //@variable The length of the RSI.
5 int lengthInput = input.int(14, "Length", 1)
6
7 //@variable The calculated RSI value.
8 float rsi = ta.rsi(close, lengthInput)
9
10 //@variable Is `true` when the `rsi` crosses below 30, `false` otherwise.
11 bool crossBelow = ta.crossover(rsi, 30.0)
12
13 // Plot the `rsi`.
14 plot(rsi, "RSI", color.rgb(136, 76, 146), linewidth = 3)
15 // Plot the `crossBelow` condition as circles near the top of the pane.
16 plotshape(crossBelow, "RSI crossed below 30", shape.circle, location.top, color.red,_
    size = size.small)

```

### Note that:

- The status line and Data Window show a value of 1 when `crossBelow` is `true` and 0 when it's `false`.

Suppose we'd like to display the shapes at *precise* locations rather than relative to the chart pane. We can achieve this by using `conditional numbers` and `location.absolute` in the `plotshape()` call.

In this example, we've modified the previous script by creating a `debugNumber` variable that returns the `rsi` value when `crossBelow` is `true` and `na` otherwise. The `plotshape()` function uses this new variable as its `series` argument and `location.absolute` as its `location` argument:



```

1 // @version=5
2 indicator("Conditional shapes demo", "RSI cross under 30")
3
4 // @variable The length of the RSI.
5 int lengthInput = input.int(14, "Length", 1)
6
7 // @variable The calculated RSI value.
8 float rsi = ta.rsi(close, lengthInput)
9
10 // @variable Is `true` when the `rsi` crosses below 30, `false` otherwise.
11 bool crossBelow = ta.crossover(rsi, 30.0)
12 // @variable Returns the `rsi` when `crossBelow` is `true`, `na` otherwise.
13 float debugNumber = crossBelow ? rsi : na
14
15 // Plot the `rsi`.
16 plot(rsi, "RSI", color.rgb(136, 76, 146), linewidth = 3)
17 // Plot circles at the `debugNumber`.
18 plotshape(debugNumber, "RSI when it crossed below 30", shape.circle, location.
    ↪absolute, color.red, size = size.small)

```

### Note that:

- Since we passed a *numeric* series to the function, our conditional plot now shows the values of the `debugNumber` in the status line and Data Window instead of 1 or 0.

Another handy way to debug conditions is to use `plotarrow()`. This function plots an arrow with a location relative to the *main chart prices* whenever the *series* argument is nonzero and not `na`. The length of each arrow varies with the *series* value supplied. As with `plotshape()` and `plotchar()`, `plotarrow()` can also display numeric results in the status line and the Data Window.

---

**Note:** Since this function always positions arrows relative to the main chart prices, we recommend only using it if the script occupies the main chart pane to avoid otherwise interfering with the scale.

---

This example shows an alternative way to inspect our `crossBelow` condition using `plotarrow()`. In this version, we've set `overlay` to `true` in the `indicator()` function and added a `plotarrow()` call to visualize the conditional values. The `debugNumber` in this example measures how far the `rsi` dropped below 30 each time the condition occurs:



```

1 // @version=5
2 indicator("Conditional shapes demo", "RSI cross under 30", true)
3
4 //@variable The length of the RSI.
5 int lengthInput = input.int(14, "Length", 1)
6
7 //@variable The calculated RSI value.
8 float rsi = ta.rsi(close, lengthInput)
9
10 //@variable Is `true` when the `rsi` crosses below 30, `false` otherwise.
11 bool crossBelow = ta.crossover(rsi, 30.0)
12 //@variable Returns `rsi - 30.0` when `crossBelow` is `true`, `na` otherwise.
13 float debugNumber = crossBelow ? rsi - 30.0 : na
14
15 // Plot the `rsi`.
16 plot(rsi, "RSI", color.rgb(136, 76, 146), display = display.data_window)
17 // Plot circles at the `debugNumber`.
18 plotarrow(debugNumber, "RSI cross below 30 distnce")

```

#### Note that:

- We set the `display` value in the `plot()` of the `rsi` to `display.data_window` to *preserve the chart's scale*.

To learn more about `plotshape()`, `plotchar()`, and `plotarrow()`, see this manual's *Text and shapes* page.

#### Conditional colors

An elegant way to visually represent conditions in Pine is to create expressions that return `color` values based on `true` or `false` states, as scripts can use them to control the appearance of `drawing objects` or the results of `plot*`(), `fill()`, `bgcolor()`, or `barcolor()` calls.

---

**Note:** As with `plot*`() functions, scripts can only call `fill()`, `bgcolor()` and `barcolor()` from the *global scope*, and the functions cannot accept any local variables.

---

For example, this script calculates the change in `close` prices over `lengthInput` bars and declares two “bool” variables to identify when the price change is positive or negative.

The script uses these “bool” values as conditions in `ternary` expressions to assign the values of three “color” variables, then uses those variables as the `color` arguments in `plot()`, `bgcolor()`, and `barcolor()` to debug the results:



```

1 // @version=5
2 indicator("Conditional colors demo", "Price change colors")
3
4 // @variable The number of bars in the price change calculation.
5 int lengthInput = input.int(10, "Length", 1)
6
7 // @variable The change in `close` prices over `lengthInput` bars.
8 float priceChange = ta.change(close, lengthInput)
9
10 // @variable Is `true` when the `priceChange` is a positive value, `false` otherwise.
11 bool isPositive = priceChange > 0
12 // @variable Is `true` when the `priceChange` is a negative value, `false` otherwise.
13 bool isNegative = priceChange < 0
14
15 // @variable Returns a color for the `priceChange` plot to show when `isPositive`, `isNegative`, or neither occurs.
16 color plotColor = isPositive ? color.teal : isNegative ? color.maroon : chart.fg_color
17 // @variable Returns an 80% transparent color for the background when `isPositive` or `isNegative`, `na` otherwise.
18 color bgColor = isPositive ? color.new(color.aqua, 80) : isNegative ? color.new(color.
19 // @variable Returns a color to emphasize chart bars when `isPositive` occurs. Otherwise, returns the `chart.bg_color`.
20 color barColor = isPositive ? color.orange : chart.bg_color
21
22 // Plot the `priceChange` and color it with the `plotColor`.
23 plot(priceChange, "Price change", plotColor, style = plot.style_area)
24 // Highlight the pane's background with the `bgColor`.
25 bgcolor(bgColor, title = "Background highlight")
26 // Emphasize the chart bars with positive price change using the `barColor`.
27 barcolor(barColor, title = "Positive change bars")

```

### Note that:

- The `barcolor()` function always colors the main chart's bars, regardless of whether the script occupies another chart pane, and the chart will only display the results if the bars are visible.

See the [Colors](#), [Fills](#), [Backgrounds](#), and [Bar coloring](#) pages for more information about working with colors, filling plots, highlighting backgrounds, and coloring bars.

## Using drawings

Pine Script™'s *drawing types* provide flexible ways to visualize conditions on the chart, especially when the conditions are within local scopes.

Consider the following script, which calculates a custom `filter` with a smoothing parameter (`alpha`) that changes its value within an `if` structure based on recent `volume` conditions:



```

1 // @version=5
2 indicator("Conditional drawings demo", "Volume-based filter", true)
3
4 //@variable The number of bars in the volume average.
5 int lengthInput = input.int(20, "Volume average length", 1)
6
7 //@variable The average `volume` over `lengthInput` bars.
8 float avgVolume = ta.sma(volume, lengthInput)
9
10 //@variable A custom price filter based on volume activity.
11 float filter = close
12 //@variable The smoothing parameter of the filter calculation. Its value depends on
13 //→ multiple volume conditions.
14 float alpha = na
15
16 // Set the `alpha` to 1 if `volume` exceeds its `lengthInput`-bar moving average.
17 if volume > avgVolume
18     alpha := 1.0
19 // Set the `alpha` to 0.5 if `volume` exceeds its previous value.
20 else if volume > volume[1]
21     alpha := 0.5
22 // Set the `alpha` to 0.01 otherwise.
23 else
24     alpha := 0.01
25
26 // Calculate the new `filter` value.
27 filter := (1.0 - alpha) * nz(filter[1], filter) + alpha * close
28
29 // Plot the `filter`.
30 plot(filter, "Filter", linewidth = 3)

```

Suppose we'd like to inspect the conditions that control the `alpha` value. There are several ways we could approach the task with chart visuals. However, some approaches will involve more code and careful handling.

For example, to visualize the `if` structure's conditions using *plotted shapes* or *background colors*, we'd have to create additional variables or expressions in the global scope for the `plot*` or `bgcolor()` functions to access.

Alternatively, we can use *drawing types* to visualize the conditions concisely without those extra steps.

The following is a modification of the previous script that calls `label.new()` within specific branches of the *conditional structure* to draw *labels* on the chart whenever those branches execute. These simple changes allow us to identify those conditions on the chart without much extra code:



```

1 // @version=5
2 indicator("Conditional drawings demo", "Volume-based filter", true, max_labels_count=500)
3
4 // @variable The number of bars in the volume average.
5 int lengthInput = input.int(20, "Volume average length", 1)
6
7 // @variable The average `volume` over `lengthInput` bars.
8 float avgVolume = ta.sma(volume, lengthInput)
9
10 // @variable A custom price filter based on volume activity.
11 float filter = close
12 // @variable The smoothing parameter of the filter calculation. Its value depends on
13 // multiple volume conditions.
14 float alpha = na
15
16 // Set the `alpha` to 1 if `volume` exceeds its `lengthInput`-bar moving average.
17 if volume > avgVolume
18     // Add debug label.
19     label.new(chart.point.now(high), "alpha = 1", color = color.teal, textcolor = color.white)
20     alpha := 1.0
21 // Set the `alpha` to 0.5 if `volume` exceeds its previous value.
22 else if volume > volume[1]
23     // Add debug label.
24     label.new(chart.point.now(high), "alpha = 0.5", color = color.green, textcolor = color.white)
25     alpha := 0.5
26 // Set the `alpha` to 0.01 otherwise.
27 else
28     alpha := 0.01
29
30 // Calculate the new `filter` value.

```

(continues on next page)

(continued from previous page)

```

30 filter := (1.0 - alpha) * nz(filter[1], filter) + alpha * close
31
32 // Plot the `filter`.
33 plot(filter, "Filter", linewidth = 3)

```

**Note that:**

- We added the `label.new()` calls *above* the `alpha` reassignment expressions, as the returned types of each branch in the `if` structure must match.
- The `indicator()` function includes `max_labels_count = 500` to specify that the script can show up to 500 *labels* on the chart.

**Compound and nested conditions**

When a programmer needs to identify situations where more than one condition can occur, they may construct *compound conditions* by aggregating individual conditions with logical operators (`and`, `or`).

For example, this line of code shows a `compoundCondition` variable that only returns `true` if `condition1` and either `condition2` or `condition3` occurs:

```
bool compoundCondition = condition1 and (condition2 or condition3)
```

One may alternatively create *nested conditions* using *conditional structures* or *ternary expressions*. For example, this `if` structure assigns `true` to the `nestedCondition` variable if `condition1` and `condition2` or `condition3` occurs. However, unlike the logical expression above, the branches of this structure also allow the script to execute additional code before assigning the “`bool`” value:

```

bool nestedCondition = false

if condition1
    // [additional_code]
    if condition2
        // [additional_code]
        nestedCondition := true
    else if condition3
        // [additional_code]
        nestedCondition := true

```

In either case, whether working with compound or nested conditions in code, one will save many headaches and ensure they work as expected by validating the behaviors of the *individual conditions* that compose them.

For example, this script calculates an `rsi` and the median of the `rsi` over `lengthInput` bars. Then, it creates five variables to represent different singular conditions. The script uses these variables in a logical expression to assign a “`bool`” value to the `compoundCondition` variable, and it displays the results of the `compoundCondition` using a *conditional background color*:



```

1 // @version=5
2 indicator("Compound conditions demo")
3
4 // @variable The length of the RSI and median RSI calculations.
5 int lengthInput = input.int(14, "Length", 2)
6
7 // @variable The `lengthInput`-bar RSI.
8 float rsi = ta.rsi(close, lengthInput)
9 // @variable The `lengthInput`-bar median of the `rsi`.
10 float median = ta.median(rsi, lengthInput)
11
12 // @variable Condition #1: Is `true` when the 1-bar `rsi` change switches from 1 to -1.
13 bool changeNegative = ta.change(math.sign(ta.change(rsi))) == -2
14 // @variable Condition #2: Is `true` when the previous bar's `rsi` is greater than 70.
15 bool prevAbove70 = rsi[1] > 70.0
16 // @variable Condition #3: Is `true` when the current `close` is lower than the
17 // previous bar's `open`.
18 bool closeBelow = close < open[1]
19 // @variable Condition #4: Is `true` when the `rsi` is between 60 and 70.
20 bool betweenLevels = bool(math.max(70.0 - rsi, 0.0) * math.max(rsi - 60.0, 0.0))
21 // @variable Condition #5: Is `true` when the `rsi` is above the `median`.
22 bool aboveMedian = rsi > median
23
24 // @variable Is `true` when the first condition occurs alongside conditions 2 and 3 or
25 // 4 and 5.
26 bool compoundCondition = changeNegative and ((prevAbove70 and closeBelow) or
27 // (betweenLevels and aboveMedian))
28
29 // Plot the `rsi` and the `median`.
30 plot(rsi, "RSI", color.rgb(201, 109, 34), 3)
31 plot(median, "RSI Median", color.rgb(180, 160, 102), 2)
32
33 // Highlight the background red when the `compoundCondition` occurs.
34 bgcolor(compoundCondition ? color.new(color.red, 60) : na, title = "compoundCondition")

```

As we see above, it's not necessarily easy to understand the behavior of the `compoundCondition` by only visualizing its end result, as five underlying singular conditions determine the final value. To effectively debug the `compoundCondition` in this case, we must also inspect the conditions that compose it.

In the example below, we've added five `plotchar()` calls to display *characters* on the chart and numeric values in the status line and Data Window when each singular condition occurs. Inspecting each of these results provides us with more complete information about the `compoundCondition`'s behavior:



```

1 //@version=5
2 indicator("Compound conditions demo")
3
4 //@variable The length of the RSI and median RSI calculations.
5 int lengthInput = input.int(14, "Length", 2)
6
7 //@variable The `lengthInput`-bar RSI.
8 float rsi = ta.rsi(close, lengthInput)
9 //@variable The `lengthInput`-bar median of the `rsi`.
10 float median = ta.median(rsi, lengthInput)
11
12 //@variable Condition #1: Is `true` when the 1-bar `rsi` change switches from 1 to -1.
13 bool changeNegative = ta.change(math.sign(ta.change(rsi))) == -2
14 //@variable Condition #2: Is `true` when the previous bar's `rsi` is greater than 70.
15 bool prevAbove70 = rsi[1] > 70.0
16 //@variable Condition #3: Is `true` when the current `close` is lower than the
17 //← previous bar's `open`.
18 bool closeBelow = close < open[1]
19 //@variable Condition #4: Is `true` when the `rsi` is between 60 and 70.
20 bool betweenLevels = bool(math.max(70.0 - rsi, 0.0) * math.max(rsi - 60.0, 0.0))
21 //@variable Condition #5: Is `true` when the `rsi` is above the `median`.
22 bool aboveMedian = rsi > median
23
24 //@variable Is `true` when the first condition occurs alongside conditions 2 and 3 or
25 //← 4 and 5.
26 bool compoundCondition = changeNegative and ((prevAbove70 and closeBelow) or
27 //← (betweenLevels and aboveMedian))
28
29 //Plot the `rsi` and the `median`.
30 plot(rsi, "RSI", color.rgb(201, 109, 34), 3)
31 plot(median, "RSI Median", color.rgb(180, 160, 102), 2)
32
33 // Highlight the background red when the `compoundCondition` occurs.
34 bgcolor(compoundCondition ? color.new(color.red, 60) : na, title = "compoundCondition")

```

(continues on next page)

(continued from previous page)

```

32 // Plot characters on the chart when conditions 1-5 occur.
33 plotchar(changeNegative ? rsi : na, "changeNegative (1)", "1", location.absolute,
34   ↵chart.fg_color)
35 plotchar(prevAbove70 ? 70.0 : na, "prevAbove70 (2)", "2", location.absolute, chart.fg_
36   ↵color)
37 plotchar(closeBelow ? close : na, "closeBelow (3)", "3", location.bottom, chart.fg_
38   ↵color)
39 plotchar(betweenLevels ? 60 : na, "betweenLevels (4)", "4", location.absolute, chart.
40   ↵fg_color)
41 plotchar(aboveMedian ? median : na, "aboveMedian (5)", "5", location.absolute, chart.
42   ↵fg_color)

```

**Note that:**

- Each `plotchar()` call uses a *conditional number* as the `series` argument. The functions display the numeric values in the status line and Data Window.
- All the `plotchar()` calls, excluding the one for the `closeBelow` condition, use `location.absolute` as the `location` argument to display characters at precise locations whenever their `series` is not `na` (i.e., the condition occurs). The call for `closeBelow` uses `location.bottom` to display its characters near the bottom of the pane.
- In this section's examples, we assigned individual conditions to separate variables with straightforward names and annotations. While this format isn't required to create a compound condition since one can combine conditions directly within a logical expression, it makes for more readable code that's easier to debug, as explained in the [Tips](#) section.

## 5.2.5 Strings

*Strings* are sequences of alphanumeric, control, and other characters (e.g., Unicode). They provide utility when debugging scripts, as programmers can use them to represent a script's data types as human-readable text and inspect them with *drawing types* that have text-related properties, or by using [Pine Logs](#).

---

**Note:** This section discusses “string” conversions and inspecting strings via `labels` and `tables`. `Boxes` can also display text. However, their utility for debugging strings is more limited than the techniques covered in this section and the [Pine Logs](#) section below.

---

### Representing other types

Users can create “string” representations of virtually any data type, facilitating effective debugging when other approaches may not suffice. Before exploring “string” inspection techniques, let’s briefly review ways to *represent* a script’s data using strings.

Pine Script™ includes predefined logic to construct “string” representations of several other built-in types, such as `int`, `float`, `bool`, `array`, and `matrix`. Scripts can conveniently represent such types as strings via the `str.tostring()` and `str.format()` functions.

For example, this snippet creates strings to represent multiple values using these functions:

```

//@variable Returns: "1.25"
string floatRepr = str.tostring(1.25)
//@variable Returns: "1"

```

(continues on next page)

(continued from previous page)

```

string rounded0 = str.tostring(1.25, "#")
//@variable Returns: "1.3"
string rounded1 = str.tostring(1.25, "#.#")
//@variable Returns: "1.2500"
string trailingZeros = str.tostring(1.25, "#.0000")
//@variable Returns: "true"
string trueRepr = str.tostring(true)
//@variable Returns: "false"
string falseRepr = str.tostring(5 == 3)
//@variable Returns: "[1, 2, -3.14]"
string floatArrayRepr = str.tostring(array.from(1, 2.0, -3.14))
//@variable Returns: "[2, 20, 0]"
string roundedArrayRepr = str.tostring(array.from(2.22, 19.6, -0.43), "#")
//@variable Returns: "[Hello, World, !]"
string stringArrayRepr = str.tostring(array.from("Hello", "World", "!"))
//@variable Returns: "Test: 2.718 ^ 2 > 5: true"
string mixedTypeRepr = str.format("{0}{1, number, #.###} ^ 2 > {2}: {3}", "Test: ", ↵
    math.e, 5, math.e * math.e > 5)

//@variable Combines all the above strings into a multi-line string.
string combined = str.format(
    "{0}\n{1}\n{2}\n{3}\n{4}\n{5}\n{6}\n{7}\n{8}\n{9}",
    floatRepr, rounded0, rounded1, trailingZeros, trueRepr,
    falseRepr, floatArrayRepr, roundedArrayRepr, stringArrayRepr,
    mixedTypeRepr
)

```

When working with “int” values that symbolize UNIX timestamps, such as those returned from time-related functions and variables, one can also use `str.format()` or `str.format_time()` to convert them to human-readable date strings. This code block demonstrates multiple ways to convert a timestamp using these functions:

```

//@variable A UNIX timestamp, in milliseconds.
int unixTime = 1279411200000

//@variable Returns: "2010-07-18T00:00:00+0000"
string default = str.format_time(unixTime)
//@variable Returns: "2010-07-18"
string ymdRepr = str.format_time(unixTime, "yyyy-MM-dd")
//@variable Returns: "07-18-2010"
string mdyRepr = str.format_time(unixTime, "MM-dd-yyyy")
//@variable Returns: "20:00:00, 2010-07-17"
string hmsymdRepr = str.format_time(unixTime, "HH:mm:ss, yyyy-MM-dd", "America/New_"
    ↵ York")
//@variable Returns: "Year: 2010, Month: 07, Day: 18, Time: 12:00:00"
string customFormat = str.format(
    "Year: {0, time, yyyy}, Month: {1, time, MM}, Day: {2, time, dd}, Time: {3, time, "
    ↵ hh:mm:ss}",
    unixTime, unixTime, unixTime, unixTime
)

```

When working with types that *don't* have built-in “string” representations, e.g., `color`, `map`, `user-defined types`, etc., programmers can use custom logic or formatting to construct representations. For example, this code calls `str.format()` to represent a “color” value using its `r`, `g`, `b`, and `t` components:

```

//@variable The built-in `color.maroon` value with 17% transparency.
color myColor = color.new(color.maroon, 17)

```

(continues on next page)

(continued from previous page)

```
// Get the red, green, blue, and transparency components from `myColor`.
float r = color.r(myColor)
float g = color.g(myColor)
float b = color.b(myColor)
float t = color.t(myColor)

//@variable Returns: "color (r = 136, g = 14, b = 79, t = 17)"
string customRepr = str.format("color (r = {0}, g = {1}, b = {2}, t = {3})", r, g, b, t)
```

There are countless ways one can represent data using strings. When choosing string formats for debugging, ensure the results are **readable** and provide enough information for proper inspection. The following segments explain ways to validate strings by displaying them on the chart using *labels* or *tables*, and the section after these segments explains how to display strings as messages in the *Pine Logs* pane.

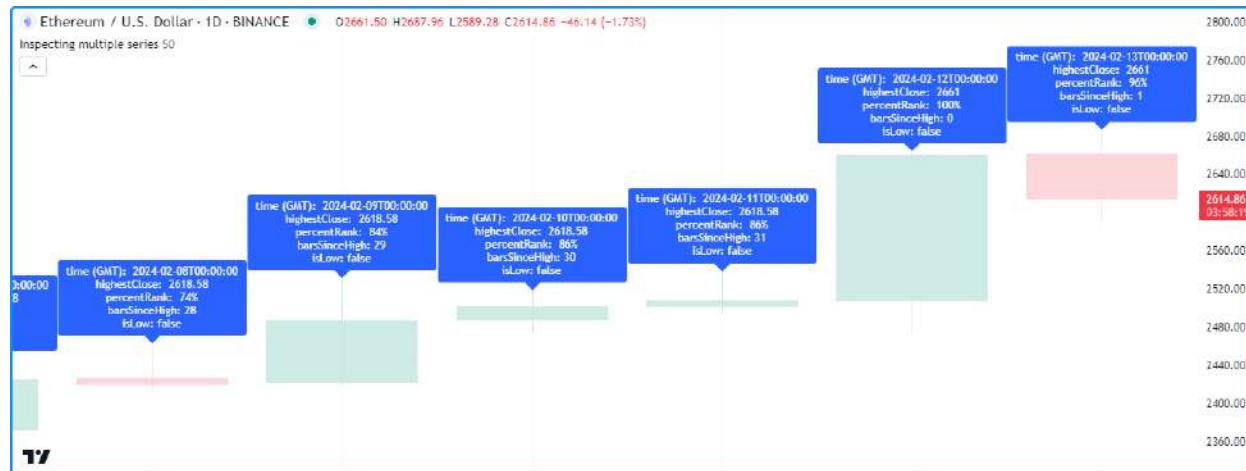
## Using labels

*Labels* allow scripts to display dynamic text (“series strings”) at any available location on the chart. Where to display such text on the chart depends on the information the programmer wants to inspect and their debugging preferences.

### On successive bars

When inspecting the history of values that affect the chart’s scale or working with multiple series that have different types, a simple, handy debugging approach is to draw *labels* that display *string representations* on successive bars.

For example, this script calculates four series: highestClose, percentRank, barsSinceHigh, and isLow. It uses `str.format()` to create a formatted “string” representing the series values and a timestamp, then it calls `label.new()` to draw a *label* that display the results at the `high` on each bar:



```
1 // @version=5
2 indicator("Labels on successive bars demo", "Inspecting multiple series", true, max_
3 →labels_count = 500)
4
5 // @variable The number of bars in the calculation window.
6 int lengthInput = input.int(50, "Length", 1)
```

(continues on next page)

(continued from previous page)

```

7 // @variable The highest `close` over `lengthInput` bars.
8 float highestClose = ta.highest(close, lengthInput)
9 // @variable The percent rank of the current `close` compared to previous values over
10 // `lengthInput` bars.
11 float percentRank = ta.percentrank(close, lengthInput)
12 int barsSinceHigh = ta.barssince(close == highestClose)
13 // @variable Is `true` when the `percentRank` is 0, i.e., when the `close` is the
14 // lowest.
15 bool isLow = percentRank == 0.0
16
17 // @variable A multi-line string representing the `time`, `highestClose`, `percentRank`,
18 // `barsSinceHigh`, and `isLow`.
19 string debugString = str.format(
20     "time (GMT): {0}, time, yyyy-MM-dd'T'HH:mm:ss}\nhighestClose: {1, number, #.####}\n"
21     "\npercentRank: {2, number, #.##}\n\nbarsSinceHigh: {3, number, integer}\nisLow:\n"
22     "{4}",
23     time, highestClose, percentRank, barsSinceHigh, isLow
24 )
25
26 // @variable Draws a label showing the `debugString` at each bar's `high`.
27 label debugLabel = label.new(chart.point.now(high), debugString, textcolor = color.
28     white)

```

While the above example allows one to inspect the results of the script's series on any bar with a `label` drawing, consecutive drawings like these can clutter the chart, especially when viewing longer strings.

An alternative, more visually compact way to inspect successive bars' values with `labels` is to utilize the `tooltip` property instead of the `text` property, as a `label` will only show its tooltip when the cursor *hovers* over it.

Below, we've modified the previous script by using the `debugString` as the `tooltip` argument instead of the `text` argument in the `label.new()` call. Now, we can view the results on specific bars without the extra noise:



```

1 // @version=5
2 indicator("Tooltips on successive bars demo", "Inspecting multiple series", true, max_
3 // labels_count = 500)
4
5 // @variable The number of bars in the calculation window.
int lengthInput = input.int(50, "Length", 1)

```

(continues on next page)

(continued from previous page)

```

6 // @variable The highest `close` over `lengthInput` bars.
7 float highestClose = ta.highest(close, lengthInput)
8 // @variable The percent rank of the current `close` compared to previous values over
9 // `lengthInput` bars.
10 float percentRank = ta.percentrank(close, lengthInput)
11 // @variable The number of bars since the `close` was equal to the `highestClose`.
12 int barsSinceHigh = ta.barssince(close == highestClose)
13 // @variable Is `true` when the `percentRank` is 0, i.e., when the `close` is the
14 // lowest.
15 bool isLow = percentRank == 0.0
16
17 // @variable A multi-line string representing the `time`, `highestClose`,
18 // `percentRank`, `barsSinceHigh`, and `isLow`.
19 string debugString = str.format(
20     "time (GMT): {0, time, yyyy-MM-dd'T'HH:mm:ss}\nhighestClose: {1, number, #.####}\n"
21     "\npercentRank: {2, number, #.##}%\nbarsSinceHigh: {3, number, integer}\nisLow: "
22     "{4}",
23     time, highestClose, percentRank, barsSinceHigh, isLow
24 )
25
26 // @variable Draws a label showing the `debugString` in a tooltip at each bar's `high`.
27 label debugLabel = label.new(chart.point.now(high), tooltip = debugString)

```

It's important to note that a script can display up to 500 `label` drawings, meaning the above examples will only allow users to inspect the strings from the most recent 500 chart bars.

If a programmer wants to see the results from *earlier* chart bars, one approach is to create conditional logic that only allows drawings within a specific time range, e.g.:

```

if time >= startTime and time <= endTime
    <create_drawing_id>

```

If we use this structure in our previous example with `chart.left_visible_bar_time` and `chart.right_visible_bar_time` as the `startTime` and `endTime` values, the script will only create *labels* on **visible chart bars** and avoid drawing on others. With this logic, we can scroll to view labels on *any* chart bar, as long as there are up to `max_labels_count` bars in the visible range:



```

1 // @version=5
2 indicator("Tooltips on visible bars demo", "Inspecting multiple series", true, max_

```

(continues on next page)

(continued from previous page)

```

3     ↵labels_count = 500)
4
5 // @variable The number of bars in the calculation window.
6 int lengthInput = input.int(50, "Length", 1)
7
8 // @variable The highest `close` over `lengthInput` bars.
9 float highestClose = ta.highest(close, lengthInput)
10 // @variable The percent rank of the current `close` compared to previous values over
11 // `lengthInput` bars.
12 float percentRank = ta.percentrank(close, lengthInput)
13 // @variable The number of bars since the `close` was equal to the `highestClose`.
14 int barsSinceHigh = ta.barssince(close == highestClose)
15 // @variable Is `true` when the `percentRank` is 0, i.e., when the `close` is the
16 // lowest.
17 bool isLow = percentRank == 0.0
18
19 // @variable A multi-line string representing the `time`, `highestClose`,
20 // `percentRank`, `barsSinceHigh`, and `isLow`.
21 string debugString = str.format(
22     "time (GMT): {0, time, yyyy-MM-dd'T'HH:mm:ss}\nhighestClose: {1, number, #.####}"
23     "\npercentRank: {2, number, #.##}%\nbarsSinceHigh: {3, number, integer}\nisLow:"
24     "{4}",
25     time, highestClose, percentRank, barsSinceHigh, isLow
26 )
27
28 if time >= chart.left_visible_bar_time and time <= chart.right_visible_bar_time
29     // @variable Draws a label showing the `debugString` in a tooltip at each visible
30     // bar's `high`.
31     label debugLabel = label.new(chart.point.now(high), tooltip = debugString)

```

**Note that:**

- If the visible chart contains more bars than allowed drawings, the script will only show results on the latest bars in the visible range. For best results with this technique, zoom on the chart to keep the visible range limited to the allowed number of drawings.

**At the end of the chart**

A frequent approach to debugging a script's strings with `label` is to display them at the *end* of the chart, namely when the strings do not change or when only a specific bar's values require analysis.

The script below contains a user-defined `printLabel()` function that draws a `label` at the last available time on the chart, regardless of when the script calls it. We've used the function in this example to display a “Hello world!” string, some basic chart information, and the data feed's current OHLCV values:



```

1 //@version=5
2 indicator("Labels at the end of the chart demo", "Chart info", true)
3
4 //@function      Draws a label to print the `txt` at the last available time on the
5 //          chart.
6 //          When called from the global scope, the label updates its text using
7 //          the specified `txt` on every bar.
8 //param txt      The string to display on the chart.
9 //param price    The optional y-axis location of the label. If not specified, draws
10 //               the label above the last chart bar.
11 //returns        The resulting label ID.
12 printLabel(string txt, float price = na) =>
13     int labelTime = math.max(last_bar_time, chart.right_visible_bar_time)
14     var label result = label.new(
15         labelTime, na, txt, xloc.bar_time, na(price) ? yloc.abovebar : yloc.price,
16         na,
17             label.style_none, chart.fg_color, size.large
18     )
19     label.set_text(result, txt)
20     label.set_y(result, price)
21     result
22
23 //@variable A formatted string containing information about the current chart.
24 string chartInfo = str.format(
25     "Symbol: {0}:{1}\nTimeframe: {2}\nStandard chart: {3}\nReplay active: {4}",
26     syminfo.prefix, syminfo.ticker, timeframe.period, chart.is_standard,
27     str.contains(syminfo.tickerid, "replay")
28 )
29
30 //@variable A formatted string containing OHLCV values.
31 string ohlcvInfo = str.format(
32     "O: {0, number, #.#####}, H: {1, number, #.#####}, L: {2, number, #.#####}, C:
33     {3, number, #.#####}, V: {4}",
34     open, high, low, close, str.tostring(volume, format.volume)
35 )
36
37 // Print "Hello world!" and the `chartInfo` at the end of the chart on the first bar.
38 if barstate.isfirst
39     printLabel("Hello world!" + "\n\n\n\n\n\n\n")
40     printLabel(chartInfo + "\n\n")

```

(continues on next page)

(continued from previous page)

```

37 // Print current `ohlcInfo` at the end of the chart, updating the displayed text as
38 // new data comes in.
printLabel(ohlcInfo)

```

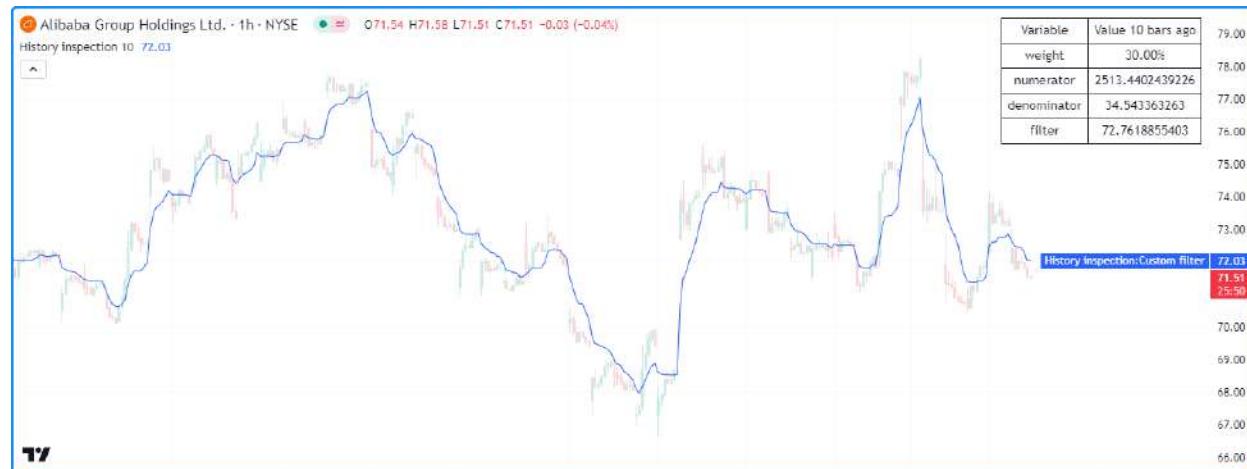
**Note that:**

- The `printLabel()` function sets the x-coordinate of the drawn `label` using the `max` of the `last_bar_time` and the `chart.right_visible_bar_time` to ensure it always shows the results at the last available bar.
- When called from the *global scope*, the function creates a `label` with `text` and `y` properties that update on every bar.
- We've made three calls to the function and added linefeed characters (`\n`) to demonstrate that users can superimpose the results from multiple `labels` at the end of the chart if the strings have adequate line spacing.

**Using tables**

*Tables* display strings within cells arranged in columns and rows at fixed locations relative to a chart pane's visual space. They can serve as versatile chart-based debugging tools, as unlike *labels*, they allow programmers to inspect one or *more* "series strings" in an organized visual structure agnostic to the chart's scale or bar index.

For example, this script calculates a custom `filter` whose result is the ratio of the `EMA` of weighted `close` prices to the `EMA` of the `weight` series. For inspection of the variables used in the calculation, it creates a `table` instance on the first bar, initializes the table's cells on the last historical bar, then updates necessary cells with "string" representations of the values from `barsBack` bars ago on the latest chart bar:



```

1 //@version=5
2 indicator("Debugging with tables demo", "History inspection", true)
3
4 //@variable The number of bars back in the chart's history to inspect.
5 int barsBack = input.int(10, "Bars back", 0, 4999)
6
7 //@variable The percent rank of `volume` over 10 bars.
8 float weight = ta.percentrank(volume, 10)
9 //@variable The 10-bar EMA of `weight * close` values.
10 float numerator = ta.ema(weight * close, 10)
11 //@variable The 10-bar EMA of `weight` values.
12 float denominator = ta.ema(weight, 10)
13 //@variable The ratio of the `numerator` to the `denominator`.

```

(continues on next page)

(continued from previous page)

```

14 float filter = numerator / denominator
15
16 // Plot the `filter`.
17 plot(filter, "Custom filter")
18
19 //@variable The color of the frame, border, and text in the `debugTable`.
20 color tableColor = chart.fg_color
21
22 //@variable A table that contains "string" representations of variable names and_
23 // values on the latest chart bar.
23 var table debugTable = table.new(
24     position.top_right, 2, 5, frame_color = tableColor, frame_width = 1, border_
25 //color = tableColor, border_width = 1
25 )
26
27 // Initialize cells on the last confirmed historical bar.
28 if barstate.islastconfirmedhistory
29     table.cell(debugTable, 0, 0, "Variable", text_color = tableColor)
30     table.cell(debugTable, 1, 0, str.format("Value {0, number, integer} bars ago",_
31 //barsBack), text_color = tableColor)
32     table.cell(debugTable, 0, 1, "weight", text_color = tableColor)
32     table.cell(debugTable, 1, 1, "", text_color = tableColor)
33     table.cell(debugTable, 0, 2, "numerator", text_color = tableColor)
34     table.cell(debugTable, 1, 2, "", text_color = tableColor)
35     table.cell(debugTable, 0, 3, "denominator", text_color = tableColor)
36     table.cell(debugTable, 1, 3, "", text_color = tableColor)
37     table.cell(debugTable, 0, 4, "filter", text_color = tableColor)
38     table.cell(debugTable, 1, 4, "", text_color = tableColor)
39
40 // Update value cells on the last available bar.
41 if barstate.islast
42     table.cell_set_text(debugTable, 1, 1, str.tostring(weight[barsBack], format._
43 //percent))
43     table.cell_set_text(debugTable, 1, 2, str.tostring(numerator[barsBack]))
44     table.cell_set_text(debugTable, 1, 3, str.tostring(denominator[barsBack]))
45     table.cell_set_text(debugTable, 1, 4, str.tostring(filter[barsBack]))

```

**Note that:**

- The script uses the `var` keyword to specify that the `table` assigned to the `debugTable` variable on the first bar persists throughout the script's execution.
- This script modifies the table within two `if` structures. The first structure initializes the cells with `table.cell()` only on the last confirmed historical bar (`barstate.islastconfirmedhistory`). The second structure updates the `text` properties of relevant cells with *string representations* of our variables' values using `table.cell_set_text()` calls on the latest available bar (`barstate.islast`).

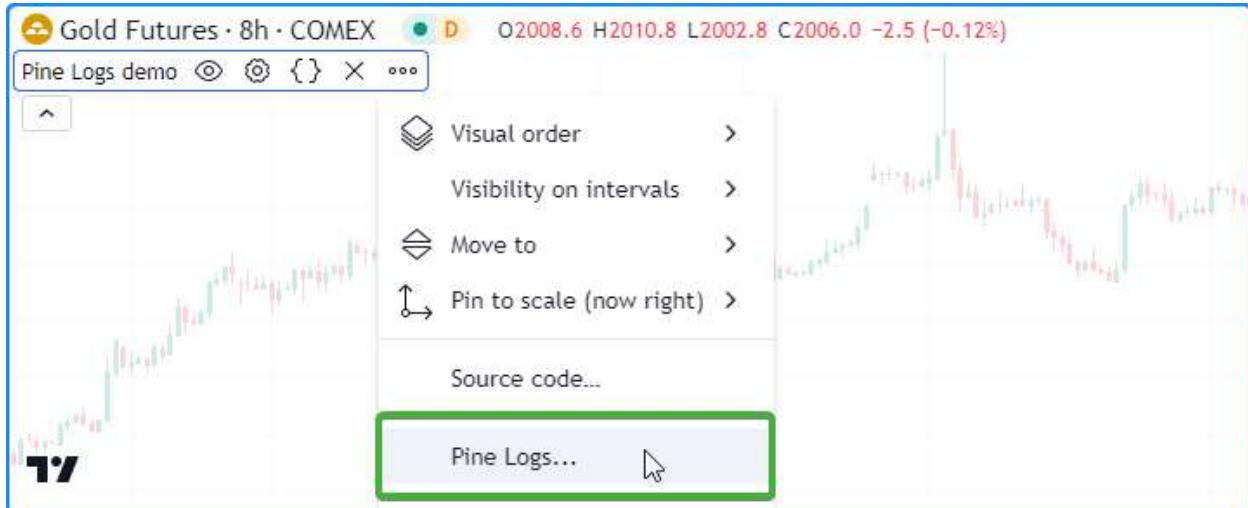
It's important to note that although tables can provide debugging utility, namely when working with multiple series or creating on-chart logs, they carry a higher computational cost than other techniques discussed on this page and may require *more code*. Additionally, unlike `labels`, one can only view a table's state from the latest script execution. We therefore recommend using them *wisely* and *sparingly* while debugging, opting for *simplified* approaches where possible. For more information about using `table` objects, see the [Tables](#) page.

## 5.2.6 Pine Logs

Pine Logs are *interactive messages* that scripts can output at specific points in their execution. They provide a powerful way for programmers to inspect a script's data, conditions, and execution flow with minimal code.

Unlike the other tools discussed on this page, Pine Logs have a deliberate design for in-depth script debugging. Scripts do not display Pine Logs on the chart or in the Data Window. Instead, they print messages with timestamps in the dedicated *Pine Logs pane*, which provides specialized navigation features and filtering options.

To access the Pine Logs pane, select “Pine Logs...” from the Editor’s “More” menu or from the “More” menu of a script loaded on the chart that uses `log.*()` functions:



**Note:** Only **personal scripts** can generate Pine Logs. A published script *cannot* create logs, even if it has `log.*()` function calls in its code. One must consider alternative approaches, such as those outlined in the sections above, when *publishing scripts* with debugging functionality.

### Creating logs

Scripts can create logs by calling the functions in the `log.*()` namespace.

All `log.*()` functions have the following signatures:

```
log.*(message) → void
log.*(formatString, arg0, arg1, ...) → void
```

The first overload logs a specified `message` in the Pine Logs pane. The second overload is similar to `str.format()`, as it logs a formatted message based on the `formatString` and the additional arguments supplied in the call.

Each `log.*()` function has a different *debug level*, allowing programmers to categorize and *filter* results shown in the pane:

- The `log.info()` function logs an entry with the “*info*” level that appears in the pane with gray text.
- The `log.warning()` function logs an entry with the “*warning*” level that appears in the pane with orange text.
- The `log.error()` function logs an entry with the “*error*” level that appears in the pane with red text.

This code demonstrates the difference between all three `log.*()` functions. It calls `log.info()`, `log.warning()`, and `log.error()` on the first available bar:



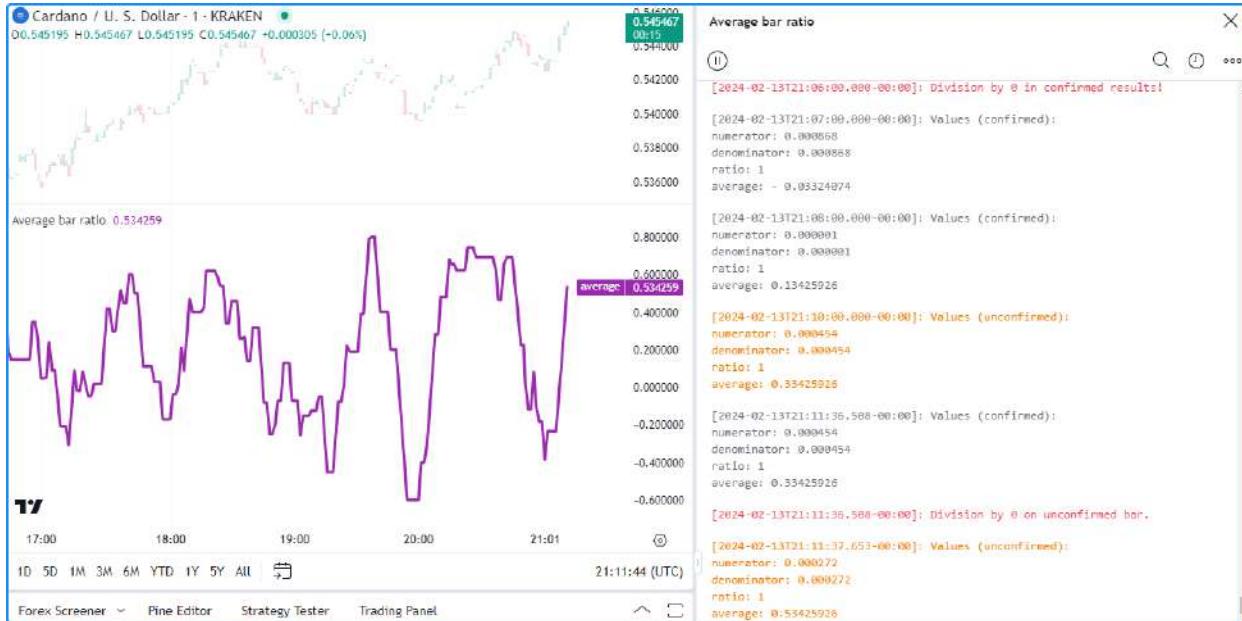
```

1 //@version=5
2 indicator("Debug levels demo", overlay = true)
3
4 if barstate.isfirst
5     log.info("This is an 'info' message.")
6     log.warning("This is a 'warning' message.")
7     log.error("This is an 'error' message.")

```

Pine Logs can execute anywhere within a script's execution. They allow programmers to track information from historical bars and monitor how their scripts behave on realtime, *unconfirmed* bars. When executing on historical bars, scripts generate a new message once for each `log.*()` call on a bar. On realtime bars, calls to `log.*()` functions can create new entries on *each new tick*.

For example, this script calculates the average ratio between each bar's `close - open` value to its `high - low` range. When the denominator is nonzero, the script calls `log.info()` to print the values of the calculation's variables on confirmed bars and `log.warning()` to print the values on unconfirmed bars. Otherwise, it uses `log.error()` to indicate that division by zero occurred, as such cases can affect the average result:



```

1 // @version=5
2 indicator("Logging historical and realtime data demo", "Average bar ratio")
3
4 //@variable The current bar's change from the `open` to `close`.
5 float numerator = close - open
6 //@variable The current bar's `low` to `high` range.
7 float denominator = high - low
8 //@variable The ratio of the bar's open-to-close range to its full range.
9 float ratio = numerator / denominator
10 //@variable The average `ratio` over 10 non-na values.
11 float average = ta.sma(ratio, 10)
12
13 // Plot the `average`.
14 plot(average, "average", color.purple, 3)
15
16 if barstate.isconfirmed
17     // Log a division by zero error if the `denominator` is 0.
18     if denominator == 0.0
19         log.error("Division by 0 in confirmed results!")
20     // Otherwise, log the confirmed values.
21     else
22         log.info(
23             "Values (confirmed):\nnumerator: {1, number, #####}\ndenominator:
24             →{2, number, #####}\n\nratio: {0, number, #####}\naverage: {3, number, #####}",
25             ratio, numerator, denominator, average
26         )
27 else
28     // Log a division by zero error if the `denominator` is 0.
29     if denominator == 0.0
30         log.error("Division by 0 on unconfirmed bar.")
31     // Otherwise, log the unconfirmed values.
32     else
33         log.warning(
34             "Values (unconfirmed):\nnumerator: {1, number, #####}\ndenominator:
35             →{2, number, #####}\n\nratio: {0, number, #####}\naverage: {3, number, #####}",
36             ratio, numerator, denominator, average
37         )

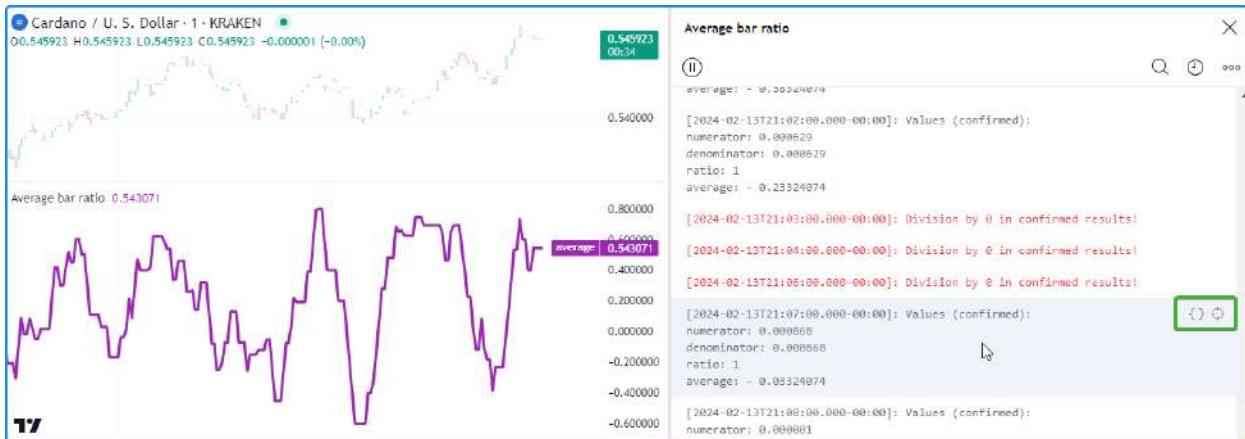
```

**Note that:**

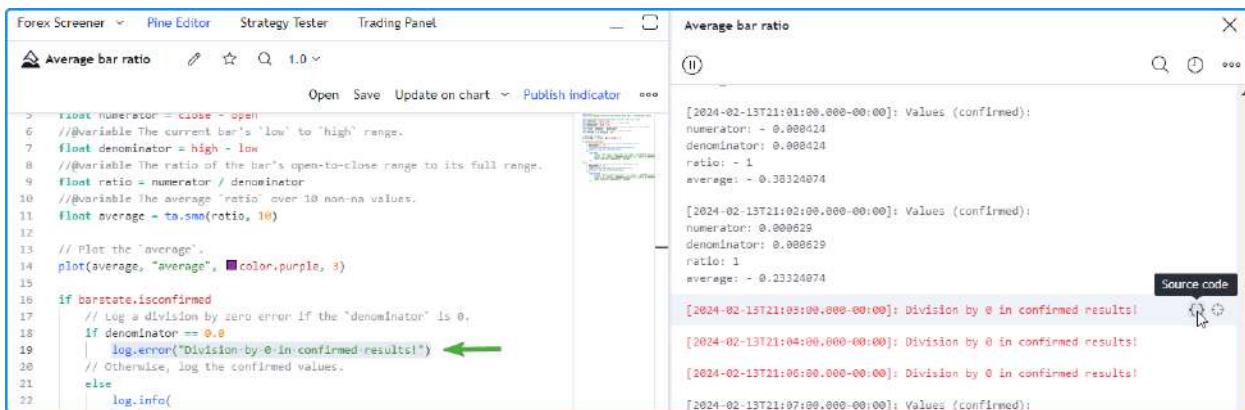
- Pine Logs *do not roll back* on each tick in an unconfirmed bar, meaning the results for those ticks show in the pane until the script restarts its execution. To only log messages on *confirmed* bars, use `barstate.isconfirmed` in the conditions that trigger a `log.*()` call.
- When logging on unconfirmed bars, we recommend ensuring those logs contain *unique information* or use different *debug levels* so you can *filter* the results as needed.
- The Pine Logs pane will show up to the most recent 10,000 entries for historical bars. If a script generates more than 10,000 logs on historical bars and a programmer needs to view earlier entries, they can use conditional logic to limit `log.*()` calls to specific occurrences. See [this](#) section for an example that limits log generation to a user-specified time range.

### Inspecting logs

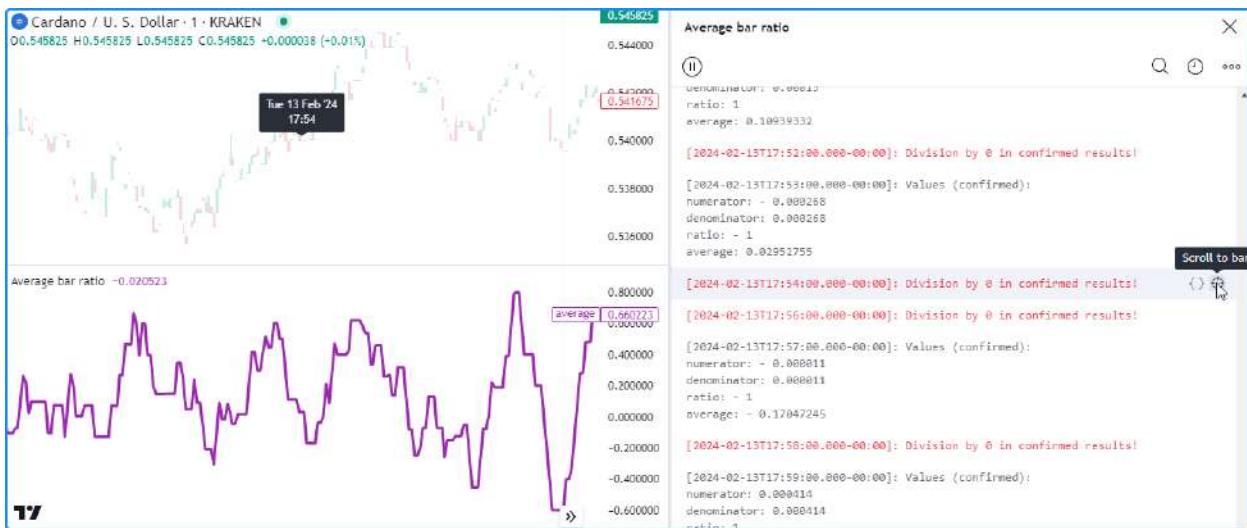
Pine Logs include some helpful features that simplify the inspection process. Whenever a script generates a log, it automatically prefixes the message with a granular timestamp to signify where the log event occurred in the time series. Additionally, each entry contains “Source code” and “Scroll to bar” icons, which appear when hovering over it in the Pine Logs pane:



Clicking an entry’s “Source code” icon opens the script in the Pine Editor and highlights the specific line of code that triggered the log:



Clicking an entry’s “Scroll to bar” icon navigates the chart to the specific bar where the log occurred, then temporarily displays a tooltip containing time information for that bar:



### Note that:

- The time information in the tooltip depends on the chart's timeframe, just like the x-axis label linked to the chart's cursor and drawing tools. For example, the tooltip on an EOD chart will only show the weekday and the date, whereas the tooltip on a 10-second chart will also contain the time of day, including seconds.

When a chart includes more than one script that generates logs, it's important to note that each script maintains its own *independent* message history. To inspect the messages from a specific script when multiple are on the chart, select its title from the dropdown at the top of the Pine Logs pane:

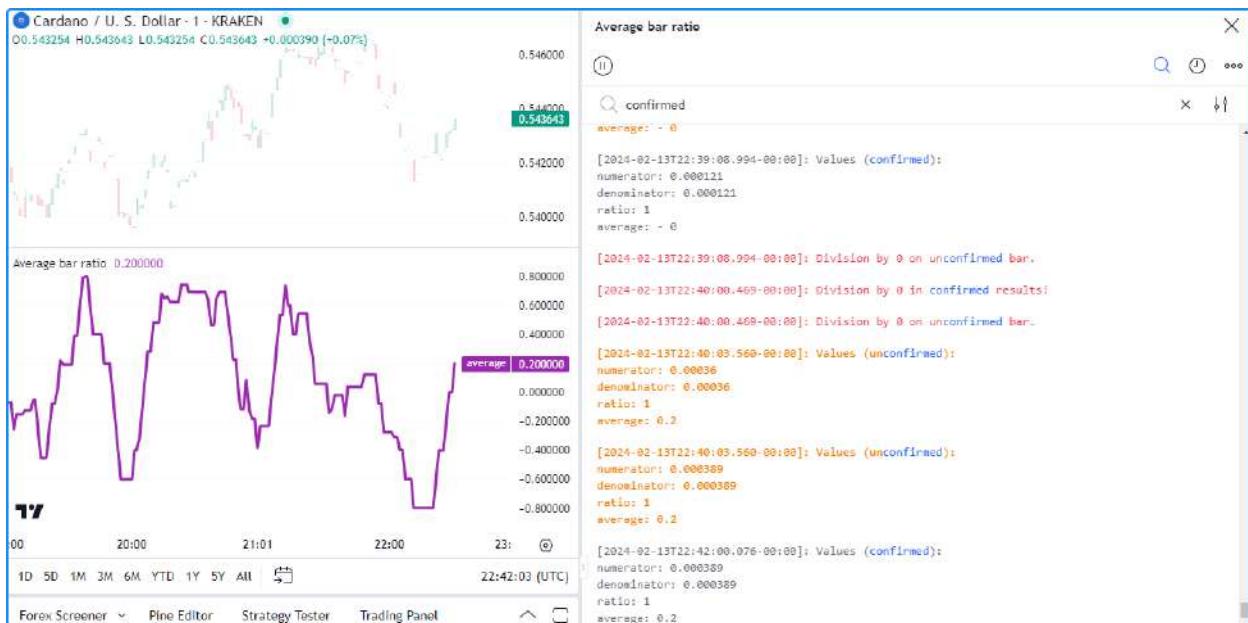


### Filtering logs

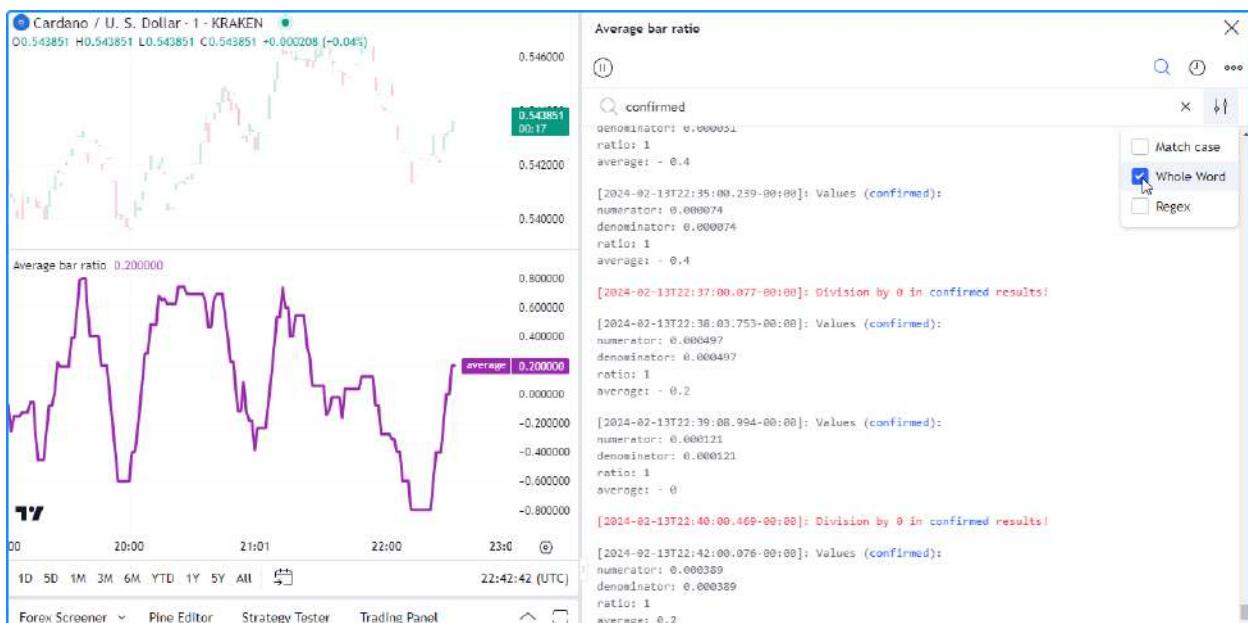
A single script can generate numerous logs, depending on the conditions that trigger its `log.*()` calls. While directly scrolling through the log history to find specific entries may suffice when a script only generates a few, it can become unwieldy when searching through hundreds or thousands of messages.

The Pine Logs pane includes multiple options for filtering messages, which allows one to simplify their results by isolating specific *character sequences*, *start times*, and *debug levels*.

Clicking the “Search” icon at the top of the pane opens a search bar, which matches text to filter logged messages. The search filter also highlights the matched portion of each message in blue for visual reference. For example, here, we entered “confirmed” to match all results generated by our previous script with the word somewhere in their text:

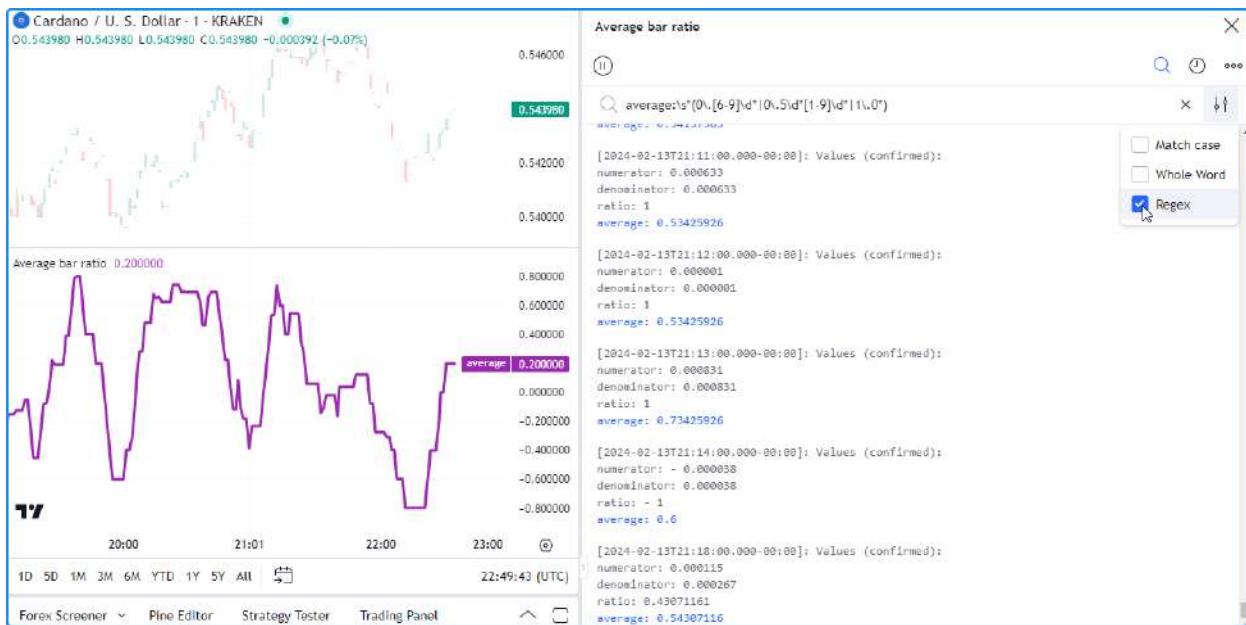


Notice that the results from this search also considered messages with “*unconfirmed*” as matches since the word contains our query. We can omit these matches by selecting the “Whole Word” checkbox in the options at the right of the search bar:

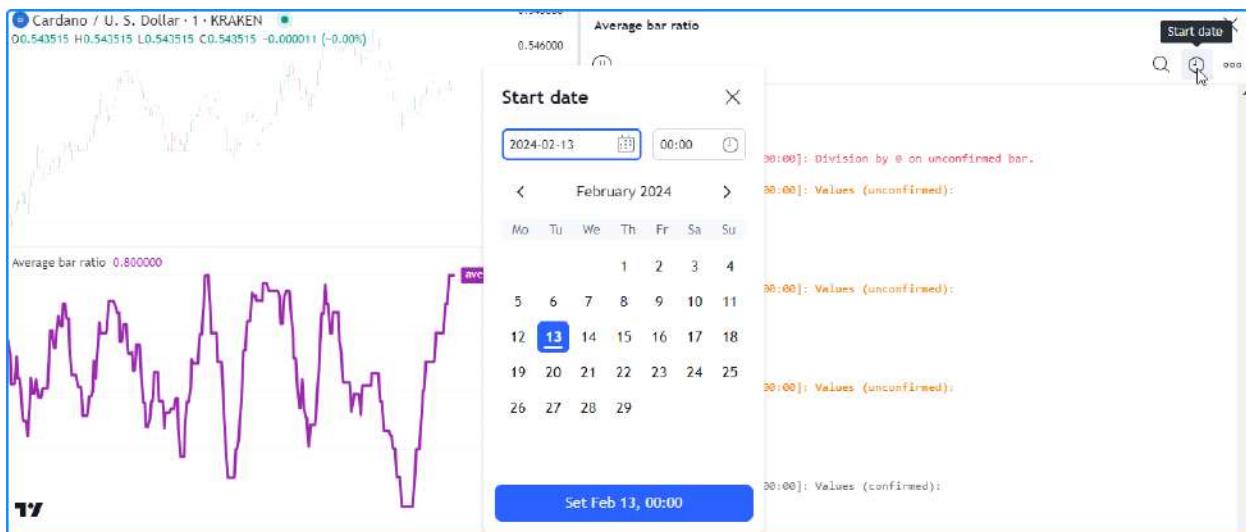


This filter also supports [regular expressions \(regex\)](#), which allow users to perform advanced searches that match custom *character patterns* when selecting the “Regex” checkbox in the search options. For example, this regex matches all entries that contain “average” followed by a sequence representing a number greater than 0.5 and less than or equal to 1:

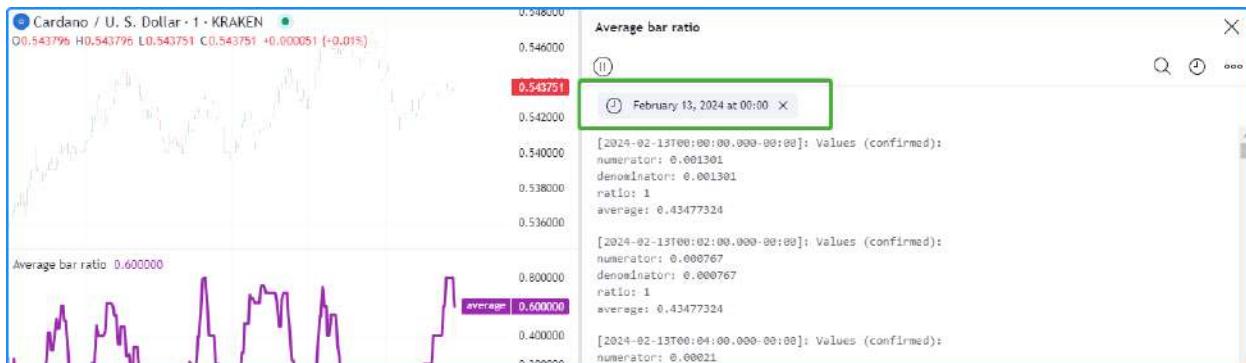
```
average:\s*(0\. [6-9] \d* | 0\.5\d* [1-9] \d* | 1\.0*)
```



Clicking the “Start date” icon opens a dialog that allows users to specify the date and time of the first log shown in the results:

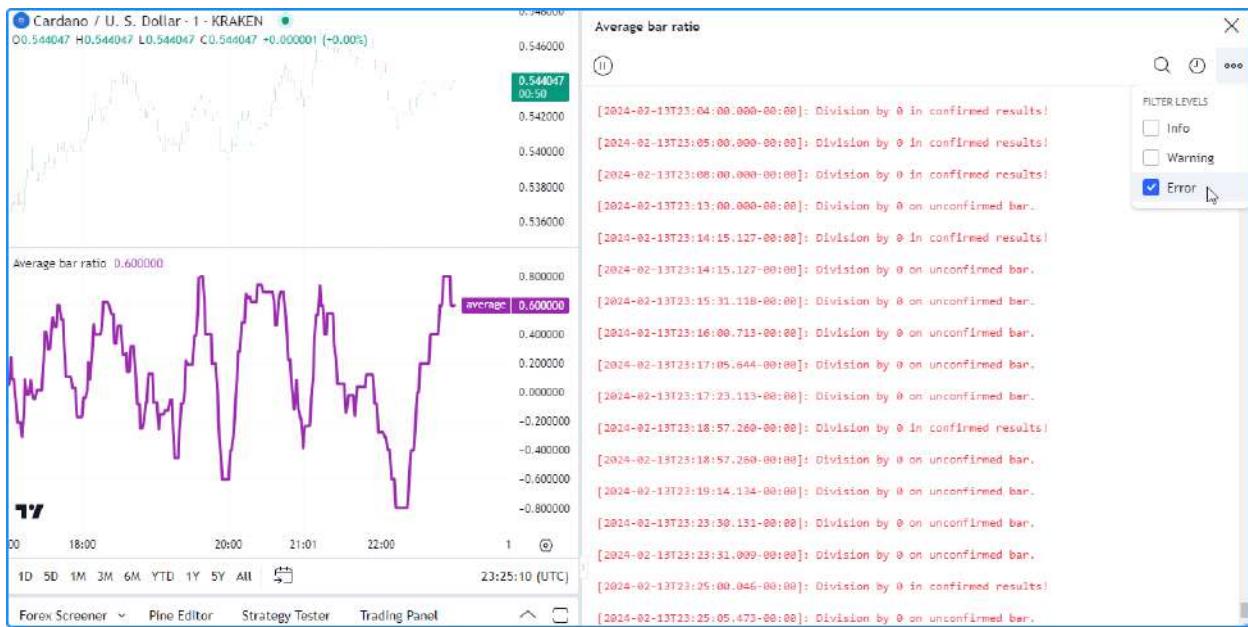


After specifying the starting point, a tag containing the starting time will appear above the log history:



Users can filter results by *debug level* using the checkboxes available when selecting the rightmost icon in the filtering

options. Here, we've deactivated the “info” and “warning” levels so the results will only contain “error” messages:



## Using inputs

Another, more involved way to interactively filter a script's logged results is to create *inputs* linked to conditional logic that activates specific `log.*()` calls in the code.

Let's look at an example. This code calculates an **RMA** of `close` prices and declares a few unique conditions to form a *compound condition*. The script uses `log.info()` to display important debugging information in the Pine Logs pane, including the values of the `compoundCondition` variable and the “bool” variables that determine its result.

We declared the `filterLogsInput`, `logStartInput`, and `logEndInput` variables respectively assigned to an `input.bool()` and two `input.time()` calls for custom log filtering. When `filterLogsInput` is `true`, the script will only generate a new log if the bar's `time` is between the `logStartInput` and `logEndInput` values, allowing us to interactively isolate the entries that occurred within a specific time range:



```

1 // @version=5
2 indicator("Filtering logs using inputs demo", "Compound condition in input range", ↵
3           true)
4
5 // @variable The length for moving average calculations.
6 int lengthInput = input.int(20, "Length", 2)
7
8 // @variable If `true`, only allows logs within the input time range.
9 bool filterLogsInput = input.bool(true, "Only log in time range", group = "Log filter" ↵)
10
11 // @variable The starting time for logs if `filterLogsInput` is `true`.
12 int logStartInput = input.time(0, "Start time", group = "Log filter", confirm = true)
13 // @variable The ending time for logs if `filterLogsInput` is `true`.
14 int logEndInput = input.time(0, "End time", group = "Log filter", confirm = true)
15
16 // @variable The RMA of `close` prices.
17 float rma = ta.rma(close, lengthInput)
18
19 // @variable Is `true` when `close` exceeds the `rma`.
20 bool priceBelow = close <= rma
21 // @variable Is `true` when the current `close` is greater than the max of the ↵
22 // previous `hl2` and `close`.
23 bool priceRising = close > math.max(hl2[1], close[1])
24 // @variable Is `true` when the `rma` is positively accelerating.
25 bool rmaAccelerating = rma - 2.0 * rma[1] + rma[2] > 0.0
26 // @variable Is `true` when the difference between `rma` and `close` exceeds 2 times ↵
27 // the current ATR.
28 bool closeAtThreshold = rma - close > ta.atr(lengthInput) * 2.0
29 // @variable Is `true` when all the above conditions occur.
30 bool compoundCondition = priceBelow and priceRising and rmaAccelerating and ↵
31 // closeAtThreshold
32
33 // Plot the `rma`.
34 plot(rma, "RMA", color.teal, 3)
35 // Highlight the chart background when the `compoundCondition` occurs.
36 bgcolor(compoundCondition ? color.new(color.aqua, 80) : na, title = "Compound" ↵
37 // condition highlight")
38
39 // @variable If `filterLogsInput` is `true`, is only `true` in the input time range. ↵
40 // Otherwise, always `true`.
41 bool showLog = filterLogsInput ? time >= logStartInput and time <= logEndInput : true
42
43 // Log results for a confirmed bar when `showLog` is `true`.
44 if barstate.isconfirmed and showLog
45     log.info(
46         "\nclose: {0, number, #.#####}\nrma: {1, number, #.#####}\npriceBelow: {2}\\" ↵
47         "npriceRising: {3}\n\nrmaAccelerating: {4}\nncloseAtThreshold: {5}\n\ncompoundCondition: {6}", ↵
48         close, rma, priceBelow, priceRising, rmaAccelerating, closeAtThreshold, ↵
49         compoundCondition
50     )

```

**Note that:**

- The `input.*()` functions assigned to the `filterLogsInput`, `logStartInput`, and `logEndInput` variables include a `group` argument to organize and distinguish them in the script's settings.
- The `input.time()` calls include `confirm = true` so that we can interactively set the start and end times directly on the chart. To reset the inputs, select “Reset points...” from the options in the script’s “More”

menu.

- The condition that triggers each `log.info()` call includes `barstate.isconfirmed` to limit log generation to *confirmed* bars.

## 5.2.7 Debugging functions

*User-defined functions* and *methods* are custom functions written by users. They encapsulate sequences of operations that a script can invoke later in its execution.

Every *user-defined function* or *method* has a *local scope* that embeds into the script's global scope. The parameters in a function's signature and the variables declared within the function body belong to that function's local scope, and they are *not* directly accessible to a script's outer scope or the scopes of other functions.

The segments below explain a few ways programmers can debug the values from a function's local scope. We will use this script as the starting point for our subsequent examples. It contains a `customMA()` function that returns an exponential moving average whose smoothing parameter varies based on the `source` distance outside the 25th and 75th percentiles over `length` bars:



```

1 // @version=5
2 indicator("Debugging functions demo", "Custom MA", true)
3
4 //@variable The number of bars in the `customMA()` calculation.
5 int lengthInput = input.int(50, "Length", 2)
6
7 //@function Calculates a moving average that only responds to values outside the
8 //→first and third quartiles.
9 //@param source The series of values to process.
10 //@param length The number of bars in the calculation.
11 //@returns The moving average value.
12 customMA(float source, int length) =>
13     //@variable The custom moving average.
14     var float result = na
15     // Calculate the 25th and 75th `source` percentiles.
16     float q1 = ta.percentile_linear_interpolation(source, length, 25)
17     float q3 = ta.percentile_linear_interpolation(source, length, 75)
18     // Calculate the range values.
19     float outerRange = math.max(source - q3, q1 - source, 0.0)
20     float totalRange = ta.range(source, length)
21     //@variable Half the ratio of the `outerRange` to the `totalRange`.

```

(continues on next page)

(continued from previous page)

```

21 float alpha = 0.5 * outerRange / totalRange
22 // Mix the `source` with the `result` based on the `alpha` value.
23 result := (1.0 - alpha) * nz(result, source) + alpha * source
24 // Return the `result`.
25 result
26
27 //@variable The `customMA()` result over `lengthInput` bars.
28 float maValue = customMA(close, lengthInput)
29
30 // Plot the `maValue`.
31 plot(maValue, "Custom MA", color.blue, 3)

```

## Extracting local variables

When a programmer wants to inspect a *user-defined function's* local variables by *plotting* its values, *coloring* the background or chart bars, etc., they must *extract* the values to the *global scope*, as the built-in functions that produce such outputs can only accept global variables and literals.

Since the values returned by a function are available to the scope where a call occurs, one straightforward extraction approach is to have the function return a *tuple* containing all the values that need inspection.

Here, we've modified the `customMA()` function to return a *tuple* containing all the function's calculated variables. Now, we can call the function with a *tuple declaration* to make the values available in the global scope and inspect them with *plots*:



```

1 //@version=5
2 indicator("Extracting local variables with tuples demo", "Custom MA", true)
3
4 //@variable The number of bars in the `customMA()` calculation.
5 int lengthInput = input.int(50, "Length", 2)
6
7 //@function Calculates a moving average that only responds to values outside the
8 //→ first and third quartiles.
9 //@param source The series of values to process.
10 //@param length The number of bars in the calculation.
11 //@returns The moving average value.

```

(continues on next page)

(continued from previous page)

```

11 customMA(float source, int length) =>
12     //@variable The custom moving average.
13     var float result = na
14     // Calculate the 25th and 75th `source` percentiles.
15     float q1 = ta.percentile_linear_interpolation(source, length, 25)
16     float q3 = ta.percentile_linear_interpolation(source, length, 75)
17     // Calculate the range values.
18     float outerRange = math.max(source - q3, q1 - source, 0.0)
19     float totalRange = ta.range(source, length)
20     //@variable Half the ratio of the `outerRange` to the `totalRange`.
21     float alpha = 0.5 * outerRange / totalRange
22     // Mix the `source` with the `result` based on the `alpha` value.
23     result := (1.0 - alpha) * nz(result, source) + alpha * source
24     // Return a tuple containing the `result` and other local variables.
25     [result, q1, q3, outerRange, totalRange, alpha]
26
27 // Declare a tuple containing all values returned by `customMA()` .
28 [maValue, q1Debug, q3Debug, outerRangeDebug, totalRangeDebug, alphaDebug] =_
29     ↪customMA(close, lengthInput)
30
31 // Plot the `maValue` .
32 plot(maValue, "Custom MA", color.blue, 3)
33
34 //@variable Display location for plots with different scale.
35 notOnPane = display.all - display.pane
36
37 // Display the extracted `q1` and `q3` values in all plot locations.
38 plot(q1Debug, "q1", color.new(color.maroon, 50))
39 plot(q3Debug, "q3", color.new(color.teal, 50))
40 // Display the other extracted values in the status line and Data Window to avoid_
41 // impacting the scale.
42 plot(outerRangeDebug, "outerRange", chart.fg_color, display = notOnPane)
43 plot(totalRangeDebug, "totalRange", chart.fg_color, display = notOnPane)
44 plot(alphaDebug, "alpha", chart.fg_color, display = notOnPane)
45 // Highlight the chart when `alphaDebug` is 0, i.e., when the `maValue` does not_
46 // change.
47 bgcolor(alphaDebug == 0.0 ? color.new(color.orange, 90) : na, title = "`alpha == 0.0`"_
48     ↪highlight")

```

**Note that:**

- We used `display.all - display.pane` for the plots of the `outerRangeDebug`, `totalRangeDebug`, and `alphaDebug` variables to *avoid impacting the chart's scale*.
- The script also uses a *conditional color* to highlight the chart pane's *background* when `debugAlpha` is 0, indicating the `maValue` does not change.

Another, more *advanced* way to extract the values of a function's local variables is to pass them to a *reference type* variable declared in the global scope.

Function scopes can access global variables for their calculations. While a script cannot directly reassign the values of global variables from within a function's scope, it can update the *elements or properties* of those values if they are reference types, such as *arrays*, *matrices*, *maps*, and *user-defined types*.

This version declares a `debugData` variable in the global scope that references a `map` with “string” keys and “float” values. Within the local scope of the `customMA()` function, the script puts *key-value pairs* containing each local variable's name and value into the map. After calling the function, the script plots the stored `debugData` values:

```

1 //@version=5
2 indicator("Extracting local variables with reference types demo", "Custom MA", true)
3
4 //@variable The number of bars in the `customMA()` calculation.
5 int lengthInput = input.int(50, "Length", 2)
6
7 //@variable A map with "string" keys and "float" values for debugging the
8 // `customMA()` .
9 map<string, float> debugData = map.new<string, float>()
10
11 //@function Calculates a moving average that only responds to values outside the
12 // first and third quartiles.
13 //@param source The series of values to process.
14 //@param length The number of bars in the calculation.
15 //@returns The moving average value.
16 customMA(float source, int length) =>
17     //@variable The custom moving average.
18     var float result = na
19     // Calculate the 25th and 75th `source` percentiles.
20     float q1 = ta.percentile_linear_interpolation(source, length, 25),      map.
21     ↪put(debugData, "q1", q1)
22     float q3 = ta.percentile_linear_interpolation(source, length, 75),      map.
23     ↪put(debugData, "q3", q3)
24     // Calculate the range values.
25     float outerRange = math.max(source - q3, q1 - source, 0.0),          map.
26     ↪put(debugData, "outerRange", outerRange)
27     float totalRange = ta.range(source, length),                          map.
28     ↪put(debugData, "totalRange", totalRange)
29     // @variable Half the ratio of the `outerRange` to the `totalRange` .
30     float alpha = 0.5 * outerRange / totalRange,                         map.
31     ↪put(debugData, "alpha", alpha)
32     // Mix the `source` with the `result` based on the `alpha` value.
33     result := (1.0 - alpha) * nz(result, source) + alpha * source
34     // Return the `result` .
35     result
36
37 //@variable The `customMA()` result over `lengthInput` bars.
38 float maValue = customMA(close, lengthInput)
39
40 // Plot the `maValue` .
41 plot(maValue, "Custom MA", color.blue, 3)
42
43 //@variable Display location for plots with different scale.
44 notOnPane = display.all - display.pane
45
46 // Display the extracted `q1` and `q3` values in all plot locations.
47 plot(map.get(debugData, "q1"), "q1", color.new(color.maroon, 50))
48 plot(map.get(debugData, "q3"), "q3", color.new(color.teal, 50))
49 // Display the other extracted values in the status line and Data Window to avoid
50 // impacting the scale.
51 plot(map.get(debugData, "outerRange"), "outerRange", chart.fg_color, display =
52 //notOnPane)
53 plot(map.get(debugData, "totalRange"), "totalRange", chart.fg_color, display =
54 //notOnPane)
55 plot(map.get(debugData, "alpha"), "alpha", chart.fg_color, display = notOnPane)
56 // Highlight the chart when the extracted `alpha` is 0, i.e., when the `maValue` does
57 // not change.

```

(continues on next page)

(continued from previous page)

```
47 bgcolor(map.get(debugData, "alpha") == 0.0 ? color.new(color.orange, 90) : na, title_
    ↵ = "`alpha == 0.0` highlight")
```

### Note that:

- We placed each `map.put()` call on the same line as each variable declaration, separated by a comma, to keep things concise and avoid adding extra lines to the `customMA()` code.
- We used `map.get()` to retrieve each value for the debug `plot()` and `bgcolor()` calls.

### Local drawings and logs

Unlike `plot.*()` functions and others that require values accessible to the global scope, scripts can generate *drawing objects* and *Pine Logs* directly within a function, allowing programmers to flexibly debug its local variables *without* extracting values to the outer scope.

In this example, we used *labels* and *Pine Logs* to display *string representations* of the values within the `customMA()` scope. Inside the function, the script calls `str.format()` to create a formatted string representing the local scope's data, then calls `label.new()` and `log.info()` to respectively display the text on the chart in a tooltip and log an “info” message containing the text in the *Pine Logs* pane:



```
1 // @version=5
2 indicator("Local drawings and logs demo", "Custom MA", true, max_labels_count = 500)
3
4 // @variable The number of bars in the `customMA()` calculation.
5 int lengthInput = input.int(50, "Length", 2)
6
7 // @function Calculates a moving average that only responds to values outside the
    ↵ first and third quartiles.
8 // @param source The series of values to process.
9 // @param length The number of bars in the calculation.
10 // @returns The moving average value.
11 customMA(float source, int length) =>
12     // @variable The custom moving average.
13     var float result = na
14     // Calculate the 25th and 75th `source` percentiles.
15     float q1 = ta.percentile_linear_interpolation(source, length, 25)
16     float q3 = ta.percentile_linear_interpolation(source, length, 75)
```

(continues on next page)

(continued from previous page)

```

17 // Calculate the range values.
18 float outerRange = math.max(source - q3, q1 - source, 0.0)
19 float totalRange = ta.range(source, length)
20 // @variable Half the ratio of the `outerRange` to the `totalRange`.
21 float alpha = 0.5 * outerRange / totalRange
22 // Mix the `source` with the `result` based on the `alpha` value.
23 result := (1.0 - alpha) * nz(result, source) + alpha * source
24
25 // @variable A formatted string containing representations of all local variables.
26 string debugText = str.format(
27     "\n`customMA()` data\n-----\nsource: {0, number, #####}\nlength:
28     {1, number, #####}\nq1: {2, number, #####}\nnq3: {3, number, #####}\nouterRange: {4, number, #####}\n
29     totalRange: {5, number, #####}\nalpha{6, number, #####}\nresult: {7, number, #####}",
30     source, length, q1, q3, outerRange, totalRange, alpha, result
31 )
32 // Draw a label with a tooltip displaying the `debugText`.
33 label.new(bar_index, high, color = color.new(chart.fg_color, 80), tooltip =_
34     debugText)
35 // Print an "info" message in the Pine Logs pane when the bar is confirmed.
36 if barstate.isconfirmed
37     log.info(debugText)
38
39 // Return the `result`.
40 result
41
42 // @variable The `customMA()` result over `lengthInput` bars.
43 float maValue = customMA(close, lengthInput)
44
45 // Plot the `maValue`.
plot(maValue, "Custom MA", color.blue, 3)

```

**Note that:**

- We included `max_labels_count = 500` in the `indicator()` function to display `labels` for the most recent 500 `customMA()` calls.
- The function uses `barstate.isconfirmed` in an `if` statement to only call `log.info()` on *confirmed* bars. It does not log a new message on each realtime tick.

## 5.2.8 Debugging loops

*Loops* are structures that repeatedly execute a code block based on a *counter* (`for`), the contents of a *collection* (`for...in`), or a *condition* (`while`). They allow scripts to perform repetitive tasks without the need for redundant lines of code.

Each loop instance maintains a separate local scope, which all outer scopes cannot access. All variables declared within a loop's scope are specific to that loop, meaning one cannot use them in an outer scope.

As with other structures in Pine, there are numerous possible ways to debug loops. This section explores a few helpful techniques, including extracting local values for `plots`, inspecting values with `drawings`, and tracing a loop's execution with `Pine Logs`.

We will use this script as a starting point for the examples in the following segments. It aggregates the `close` value's rates of change over `1 - lookbackInput` bars and accumulates them in a `for` loop, then divides the result by the `lookbackInput` to calculate a final average value:



```

1 //@version=5
2 indicator("Debugging loops demo", "Aggregate ROC")
3
4 //@variable The number of bars in the calculation.
5 int lookbackInput = input.int(20, "Lookback", 1)
6
7 //@variable The average ROC of `close` prices over each length from 1 to
8 // `lookbackInput` bars.
9 float aroc = 0.0
10
11 // Calculation loop.
12 for length = 1 to lookbackInput
13     //@variable The `close` value `length` bars ago.
14     float pastClose = close[length]
15     //@variable The `close` rate of change over `length` bars.
16     float roc = (close - pastClose) / pastClose
17     // Add the `roc` to `aroc`.
18     aroc += roc
19
20 // Divide `aroc` by the `lookbackInput`.
21 aroc /= lookbackInput
22
23 // Plot the `aroc`.
24 plot(aroc, "aroc", color.blue, 3)

```

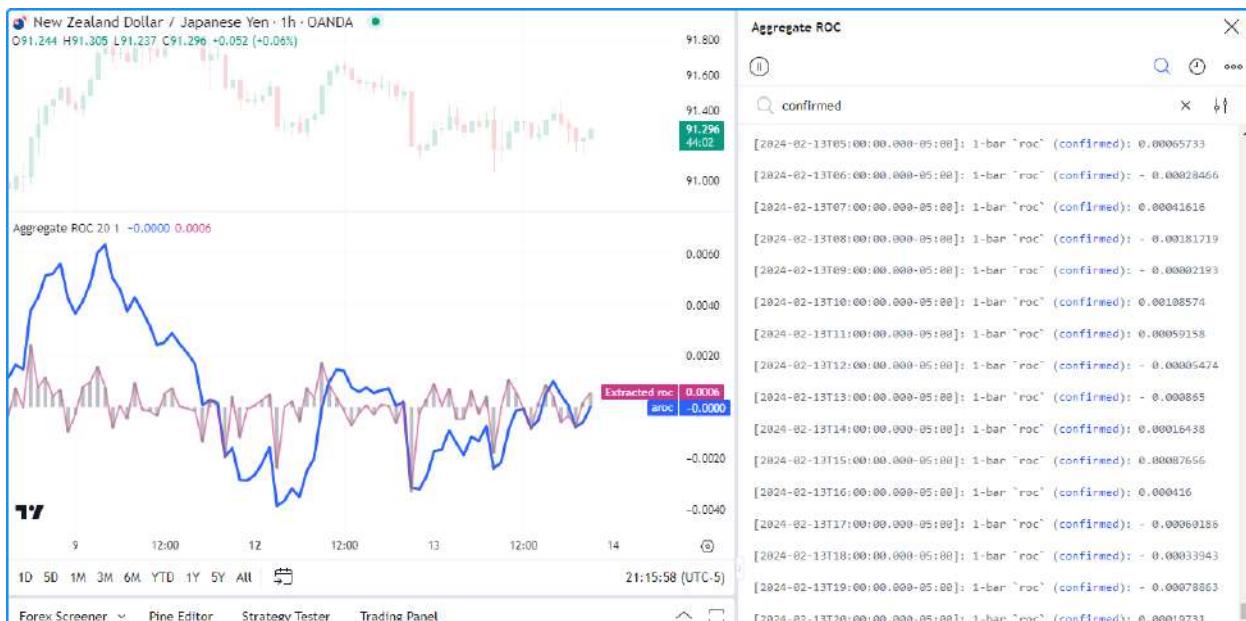
### Note that:

- The `aroc` is a *global* variable modified within the loop, whereas `pastClose` and `roc` are *local* variables inaccessible to the outer scope.

### Inspecting a single iteration

When a programmer needs to focus on a specific loop iteration, there are multiple techniques they can use, most of which entail using a *condition* inside the loop to trigger debugging actions, such as extracting values to outer variables, creating *drawings*, *logging* messages, etc.

This example inspects the local `roc` value from a single iteration of the loop in three different ways. When the loop counter's value equals the `debugCounterInput`, the script assigns the `roc` to an `rocDebug` variable from the global scope for *plotting*, draws a vertical `line` from 0 to the `roc` value using `line.new()`, and logs a message in the *Pine Logs* pane using `log.info()`:



```

1 // @version=5
2 indicator("Inspecting a single iteration demo", "Aggregate ROC", max_lines_count = ↴500)
3
4 // @variable The number of bars in the calculation.
5 int lookbackInput = input.int(20, "Lookback", 1)
6 // @variable The `length` value in the loop's execution where value extraction occurs.
7 int debugCounterInput = input.int(1, "Loop counter value", 1, group = "Debugging")
8
9 // @variable The `roc` value extracted from the loop.
10 float rocDebug = na
11
12 // @variable The average ROC of `close` over lags from 1 to `lookbackInput` bars.
13 float aroc = 0.0
14
15 // Calculation loop.
16 for length = 1 to lookbackInput
17     // @variable The `close` value `length` bars ago.
18     float pastClose = close[length]
19     // @variable The `close` rate of change over `length` bars.
20     float roc = (close - pastClose) / pastClose
21     // Add the `roc` to `aroc`.
22     aroc += roc
23
24     // Trigger debug actions when the `length` equals the `debugCounterInput`.
25     if length == debugCounterInput
26         // Assign `roc` to `rocDebug` so the script can plot its value.
27         rocDebug := roc
28         // Draw a vertical line from 0 to the `roc` at the `bar_index`.
29         line.new(bar_index, 0.0, bar_index, roc, color = color.new(color.gray, 50), ↴width = 4)
30         // Log an "info" message in the Pine Logs pane.
31         log.info("{0}-bar `roc`{1}: {2, number, #####}, length, barstate. ↴isconfirmed ? " (confirmed) : "", roc)
32
33 // Divide `aroc` by the `lookbackInput`.

```

(continues on next page)

(continued from previous page)

```

34 aroc /= lookbackInput
35
36 // Plot the `aroc`.
37 plot(aroc, "aroc", color.blue, 3)
38
39 // Plot the `rocDebug`.
40 plot(rocDebug, "Extracted roc", color.new(color.rgb(206, 55, 136), 40), 2)

```

## Note that:

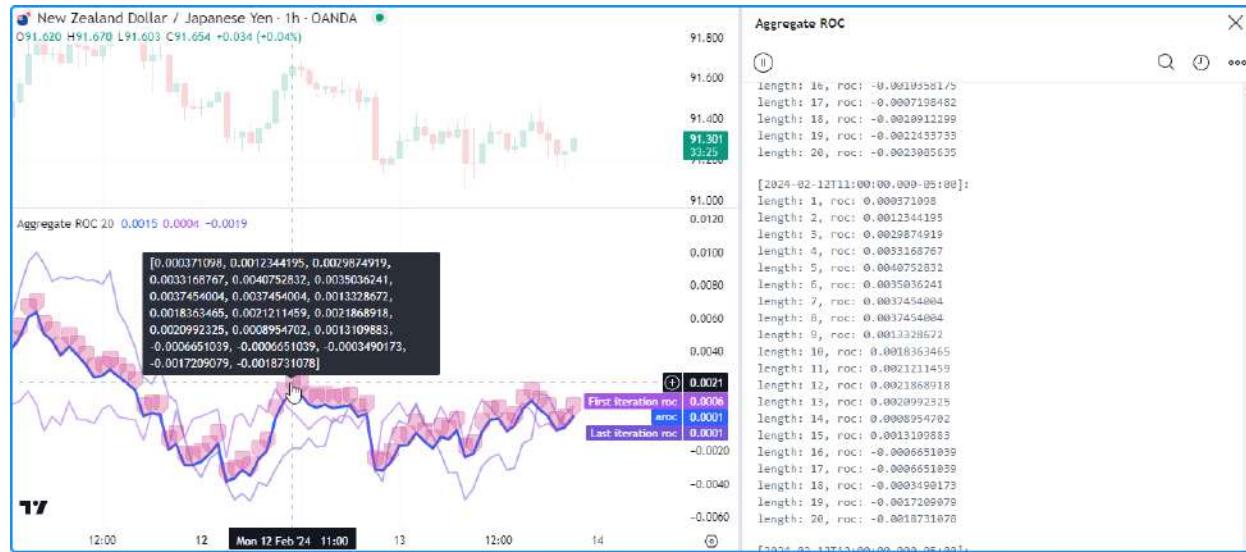
- The `input.int()` call assigned to the `debugCounterInput` includes a `group` argument to distinguish it in the script's settings.
- The `log.info()` call includes “(confirmed)” in the formatted message whenever `barstate.isconfirmed` is `true`. Searching this text in the *Pine Logs* pane will filter out the entries from unconfirmed bars. See the *Filtering logs* section above.

## Inspecting multiple iterations

When inspecting the values from several loop iterations, it's often helpful to utilize `collections` or strings to gather the results for use in output functions after the loop terminates.

This version demonstrates a few ways to collect and display the loop's values from all iterations. It declares a `logText` string and a `debugValues` array in the global scope. Inside the local scope of the `for` loop, the script *concatenates* a *string representation* of the `length` and `roc` with the `logText` and calls `array.push()` to push the iteration's `roc` value into the `debugValues` array.

After the loop ends, the script *plots* the `first` and `last` value from the `debugValues` array, draws a `label` with a *tooltip* showing a *string representation* of the array, and displays the `logText` in the *Pine Logs* pane upon the bar's confirmation:



```

1 // @version=5
2 indicator("Inspecting multiple iterations demo", "Aggregate ROC", max_labels_count = ↴
3 ↴ 500)
4
5 // @variable The number of bars in the calculation.
6 int lookbackInput = input.int(20, "Lookback", 1)

```

(continues on next page)

(continued from previous page)

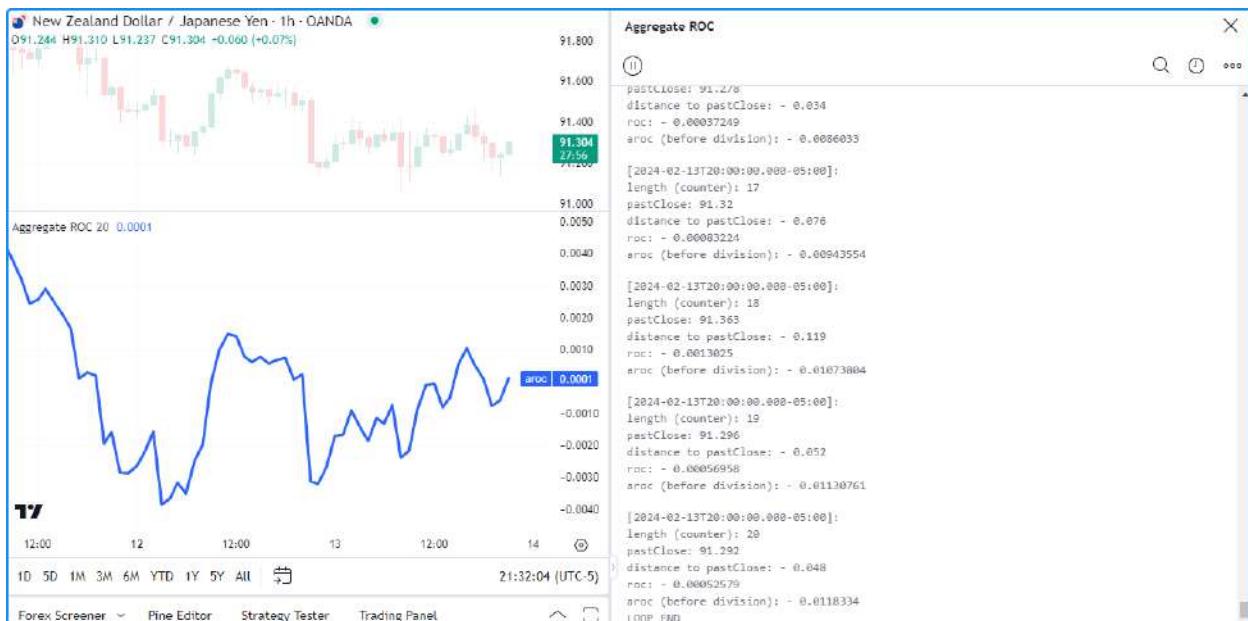
```

7 // @variable An array containing the `roc` value from each loop iteration.
8 array<float> debugValues = array.new<float>()
9 // @variable A "string" containing information about the `roc` on each iteration.
10 string logText = ""
11
12 // @variable The average ROC of `close` over lags from 1 to `lookbackInput` bars.
13 float aroc = 0.0
14
15 // Calculation loop.
16 for length = 1 to lookbackInput
17     // @variable The `close` value `length` bars ago.
18     float pastClose = close[length]
19     // @variable The `close` rate of change over `length` bars.
20     float roc = (close - pastClose) / pastClose
21     // Add the `roc` to `aroc`.
22     aroc += roc
23
24     // Concatenate a new "string" representation with the `debugText`.
25     logText += "\nlength: " + str.tostring(length) + ", roc: " + str.tostring(roc)
26     // Push the `roc` value into the `debugValues` array.
27     array.push(debugValues, roc)
28
29 // Divide `aroc` by the `lookbackInput`.
30 aroc /= lookbackInput
31
32 // Plot the `aroc`.
33 plot(aroc, "aroc", color.blue, 3)
34
35 // Plot the `roc` values from the first and last iteration.
36 plot(array.first(debugValues), "First iteration roc", color.new(color.rgb(166, 84, 233), 50), 2)
37 plot(array.last(debugValues), "Last iteration roc", color.new(color.rgb(115, 86, 218), 50), 2)
38 // Draw a label with a tooltip containing a "string" representation of the `debugValues` array.
39 label.new(bar_index, aroc, color = color.new(color.rgb(206, 55, 136), 70), tooltip = str.tostring(debugValues))
40 // Log the `logText` in the Pine Logs pane when the bar is confirmed.
41 if barstate.isconfirmed
42     log.info(logText)

```

Another way to inspect a loop over several iterations is to generate sequential [Pine Logs](#) or create/modify [drawing objects](#) within the loop's scope to trace its execution pattern with granular detail.

This example uses [Pine Logs](#) to trace the execution flow of our script's loop. It generates a new “info” message on each iteration to track the local scope's calculations as the loop progresses on each confirmed bar:



```

1 // @version=5
2 indicator("Inspecting multiple iterations demo", "Aggregate ROC")
3
4 // @variable The number of bars in the calculation.
5 int lookbackInput = input.int(20, "Lookback", 1)
6
7 // @variable The average ROC of `close` over lags from 1 to `lookbackInput` bars.
8 float aroc = 0.0
9
10 // Calculation loop.
11 for length = 1 to lookbackInput
12     // @variable The `close` value `length` bars ago.
13     float pastClose = close[length]
14     // @variable The `close` rate of change over `length` bars.
15     float roc = (close - pastClose) / pastClose
16     // Add the `roc` to `aroc`.
17     aroc += roc
18     if barstate.isconfirmed
19         log.info(
20             "{0}\nlength (counter): {1}\npastClose: {2, number, ######}\n"
21             "distance to pastClose: {3, number, #####}\nroc: {4, number, #####}\n"
22             "aroc (before division): {5, number, #####}\n{n{6}}",
23             length == 1 ? "LOOP START" : "",
24             length, pastClose, close - pastClose, roc, aroc,
25             length == lookbackInput ? "LOOP END" : ""
26         )
27
28 // Divide `aroc` by the `lookbackInput`.
29 aroc /= lookbackInput
30
31 // Plot the `aroc`.
32 plot(aroc, "aroc", color.blue, 3)

```

### Note that:

- When iteratively generating `logs` or drawings from inside a loop, make it a point to avoid unnecessary clutter

and strive for easy navigation. More is not always better for debugging, especially when working within loops.

## 5.2.9 Tips

### Organization and readability

When writing scripts, it's wise to prioritize organized, readable source codes. Code that's organized and easy to read helps streamline the debugging process. Additionally, well-written code is easier to maintain over time.

Here are a few quick tips based on our [Style guide](#) and the examples on this page:

- Aim to follow the general [script organization](#) recommendations. Organizing scripts using this structure makes things easier to locate and inspect.
- Choose variable and function names that make them easy to *identify* and *understand*. See the [Naming conventions](#) section for some examples.
- It's often helpful to temporarily assign important parts of expressions to variables with informative names while debugging. Breaking expressions down into reusable parts helps simplify inspection processes.
- Use *comments* and *annotations* (`//@function`, `//@variable`, etc.) to document your code. Annotations are particularly helpful, as the Pine Editor's autosuggest displays variable and function descriptions in a pop-up when hovering over their identifiers anywhere in the code.
- Remember that *less is more* in many cases. Don't overwhelm yourself with excessive script outputs or unnecessary information while debugging. Keep things simple, and only include as much information as you need.

### Speeding up repetitive tasks

There are a few handy techniques we often utilize when debugging our code:

- We use `plotchar()` or `plotshape()` to quickly display the results of “int”, “float”, or “bool” variables and expressions in the script's status line and the Data Window.
- We often use `bgcolor()` to visualize the history of certain *conditions* on the chart.
- We use a one-line version of our `printLabel()` function from [this section](#) to print strings at the end of the chart.
- We use a `label.new()` call with a `tooltip` argument to display strings in tooltips *on successive bars*.
- We use the `log.*()` functions to quickly display data with *string representations* in the [Pine Logs](#) pane.

When one establishes their typical debugging processes, it's often helpful to create *keyboard macros* to speed up repetitive tasks and spend less time setting up debug outputs in each code.

The following is a simple *AutoHotkey* script (**not** Pine Script™ code) that includes hotstrings for the above five techniques. The script generates code snippets by entering a specified character sequence followed by a whitespace:

```
; ----- This is AHK code, not Pine Script™. -----
; Specify that hotstrings trigger when they end with space, tab, linefeed, or
; carriage return.
#Hotstring EndChars `t `n `r

:X:,,show:::SendInput, plotchar(%Clipboard%, "%Clipboard%", "", color = chart.fg_color,
; display = display.all - display.pane){Enter}
:X:,,highlight:::SendInput, bgcolor(bool(%Clipboard%)) ? color.new(color.orange, 80) :_
; na, title = "%Clipboard% highlight"){Enter}
:X:,,print:::SendInput, printLabel(string txt, float price = na) => int labelTime =_
; (continues on next page)
```

(continued from previous page)

```
→math.max(last_bar_time, chart.right_visible_bar_time), var label result = label.
→new(labelTime, na, txt, xloc.bar_time, na(price) ? yloc.abovebar : yloc.price, na,
→label.style_none, chart.fg_color, size.large), label.set_text(result, txt), label.
→set_y(result, price), result`nprintLabel(){Left}
:X::, tooltip::SendInput, label.new(bar_index, high, color = color.new(chart.fg_color,_
→70), tooltip = str.tostring(%Clipboard%)){Enter}
:X::, log::SendInput, log.info(str.tostring(%Clipboard%)){Enter}
```

The “,,show” macro generates a `plotchar()` call that uses the clipboard’s contents for the `series` and `title` arguments. Copying a `variableName` variable or the `close > open` expression and typing “,,show” followed by a space will respectively yield:

```
plotchar(variableName, "variableName", "", color = chart.fg_color, display = display.
→all - display.pane)
plotchar(close > open, "close > open", "", color = chart.fg_color, display = display.
→all - display.pane)
```

The “,,highlight” macro generates a `bcolor()` call that highlights the chart pane’s background with a *conditional color* based on the variable or expression copied to the clipboard. For example, copying the `barstate.isrealtime` variable and typing “,,highlight” followed by a space will yield:

```
bcolor(bool(barstate.isrealtime) ? color.new(color.orange, 80) : na, title =
→"barstate.isrealtime highlight")
```

The “,,print” macro generates the one-line `printLabel()` function and creates an empty `printLabel()` call with the cursor placed inside it. All you need to do after typing “,,print” followed by a space is enter the text you want to display:

```
printLabel(string txt, float price = na) => int labelTime = math.max(last_bar_time,_
→chart.right_visible_bar_time), var label result = label.new(labelTime, na, txt,
→xloc.bar_time, na(price) ? yloc.abovebar : yloc.price, na, label.style_none, chart.
→fg_color, size.large), label.set_text(result, txt), label.set_y(result, price),
→result
printLabel()
```

The “,,tooltip” macro generates a `label.new()` call with a `tooltip` argument that uses `str.tostring()` on the clipboard’s contents. Copying the `variableName` variable and typing “,,tooltip” followed by a space yields:

```
label.new(bar_index, high, color = color.new(chart.fg_color, 70), tooltip = str.
→tostring(variableName))
```

The “,,log” macro generates a `log.info()` call with a `message` argument that uses `str.tostring()` on the clipboard’s contents to display string representations of variables and expressions in the *Pine Logs* pane. Copying the expression `bar_index % 2 == 0` and typing “,,log” followed by a space yields:

```
log.info(str.tostring(bar_index % 2 == 0))
```

### Note that:

- AHK is available for *Windows* devices. Research other software to employ a similar process if your machine uses a different operating system.





## 5.3 Profiling and optimization

- *Introduction*
- *Pine Profiler*
- *Optimization*
- *Tips*

### 5.3.1 Introduction

Pine Script™ is a cloud-based compiled language geared toward efficient repeated script execution. When a user adds a Pine script to a chart, it executes *numerous* times, once for each available bar or tick in the data feeds it accesses, as explained in this manual's [Execution model](#) page.

The Pine Script™ compiler automatically performs several internal optimizations to accommodate scripts of various sizes and help them run smoothly. However, such optimizations *do not* prevent performance bottlenecks in script executions. As such, it's up to programmers to [profile](#) a script's runtime performance and identify ways to modify critical code blocks and lines when they need to improve execution speeds.

This page covers how to profile and monitor a script's runtime and executions with the [Pine Profiler](#) and explains some ways programmers can modify their code to [optimize](#) runtime performance.

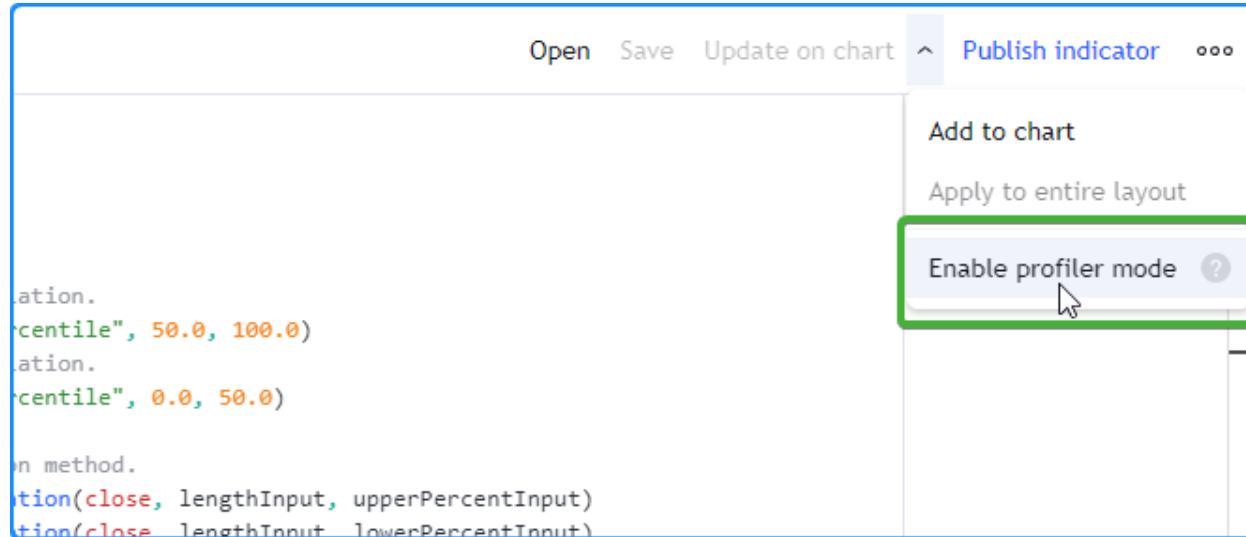
### 5.3.2 Pine Profiler

Before diving into [optimization](#), it's prudent to evaluate a script's runtime and pinpoint *bottlenecks*, i.e., areas in the code that substantially impact overall performance. With these insights, programmers can ensure they focus on optimizing where it truly matters instead of spending time and effort on low-impact code.

Enter the *Pine Profiler*, a powerful utility that analyzes the executions of all significant code lines and blocks in a script and displays helpful performance information next to the lines inside the Pine Editor. By inspecting the Profiler's results, programmers can gain a clearer perspective on a script's overall runtime, the distribution of runtime across its significant code regions, and the critical portions that may need extra attention and optimization.

### Profiling a script

The Pine Profiler can analyze the runtime performance of any *editable* script coded in Pine Script™ v5. To profile a script, add it to the chart, open the source code in the Pine Editor, and select “Enable profiler mode” from the dropdown next to the “Add to chart/Update on chart” option in the top-right corner:



We will use the script below for our initial profiling example, which calculates a custom oscillator based on average distances from the `close` price to upper and lower `percentiles` over `lengthInput` bars. It includes a few different types of *significant* code regions, which come with some differences in *interpretation* while profiling:

```

1 // @version=5
2 indicator("Pine Profiler demo")
3
4 //@variable The number of bars in the calculations.
5 int lengthInput = input.int(100, "Length", 2)
6 //@variable The percentage for upper percentile calculation.
7 float upperPercentInput = input.float(75.0, "Upper percentile", 50.0, 100.0)
8 //@variable The percentage for lower percentile calculation.
9 float lowerPercentInput = input.float(25.0, "Lower percentile", 0.0, 50.0)
10
11 // Calculate percentiles using the linear interpolation method.
12 float upperPercentile = ta.percentile_linear_interpolation(close, lengthInput, ↴
13     ↪upperPercentInput)
14 float lowerPercentile = ta.percentile_linear_interpolation(close, lengthInput, ↴
15     ↪lowerPercentInput)
16
17 // Declare arrays for upper and lower deviations from the percentiles on the same ↴
18 // line.
19 var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new ↴
20     <float>(lengthInput)
21
22 // Queue distance values through the `upperDistances` and `lowerDistances` arrays ↴
23 // based on excessive price deviations.
24 if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * ↴
25     ↪(upperPercentile - lowerPercentile)
26     array.push(upperDistances, math.max(close - upperPercentile, 0.0))
27     array.shift(upperDistances)
28     array.push(lowerDistances, math.max(lowerPercentile - close, 0.0))
29     array.shift(lowerDistances)

```

(continues on next page)

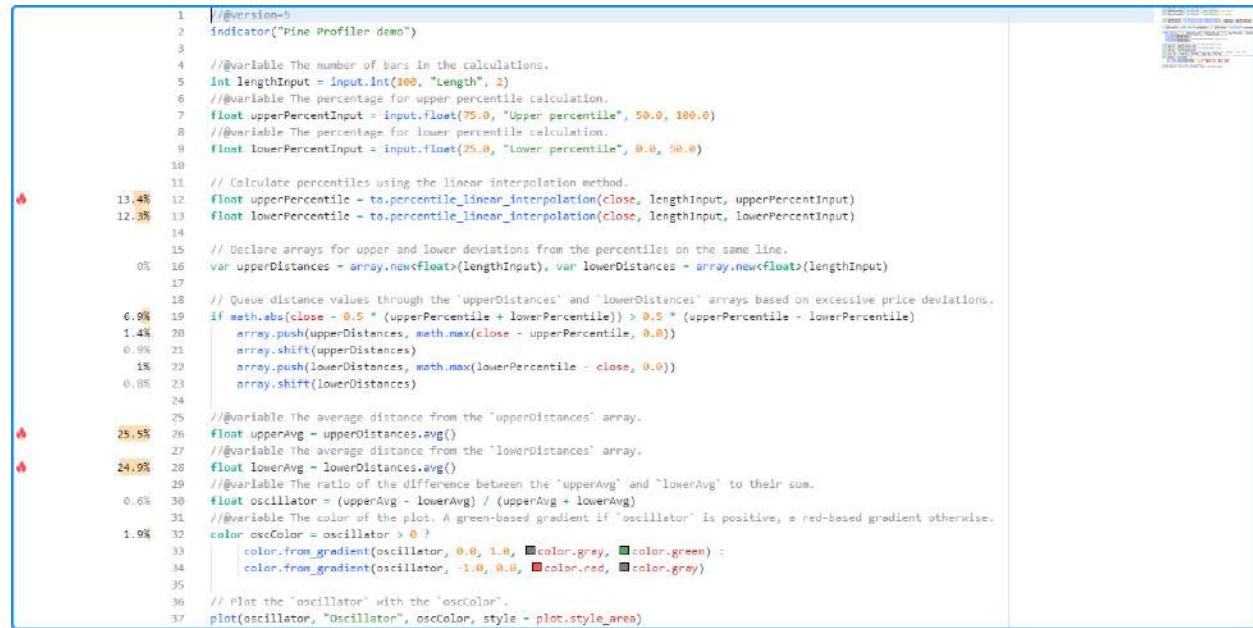
(continued from previous page)

```

24 // @variable The average distance from the `upperDistances` array.
25 float upperAvg = upperDistances.avg()
26 // @variable The average distance from the `lowerDistances` array.
27 float lowerAvg = lowerDistances.avg()
28 // @variable The ratio of the difference between the `upperAvg` and `lowerAvg` to
29 // their sum.
30 float oscillator = (upperAvg - lowerAvg) / (upperAvg + lowerAvg)
31 // @variable The color of the plot. A green-based gradient if `oscillator` is positive,
32 // a red-based gradient otherwise.
33 color oscColor = oscillator > 0 ?
34     color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) :
35     color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray)
36
37 // Plot the `oscillator` with the `oscColor`.
38 plot(oscillator, "Oscillator", oscColor, style = plot.style_area)

```

Once enabled, the Profiler collects information from all executions of the script's significant code lines and blocks, then displays bars and approximate runtime percentages to the left of the code lines inside the Pine Editor:



### Note that:

- The Profiler tracks every execution of a significant code region, including the executions on *realtime ticks*. Its information updates over time as new executions occur.
- Profiler results **do not** appear for script declaration statements, type declarations, other *insignificant* code lines such as variable declarations with no tangible impact, *unused code* that the script's outputs do not depend on, or *repetitive code* that the compiler optimizes during translation. See [this section](#) for more information.

When a script contains at least *four* significant lines of code, the Profiler will include “flame” icons next to the *top three* code regions with the highest performance impact. If one or more of the highest-impact code regions are *outside* the lines visible inside the Pine Editor, a “flame” icon and a number indicating how many critical lines are outside the view will appear at the top or bottom of the left margin. Clicking the icon will vertically scroll the Editor's window to show the nearest critical line:

The screenshot shows a Pine Script code editor with performance analysis annotations. A green arrow points to the first line of code, which is annotated with a tooltip showing execution time (0.9%), number of executions (20), and a small icon indicating it's a function call.

```

1 //@version=5
2 indicator("Pine Profiler demo")
3
4 //@variable The number of bars in the calculations.
5 int lengthInput = input.int(100, "Length", 2)
6 //@variable The percentage for upper percentile calculation.
7 float upperPercentInput = input.float(75.0, "Upper percentile", 50.0, 100.0)
8 //@variable The percentage for lower percentile calculation.
9 float lowerPercentInput = input.float(25.0, "Lower percentile", 0.0, 50.0)
10
11 // Calculate percentiles using the linear interpolation method.
12 float upperPercentile = ta.percentile_linear_interpolation(close, lengthInput, upperPercentInput)
13 float lowerPercentile = ta.percentile_linear_interpolation(close, lengthInput, lowerPercentInput)
14
15 // Declare arrays for upper and lower deviations from the percentiles on the same line.
16 var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new<float>(lengthInput)
17
18 // Queue distance values through the 'upperDistances' and 'lowerDistances' arrays based on excessive price deviations.
19 if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile)
20 | array.push(upperDistances, math.max(close - upperPercentile, 0.0))
21 | array.shift(lowerDistances)
22 | s, math.max(lowerPercentile - close, 0.0))
23 | es)
24 Time: 25.5% (26.7ms from 104.5ms total)
25 Executions: 20685
26 // @variable The average distance from the 'upperDistances' array.
27 float upperAvg = upperDistances.avg()
28 // @variable The average distance from the 'lowerDistances' array.
29 float lowerAvg = lowerDistances.avg()
30 // @variable The ratio of the difference between the 'upperAvg' and 'lowerAvg' to their sum.
31 float oscillator = (upperAvg - lowerAvg) / (upperAvg + lowerAvg)
32 // @variable The color of the plot. A green-based gradient if 'oscillator' is positive, a red-based gradient otherwise.
33 color oscColor = oscillator > 0 ? color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) :
34 : color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray);
35
36 // Plot the 'oscillator' with the 'oscColor'.
37 plot(oscillator, "Oscillator", oscColor, style + plot.style_area)

```

Hovering the mouse pointer over the space next to a line highlights the analyzed code and exposes a tooltip with additional information, including the time spent and the number of executions. The information shown next to each line and in the corresponding tooltip depends on the profiled code region. The [section below](#) explains different types of code the Profiler analyzes and how to interpret their performance results.

The screenshot shows a Pine Script code editor with performance analysis annotations. A green arrow points to the first line of code, which is annotated with a tooltip showing execution time (0.9%), number of executions (20), and a small icon indicating it's a function call.

```

1 //@version=5
2 indicator("Pine Profiler demo")
3
4 //@variable The number of bars in the calculations.
5 int lengthInput = input.int(100, "Length", 2)
6 //@variable The percentage for upper percentile calculation.
7 float upperPercentInput = input.float(75.0, "Upper percentile", 50.0, 100.0)
8 //@variable The percentage for lower percentile calculation.
9 float lowerPercentInput = input.float(25.0, "Lower percentile", 0.0, 50.0)
10
11 // Calculate percentiles using the linear interpolation method.
12 float upperPercentile = ta.percentile_linear_interpolation(close, lengthInput, upperPercentInput)
13 float lowerPercentile = ta.percentile_linear_interpolation(close, lengthInput, lowerPercentInput)
14
15 // Declare arrays for upper and lower deviations from the percentiles on the same line.
16 var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new<float>(lengthInput)
17
18 // Queue distance values through the 'upperDistances' and 'lowerDistances' arrays based on excessive price deviations.
19 if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile)
20 | array.push(upperDistances, math.max(close - upperPercentile, 0.0))
21 | array.shift(lowerDistances)
22 | s, math.max(lowerPercentile - close, 0.0))
23 | es)
24 Time: 25.5% (26.7ms from 104.5ms total)
25 Executions: 20685
26 // @variable The average distance from the 'upperDistances' array.
27 float upperAvg = upperDistances.avg()
28 // @variable The average distance from the 'lowerDistances' array.
29 float lowerAvg = lowerDistances.avg()
30 // @variable The ratio of the difference between the 'upperAvg' and 'lowerAvg' to their sum.
31 float oscillator = (upperAvg - lowerAvg) / (upperAvg + lowerAvg)
32 // @variable The color of the plot. A green-based gradient if 'oscillator' is positive, a red-based gradient otherwise.
33 color oscColor = oscillator > 0 ? color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) :
34 : color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray);
35
36 // Plot the 'oscillator' with the 'oscColor'.
37 plot(oscillator, "Oscillator", oscColor, style + plot.style_area)

```

**Note:** As with profiling tools for other languages, the Pine Profiler *wraps* a script and its significant code with *extra calculations* required to collect performance data. As such, a script's resource usage **increases** while profiling, and the Profiler's results reflect the script's runtime with those calculations included. Therefore, one should interpret the results as **estimates** rather than precise performance measurements.

Furthermore, the Profiler cannot collect and display individual performance data for the *internal calculations* that also affect runtime, including the calculations required to track performance, meaning the time values shown for all a script's code regions **will not** add up to exactly 100% of its overall runtime.

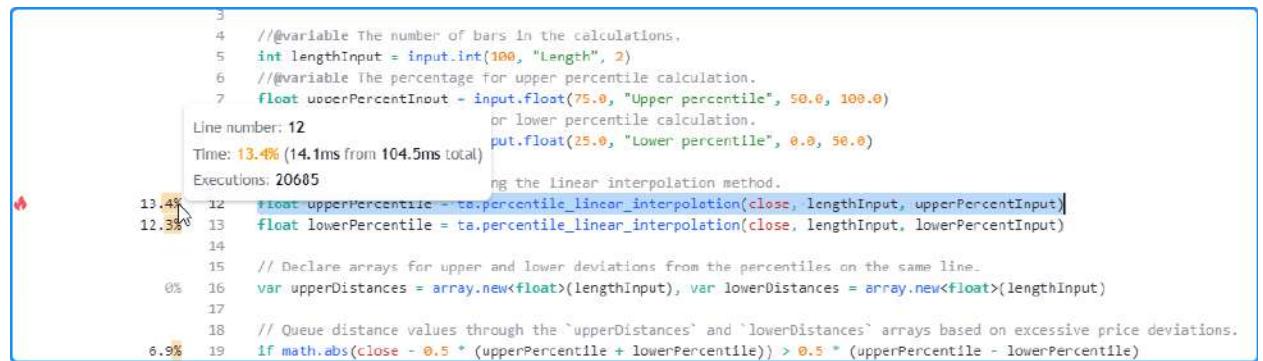
## Interpreting profiled results

### Single-line results

For a code line containing single-line expressions, the Profiler bar and displayed percentage represent the relative portion of the script's total runtime spent on that line. The corresponding tooltip displays three fields:

- The “Line number” field indicates the analyzed code line.
- The “Time” field shows the runtime percentage for the line of code, the runtime spent on that line, and the script’s total runtime.
- The “Executions” field shows the number of times that specific line executed while running the script.

Here, we hovered the pointer over the space next to line 12 of our profiled code to view its tooltip:



```

3
4 //variable The number of bars in the calculations,
5 int lengthInput = input.int(100, "Length", 2)
6 //variable The percentage for upper percentile calculation.
7 float upperPercentileInput = input.float(75.0, "Upper percentile", 50.0, 100.0)
Line number: 12
Time: 13.4% (14.1ms from 104.5ms total)
Executions: 20685
13.4% 12 float upperPercentile = ta.percentile_linear_interpolation(close, lengthInput, upperPercentileInput)
12.3% 13 float lowerPercentile = ta.percentile_linear_interpolation(close, lengthInput, lowerPercentileInput)
14
15 // Declare arrays for upper and lower deviations from the percentiles on the same line.
16 var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new<float>(lengthInput)
17
18 // Queue distance values through the `upperDistances` and `lowerDistances` arrays based on excessive price deviations.
19 if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile)

```

```

float upperPercentile = ta.percentile_linear_interpolation(close, lengthInput,_
    ↴upperPercentileInput)

```

#### Note that:

- The time information for the line represents the time spent completing *all* executions, **not** the time spent on a single execution.
- To estimate the *average* time spent per execution, divide the line’s time by the number of executions. In this case, the tooltip shows that line 12 took about 14.1 milliseconds to execute 20,685 times, meaning the average time per execution was approximately  $14.1 \text{ ms} / 20685 = 0.0006816534$  milliseconds (0.6816534 microseconds).

When a line of code consists of more than one expression separated by commas, the number of executions shown in the tooltip represents the *sum* of each expression’s total executions, and the time value displayed represents the total time spent evaluating all the line’s expressions.

For instance, this global line from our initial example includes two *variable declarations* separated by commas. Each uses the `var` keyword, meaning the script only executes them once on the first available bar. As we see in the Profiler tooltip for the line, it counted *two* executions (one for each expression), and the time value shown is the *combined* result from both expressions on the line:

```

5 int lengthInput = input.int(100, "Length", 2)
6 //@variable The percentage for upper percentile calculation.
7 float upperPercentileInput = input.float(75.0, "Upper percentile", 50.0, 100.0)
8 //@variable The percentage for lower percentile calculation.
9 float lowerPercentileInput = input.float(25.0, "Lower percentile", 0.0, 50.0)
10
11 // Calculate percentiles using the linear interpolation method.
12.4% Line number: 16 ta.percentile_linear_interpolation(close, lengthInput, upperPercentileInput)
12.3% ta.percentile_linear_interpolation(close, lengthInput, lowerPercentileInput)
Time: 0% (23mcs from 104.5ms total)
Executions: 2
13.4%     upper and lower deviations from the percentiles on the same line.
13.3% 10 var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new<float>(lengthInput)
11.3%
17
18 // Queue distance values through the "upperDistances" and "lowerDistances" arrays based on excessive price deviations.
6.9% 19 if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile)
1.4% 20     array.push(upperDistances, math.max(close - upperPercentile, 0.0))
0.9% 21     array.shift(upperDistances)

```

```

var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new
-><float>(lengthInput)

```

### Note that:

- When analyzing scripts with more than one expression on the same line, we recommend moving each expression to a *separate line* for more detailed insights while profiling, namely if they may contain *higher-impact* calculations.

When using *line wrapping* for readability or stylistic purposes, the Profiler considers all portions of a wrapped line as part of the *first line* where it starts in the Pine Editor.

For example, although this code from our initial script occupies more than one line in the Pine Editor, it's still treated as a *single* line of code, and the Profiler tooltip displays single-line results, with the “Line number” field showing the *first* line in the Editor that the wrapped line occupies:

```

23
24
25 // @variable The average distance from the "upperDistances" array.
25.5% 26 float upperAvg = upperDistances.avg()
27 // @variable The average distance from the "lowerDistances" array.
24.9% 28 float lowerAvg = lowerDistances.avg()
Line number: 32
24.9%     stances.avg()
Time: 1.9% (2ms from 104.5ms total)
0.6%     the difference between the "upperAvg" and "lowerAvg" to their sum.
0.6%     ~Avg - lowerAvg) / (upperAvg + lowerAvg)
Executions: 20685
0.6%     the plot. A green-based gradient if "oscillator" is positive, a red-based gradient otherwise.
1.9% 32 color oscColor = oscillator > 0 ?
1.9%     color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) :
33     color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray)
34
35
36 // Plot the "oscillator" with the "oscColor".
37 plot(oscillator, "Oscillator", oscColor, style = plot.style_area)
38

```

```

color oscColor = oscillator > 0 ?
    color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) :
    color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray)

```

### Code block results

For a line at the start of a *loop* or *conditional structure*, the Profiler bar and percentage represent the relative portion of the script's runtime spent on the **entire code block**, not just the single line. The corresponding tooltip displays four fields:

- The “Code block range” field indicates the range of lines included in the structure.
- The “Time” field shows the code block's runtime percentage, the time spent on all block executions, and the script's total runtime.
- The “Line time” field shows the runtime percentage for the block's initial line, the time spent on that line, and the script's total runtime. The interpretation differs for *switch* blocks or *if* blocks *with else* statements, as the values

represent the total time spent on **all** the structure's conditional statements. See below for more information.

- The “Executions” field shows the number of times the code block executed while running the script.

Here, we hovered over the space next to line 19 in our initial script, the beginning of a simple `if` structure *without else if* statements. As we see below, the tooltip shows performance information for the entire code block and the current line:

The screenshot shows a code editor with Pine Script syntax highlighting. A tooltip is displayed over line 19, which contains the condition for the `if` statement. The tooltip provides the following information:

- Code block range: [19,25]**
- Time: 6.9% (7.2ms from 104.5ms total)**
- Line time: 2.9% (3ms from 104.5ms total)**
- Executions: 20685**
- Description:** through the `upperDistances` and `lowerDistances` arrays based on excessive price deviations.

The code block itself is as follows:

```
if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 *_
    (upperPercentile - lowerPercentile)
    array.push(upperDistances, math.max(close - upperPercentile, 0.0))
    array.shift(upperDistances)
    array.push(lowerDistances, math.max(lowerPercentile - close, 0.0))
    array.shift(lowerDistances)
```

The screenshot shows a code editor with Pine Script syntax highlighting. A tooltip is displayed over line 20, which contains the first part of the `array.push` statement. The tooltip provides the following information:

- Line number: 20**
- Time: 1.4% (1.5ms from 104.5ms total)**
- Executions: 13199**
- Description:** through the `upperDistances` and `lowerDistances` arrays based on excessive price deviations.

The code line is:

```
array.push(upperDistances, math.max(close - upperPercentile, 0.0))
```

#### Note that:

- The “Time” field shows that the total time spent evaluating the structure 20,685 times was 7.2 milliseconds.
- The “Line time” field indicates that the runtime spent on the *first line* of this `if` structure was about 3 milliseconds.

Users can also inspect the results from lines and nested blocks within a code block's range to gain more granular performance insights. Here, we hovered over the space next to line 20 within the code block to view its *single-line result*:

The screenshot shows a code editor with Pine Script syntax highlighting. A tooltip is displayed over line 20, which contains the first part of the `array.push` statement. The tooltip provides the following information:

- Line number: 20**
- Time: 1.4% (1.5ms from 104.5ms total)**
- Executions: 13199**
- Description:** through the `upperDistances` and `lowerDistances` arrays based on excessive price deviations.

The code line is:

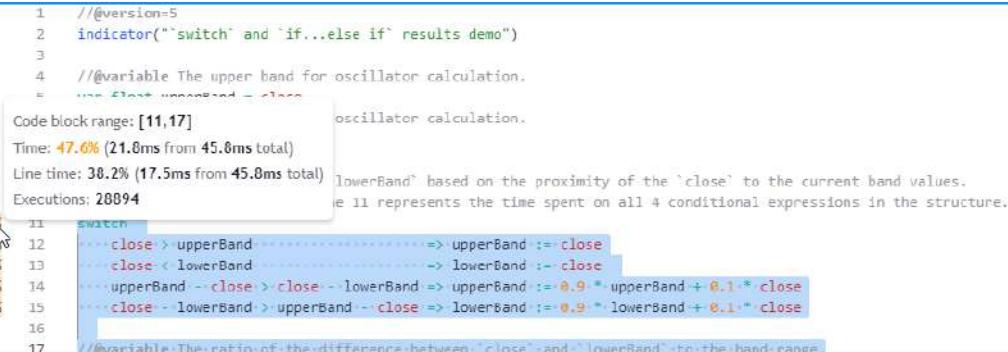
```
array.push(upperDistances, math.max(close - upperPercentile, 0.0))
```

#### Note that:

- The number of executions shown is *less than* the result for the entire code block, as the condition that controls the execution of this line does not return `true` all the time. The opposite applies to the code inside `loops` since each execution of a loop statement can trigger **several** executions of the loop's local block.

When profiling a `switch` structure or an `if` structure that includes `else if` statements, the “Line time” field will show the time spent executing **all** the structure's conditional expressions, **not** just the block's first line. The results for the lines inside the code block range will show runtime and executions for each **local block**. This format is necessary for these structures due to the Profiler's calculation and display constraints. See [this section](#) for more information.

For example, the “Line time” for the `switch` structure in this script represents the time spent evaluating *all four* conditional statements within its body, as the Profiler *cannot* track them separately. The results for each line in the code block’s range represent the performance information for each *local block*:



```

1 //version=5
2 indicator(`switch` and `if...else if` results demo")
3
4 //@variable The upper band for oscillator calculation.
5 var float upperBand = close
6 //@variable The lower band for oscillator calculation.
7 var float lowerBand = close
8
9 // Update the `upperBand` and `lowerBand` based on the proximity of the `close` to
10 // the current band values.
11 // The "Line time" field on line 11 represents the time spent on all 4 conditional
12 // expressions in the structure.
13 switch
14     close > upperBand          => upperBand := close
15     close < lowerBand          => lowerBand := close
16     upperBand - close > close - lowerBand => upperBand := 0.9 * upperBand + 0.1 * close
17     close - lowerBand > upperBand - close => lowerBand := 0.9 * lowerBand + 0.1 * close
18
19
20
21
22
23
24

```

When the conditional logic in such structures involves significant calculations, programmers may require more granular performance information for each calculated condition. An effective way to achieve this analysis is to use *nested if* blocks instead of the more compact `switch` or `if...else if` structures. For example, instead of:

```

switch
<expression1> => <localBlock1>
<expression2> => <localBlock2>
=>           <localBlock3>

```

or:

```

if <expression1>
    <localBlock1>
else if <expression2>
    <localBlock2>
else
    <localBlock3>

```

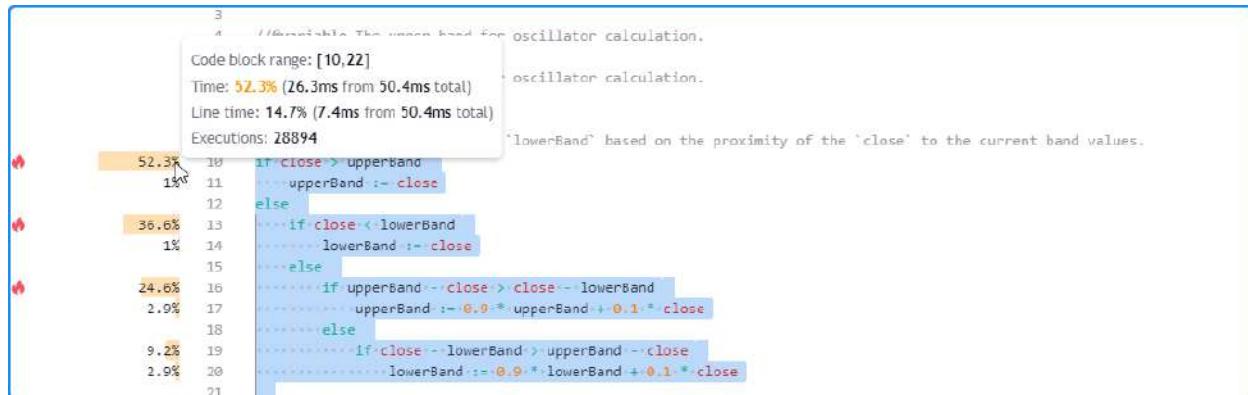
one can use nested `if` blocks for more in-depth profiling while maintaining the same logical flow:

```

if <expression1>
    <localBlock1>
else
    if <expression2>
        <localBlock2>
    else
        <localBlock3>

```

Below, we changed the previous `switch` example to an equivalent nested `if` structure. Now, we can view the runtime and executions for each significant part of the conditional pattern individually:



```

1 // @version=5
2 indicator(`switch` and `if...else if` results demo")
3
4 //@variable The upper band for oscillator calculation.
5 var float upperBand = close
6 //@variable The lower band for oscillator calculation.
7 var float lowerBand = close
8
9 // Update the `upperBand` and `lowerBand` based on the proximity of the `close` to
10 // the current band values.
11 if close > upperBand
12     upperBand := close
13 else
14     if close < lowerBand
15         lowerBand := close
16     else
17         if upperBand - close > close - lowerBand
18             upperBand := 0.9 * upperBand + 0.1 * close
19         else
20             if close - lowerBand > upperBand - close
21                 lowerBand := 0.9 * lowerBand + 0.1 * close
22
23 //@variable The ratio of the difference between `close` and `lowerBand` to the band_

```

(continues on next page)

(continued from previous page)

```

1  ↵range.
2  float oscillator = 100.0 * (close - lowerBand) / (upperBand - lowerBand)
3
4
5 // Plot the `oscillator` as columns with a dynamic color.
6 plot(
7     oscillator, "Oscillator", oscillator > 50.0 ? color.teal : color.maroon,
8     style = plot.style_columns, histbase = 50.0
9 )

```

### Note that:

- This same process can also apply to [ternary expressions](#). When a complex ternary expression's operands contain significant calculations, reorganizing the logic into a nested `if` structure allows more detailed Profiler results, making it easier to spot critical parts.

## User-defined function calls

[User-defined functions](#) and [methods](#) are functions written by users. They encapsulate code sequences that a script may execute several times. Users often write functions and methods for improved code modularity, reusability, and maintainability.

The indented lines of code within a function represent its *local scope*, i.e., the sequence that executes *each time* the script calls it. Unlike code in a script's global scope, which a script evaluates once on each execution, the code inside a function may activate zero, one, or *multiple times* on each script execution, depending on the conditions that trigger the calls, the number of calls that occur, and the function's logic.

This distinction is crucial to consider while interpreting Profiler results. When a profiled code contains [user-defined function](#) or [method](#) calls:

- The results for each *function call* reflect the runtime allocated toward it and the total number of times the script activated that specific call.
- The time and execution information for all local code *inside* a function's scope reflects the combined results from **all** calls to the function.

This example contains a user-defined `similarity()` function that estimates the similarity of two series, which the script calls only *once* from the global scope on each execution. In this case, the Profiler's results for the code inside the function's body correspond to that specific call:

```

1  //@version=5
2  indicator("User-defined function calls demo")
3
4  //Function Estimates the similarity between two standardized series over `length` bars.
5  // Line number: 9
6  // Time: 14.7% (7.1ms from 48.5ms total)
7  // Executions: 27581
8
9  float normB = (sourceA - ta.sma(sourceA, length)) / ta.stdev(sourceA, length)
10 // Calculate and return the estimated similarity of "normA" and "normB".
11 float abSum = math.sum(normA * normB, length)
12 float a2Sum = math.sum(normA * normA, length)
13 float b2Sum = math.sum(normB * normB, length)
14 abSum / math.sqrt(a2Sum * b2Sum)
15
16 // Plot the similarity between the `close` and an offset `close` series.
17 plot(similarity(close, close[1], 100), "Similarity 1", color.red)
18

```

```

1  //@version=5
2  indicator("User-defined function calls demo")
3

```

(continues on next page)

(continued from previous page)

```

4 // @function Estimates the similarity between two standardized series over `length` ↵
5 // bars.
6 //           Each individual call to this function activates its local scope.
7 similarity(float sourceA, float sourceB, int length) =>
8     // Standardize `sourceA` and `sourceB` for comparison.
9     float normA = (sourceA - ta.sma(sourceA, length)) / ta.stdev(sourceA, length)
10    float normB = (sourceB - ta.sma(sourceB, length)) / ta.stdev(sourceB, length)
11    // Calculate and return the estimated similarity of `normA` and `normB`.
12    float abSum = math.sum(normA * normB, length)
13    float a2Sum = math.sum(normA * normA, length)
14    float b2Sum = math.sum(normB * normB, length)
15    abSum / math.sqrt(a2Sum * b2Sum)
16
17 // Plot the similarity between the `close` and an offset `close` series.
18 plot(similarity(close, close[1], 100), "Similarity 1", color.red)

```

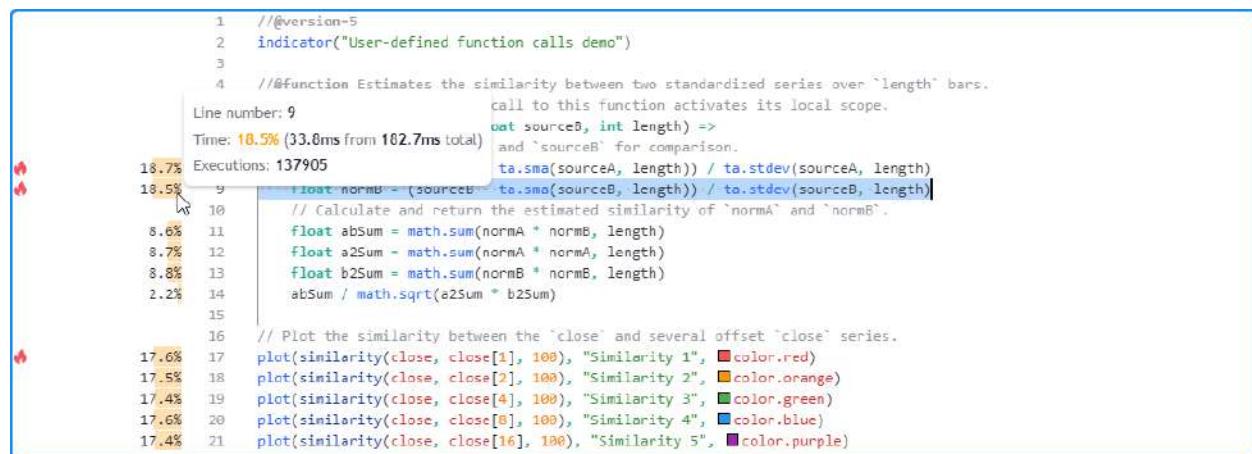
Let's increase the number of times the script calls the function each time it executes. Here, we changed the script to call our *user-defined function five times*:

```

1 // @version=5
2 indicator("User-defined function calls demo")
3
4 // @function Estimates the similarity between two standardized series over `length` ↵
5 // bars.
6 //           Each individual call to this function activates its local scope.
7 similarity(float sourceA, float sourceB, int length) =>
8     // Standardize `sourceA` and `sourceB` for comparison.
9     float normA = (sourceA - ta.sma(sourceA, length)) / ta.stdev(sourceA, length)
10    float normB = (sourceB - ta.sma(sourceB, length)) / ta.stdev(sourceB, length)
11    // Calculate and return the estimated similarity of `normA` and `normB`.
12    float abSum = math.sum(normA * normB, length)
13    float a2Sum = math.sum(normA * normA, length)
14    float b2Sum = math.sum(normB * normB, length)
15    abSum / math.sqrt(a2Sum * b2Sum)
16
17 // Plot the similarity between the `close` and several offset `close` series.
18 plot(similarity(close, close[1], 100), "Similarity 1", color.red)
19 plot(similarity(close, close[2], 100), "Similarity 2", color.orange)
20 plot(similarity(close, close[4], 100), "Similarity 3", color.green)
21 plot(similarity(close, close[8], 100), "Similarity 4", color.blue)
22 plot(similarity(close, close[16], 100), "Similarity 5", color.purple)

```

In this case, the local code results no longer correspond to a *single* evaluation per script execution. Instead, they represent the *combined* runtime and executions of the local code from **all five** calls. As we see below, the results after running this version of the script across the same data show 137,905 executions of the local code, *five times* the number from when the script only contained one `similarity()` function call:



**Note:** When the local scopes of a script's *user-defined functions* or *methods* contain calls to `request.*()` functions, the *translated form* of the script extracts such calls **outside** the functions' scopes and evaluates them **separately**. Consequently, the Profiler's results for lines with calls to those *user-defined functions* **will not** include the time spent on the `request.*()` calls. See the [section below](#) to learn more.

---

### When requesting other contexts

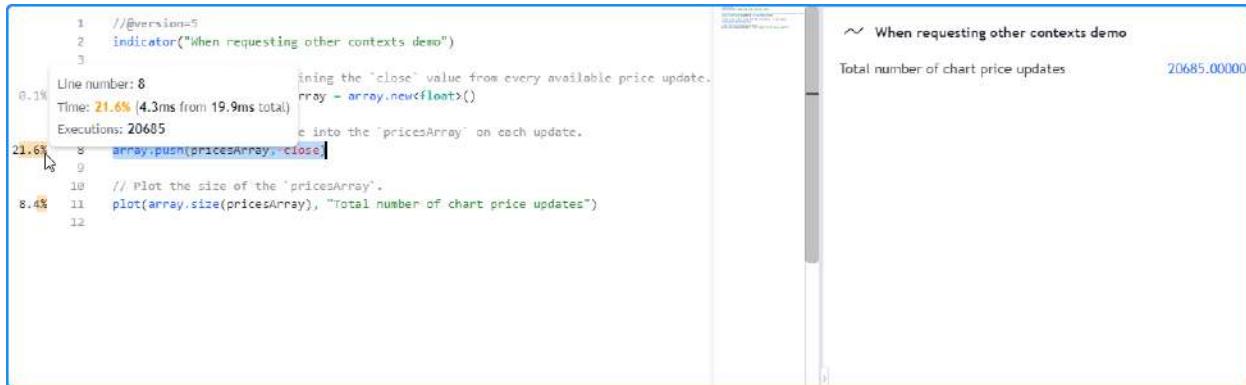
Pine scripts can request data from other *contexts*, i.e., different symbols, timeframes, or data modifications than what the chart's data uses by calling the `request.*()` family of functions or specifying an alternate `timeframe` in the `indicator()` declaration statement.

When a script requests data from another context, it evaluates all required scopes and calculations within that context, as explained in the [Other timeframes and data](#) page. This behavior can affect the runtime of a script's code regions and the number of times they execute.

The Profiler information for any code *line* or *block* represents the results from executing the code in *all necessary contexts*, which may or may not include the chart's data. Pine Script™ determines which contexts to execute code within based on the calculations required by a script's data requests and outputs.

Let's look at a simple example. This initial script only uses the chart's data for its calculations. It declares a `pricesArray` variable with the `varip` keyword, meaning the array assigned to it persists across the data's history and all available realtime ticks. On each execution, the script calls `array.push()` to push a new `close` value into the `array`, and it *plots* the array's size.

After profiling the script across all the bars on an intraday chart, we see that the number of elements in the `pricesArray` corresponds to the number of executions the Profiler shows for the `array.push()` call on line 8:



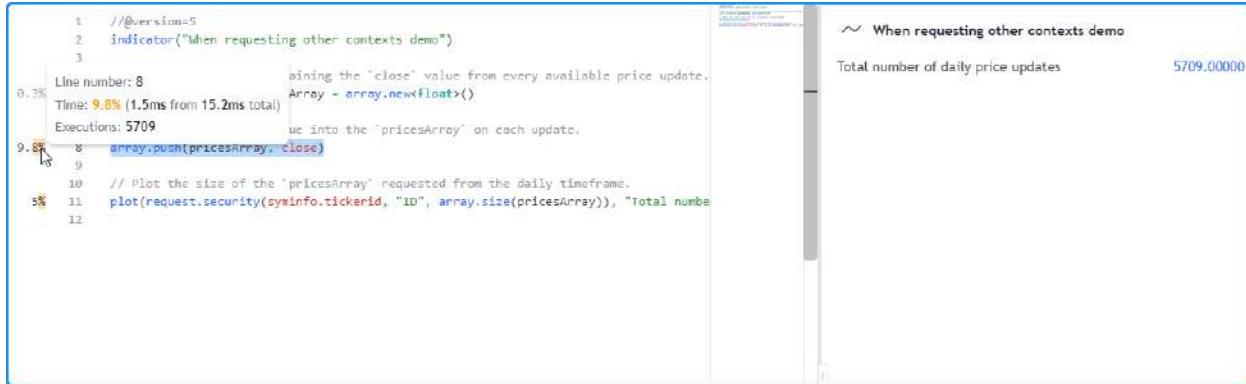
```

1 //@version=5
2 indicator("When requesting other contexts demo")
3
4 //@variable An array containing the `close` value from every available price update.
5 varip array<float> pricesArray = array.new<float>()
6
7 // Push a new `close` value into the `pricesArray` on each update.
8 array.push(pricesArray, close)
9
10 // Plot the size of the `pricesArray`.
11 plot(array.size(pricesArray), "Total number of chart price updates")

```

Now, let's try evaluating the size of the `pricesArray` from *another context* instead of using the chart's data. Below, we've added a `request.security()` call with `array.size(pricesArray)` as its expression argument to retrieve the value calculated on the “*ID*” timeframe and plotted that result instead.

In this case, the number of executions the Profiler shows on line 8 still corresponds to the number of elements in the `pricesArray`. However, it did not execute the same number of times since the script did not require the *chart's data* in the calculations. It only needed to initialize the `array` and evaluate `array.push()` across all the requested *daily data*, which has a different number of price updates than our current intraday chart:



```

1 //@version=5
2 indicator("When requesting other contexts demo")
3
4 //@variable An array containing the `close` value from every available price update.
5 varip array<float> pricesArray = array.new<float>()
6
7 // Push a new `close` value into the `pricesArray` on each update.
8 array.push(pricesArray, close)
9

```

(continues on next page)

(continued from previous page)

```
10 // Plot the size of the `pricesArray` requested from the daily timeframe.
11 plot(request.security(syminfo.tickerid, "1D", array.size(pricesArray)), "Total number
  ↪of daily price updates")
```

### Note that:

- The requested EOD data in this example had fewer data points than our intraday chart, so the `array.push()` call required fewer executions in this case. However, EOD feeds *do not* have history limitations, meaning it's also possible for requested HTF data to span **more** bars than a user's chart, depending on the timeframe, the data provider, and the user's [plan](#).

If this script were to plot the `array.size()` value directly in addition to the requested daily value, it would then require the creation of *two arrays* (one for each context) and the execution of `array.push()` across both the chart's data *and* the data from the daily timeframe. As such, the declaration on line 5 will execute *twice*, and the results on line 8 will reflect the time and executions accumulated from evaluating the `array.push()` call across **both separate datasets**:

When requesting other contexts demo	
Total number of daily price updates	5709.00000
Total number of chart price updates	20685.00000

```
1 //@version=5
2 indicator("When requesting other contexts demo")
3
4 //@variable An array containing the `close` value from every available price update.
5 varip array<float> pricesArray = array.new<float>()
6
7 // Push a new `close` value into the `pricesArray` on each update.
8 array.push(pricesArray, close)
9
10 // Plot the size of the `pricesArray` from the daily timeframe and the chart's
  ↪context.
11 // Including both in the outputs requires executing line 5 and line 8 across BOTH
  ↪datasets.
12 plot(request.security(syminfo.tickerid, "1D", array.size(pricesArray)), "Total number
  ↪of daily price updates")
13 plot(array.size(pricesArray), "Total number of chart price updates")
```

It's important to note that when a script calls a *user-defined function* or *method* that contains `request.*()` calls in its local scope, the script's *translated form* extracts the `request.*()` calls **outside** the scope and encapsulates the expressions they depend on within **separate functions**. When the script executes, it evaluates the required `request.*()` calls first, then *passes* the requested data to a *modified form* of the *user-defined function*.

Since the translated script executes a *user-defined function's* data requests separately **before** evaluating non-requested calculations in its local scope, the Profiler's results for lines containing calls to the function **will not** include the time spent on its `request.*()` calls or their required expressions.

As an example, the following script contains a user-defined `getCompositeAvg()` function with a `request.security()` call that requests the `math.avg()` of 10 `ta.wma()` calls with different length arguments from a specified symbol. The

script uses the function to request the average result using a [Heikin Ashi](#) ticker ID:

```

1 //@version=5
2 indicator("User-defined functions with `request.*()` calls demo", overlay = true)
3
4 int multInput = input.int(10, "Length multiplier", 1)
5
6 string tickerID = ticker.heikinashi(syminfo.tickerid)
7
8 getCompositeAvg(string symbol, int lengthMult) =>
9     request.security(
10         symbol, timeframe.period, math.avg(
11             ta.wma(close, lengthMult), ta.wma(close, 2 * lengthMult), ta.wma(close, ↵
12             3 * lengthMult),
13             ta.wma(close, 4 * lengthMult), ta.wma(close, 5 * lengthMult), ta. ↵
14             wma(close, 6 * lengthMult),
15             ta.wma(close, 7 * lengthMult), ta.wma(close, 8 * lengthMult), ta. ↵
16             wma(close, 9 * lengthMult),
17             ta.wma(close, 10 * lengthMult)
18         )
19     )
20
21 plot(getCompositeAvg(tickerID, multInput), "Composite average", linewidth = 3)

```

After profiling the script, users might be surprised to see that the runtime results shown inside the function's body heavily **exceed** the results shown for the *single* `getCompositeAvg()` call:



The results appear this way since the translated script includes internal modifications that *moved* the `request.security()` call and its expression **outside** the function's scope, and the Profiler has no way to represent the results from those calculations other than displaying them next to the `request.security()` line in this scenario. The code below roughly illustrates how the translated script looks:

```

1 //@version=5
2 indicator("User-defined functions with `request.*()` calls demo", overlay = true)
3
4 int multInput = input.int(10, "Length multiplier")
5
6 string tickerID = ticker.heikinashi(syminfo.tickerid)
7
8 secExpr(int lengthMult)=>
9     math.avg(
10         ta.wma(close, lengthMult), ta.wma(close, 2 * lengthMult), ta.wma(close, 3 * ↵
11         lengthMult),
12         ta.wma(close, 4 * lengthMult), ta.wma(close, 5 * lengthMult), ta.wma(close, ↵
13         (continues on next page)

```

(continued from previous page)

```

12     ↵6 * lengthMult),
13         ta.wma(close, 7 * lengthMult), ta.wma(close, 8 * lengthMult), ta.wma(close, ↵
14     ↵9 * lengthMult),
15         ta.wma(close, 10 * lengthMult)
16     )
17
18 float sec = request.security(tickerID, timeframe.period, secExpr(multInput))
19
20 getCompositeAvg(float s) =>
21     s
22
23 plot(getCompositeAvg(sec), "Composite average", linewidth = 3)

```

**Note that:**

- The `secExpr()` code represents the *separate function* used by `request.security()` to calculate the required expression in the requested context.
- The `request.security()` call takes place in the **outer scope**, outside the `getCompositeAvg()` function.
- The translation substantially reduced the local code of `getCompositeAvg()`. It now solely returns a value passed into it, as all the function's required calculations take place **outside** its scope. Due to this reduction, the function call's performance results **will not** reflect any of the time spent on the data request's required calculations.

**Insignificant, unused, and redundant code**

When inspecting a profiled script's results, it's crucial to understand that *not all* code in a script necessarily impacts runtime performance. Some code has no direct performance impact, such as a script's declaration statement and `type` declarations. Other code regions with insignificant expressions, such as most `input.*()` calls, variable references, or `variable declarations` without significant calculations, have little to *no effect* on a script's runtime. Therefore, the Profiler will **not** display performance results for these types of code.

Additionally, Pine scripts do not execute code regions that their *outputs* (`plots`, `drawings`, `alerts`, `logs`, etc.) do not depend on, as the compiler automatically **removes** them during translation. Since unused code regions have *zero* impact on a script's performance, the Profiler will **not** display any results for them.

The following example contains a `barsInRange` variable and a `for` loop that adds 1 to the variable's value for each historical `close` price between the current `high` and `low` over `lengthInput` bars. However, the script **does not use** these calculations in its outputs, as it only *plots* the `close` price. Consequently, the script's compiled form **discards** that unused code and only considers the `plot(close)` call.

The Profiler does not display **any** results for this script since it does not execute any **significant** calculations:

```

1  //version=5
2  indicator("Unused code demo")
3
4  //@variable The number of historical bars in the calculation.
5  int lengthInput = input.int(100, "Length", 1)
6
7  //@variable The number of closes over `lengthInput` bars between the current bar's `high` and `low`.
8  int barsInRange = 0
9
10 for i = 1 to lengthInput
11     //@variable The `close` price from `i` bars ago.
12     float pastClose = close[i]
13     // Add 1 to `barsInRange` if the `pastClose` is between the current bar's `high` and `low`.
14     if pastClose > low and pastClose < high
15         barsInRange += 1
16
17 // Plot the `close` price. This is the only output.
18 // Since the outputs do not require any of the above calculations, the compiled script will not execute them.
19 plot(close)

```

```

1  //version=5
2  indicator("Unused code demo")
3
4  //@variable The number of historical bars in the calculation.
5  int lengthInput = input.int(100, "Length", 1)
6
7  //@variable The number of closes over `lengthInput` bars between the current bar's
8  ↵`high` and `low`.
9  int barsInRange = 0
10
11 for i = 1 to lengthInput
12     //@variable The `close` price from `i` bars ago.
13     float pastClose = close[i]
14     // Add 1 to `barsInRange` if the `pastClose` is between the current bar's `high` ↵
15     ↵and `low`.
16     if pastClose > low and pastClose < high
17         barsInRange += 1
18
19 // Plot the `close` price. This is the only output.
20 // Since the outputs do not require any of the above calculations, the compiled
21 ↵script will not execute them.
22 plot(close)

```

#### Note that:

- Although this script does not use the `input.int()` from line 5 and discards all its associated calculations, the “Length” input *will* still appear in the script’s settings, as the compiler **does not** completely remove unused *inputs*.

If we change the script to plot the `barsInRange` value instead, the declared variables and the `for` loop are no longer unused since the output depends on them, and the Profiler will now display performance information for that code:

```

1  //@version=5
2  indicator("Unused code demo")
3
4  //@variable The number of historical bars in the calculation.
5  int lengthInput = input.int(100, "Length", 1)
6
7  //@variable The number of closes over `lengthInput` bars between the current bar's "high" and "low".
8  int barsInRange = 0
9
10 99.7% for i = 1 to lengthInput
11    // @variable The `close` price from `i` bars ago.
12    float pastClose = close[i]
13    // Add 1 to `barsInRange` if the `pastClose` is between the current bar's "high" and "low".
14    if pastClose > low and pastClose < high
15      barsInRange += 1
16
17 // Plot the `barsInRange` value. The above calculations will execute since the output requires them.
18 plot(barsInRange, "Bars in range")

```

```

1 //@version=5
2 indicator("Unused code demo")
3
4 //@variable The number of historical bars in the calculation.
5 int lengthInput = input.int(100, "Length", 1)
6
7 //@variable The number of closes over `lengthInput` bars between the current bar's
8 ↪`high` and `low`.
9 int barsInRange = 0
10
11 for i = 1 to lengthInput
12   // @variable The `close` price from `i` bars ago.
13   float pastClose = close[i]
14   // Add 1 to `barsInRange` if the `pastClose` is between the current bar's `high` ↪
15   ↪and `low`.
16   if pastClose > low and pastClose < high
17     barsInRange += 1
18
19 // Plot the `barsInRange` value. The above calculations will execute since the output
20 ↪requires them.
21 plot(barsInRange, "Bars in range")

```

**Note that:**

- The Profiler does not show performance information for the `lengthInput` declaration on line 5 or the `barsInRange` declaration on line 8 since the expressions on these lines do not impact the script's performance.

When possible, the compiler also simplifies certain instances of *redundant code* in a script, such as some forms of identical expressions with the same *fundamental type* values. This optimization allows the compiled script to only execute such calculations *once*, on the first occurrence, and *reuse* the calculated result for each repeated instance that the outputs depend on.

If a script contains repetitive code and the compiler simplifies it, the Profiler will only show results for the **first occurrence** of the code since that's the only time the script requires the calculation.

For example, this script contains a code line that plots the value of `ta.sma(close, 100)` and 12 code lines that plot the value of `ta.sma(close, 500)`:

```

1 //@version=5
2 indicator("Redundant calculations demo", overlay = true)
3
4 // Plot the 100-bar SMA of `close` values one time.
5 plot(ta.sma(close, 100), "100-bar SMA", color.teal, 3)

```

(continues on next page)

(continued from previous page)

```

6 // Plot the 500-bar SMA of `close` values 12 times. After compiler optimizations, ↵
7 // only the first `ta.sma(close, 500)` ↵
8 // call on line 9 requires calculation in this case. ↵
9 plot(ta.sma(close, 500), "500-bar SMA", #001aff, 12) ↵
10 plot(ta.sma(close, 500), "500-bar SMA", #4d0bff, 11) ↵
11 plot(ta.sma(close, 500), "500-bar SMA", #7306f7, 10) ↵
12 plot(ta.sma(close, 500), "500-bar SMA", #920be9, 9) ↵
13 plot(ta.sma(close, 500), "500-bar SMA", #ae11d5, 8) ↵
14 plot(ta.sma(close, 500), "500-bar SMA", #c618be, 7) ↵
15 plot(ta.sma(close, 500), "500-bar SMA", #db20a4, 6) ↵
16 plot(ta.sma(close, 500), "500-bar SMA", #eb2c8a, 5) ↵
17 plot(ta.sma(close, 500), "500-bar SMA", #f73d6f, 4) ↵
18 plot(ta.sma(close, 500), "500-bar SMA", #fe5053, 3) ↵
19 plot(ta.sma(close, 500), "500-bar SMA", #ffe534, 2) ↵
20 plot(ta.sma(close, 500), "500-bar SMA", #ff7a00, 1)

```

Since the last 12 lines all contain identical `ta.sma()` calls, the compiler can automatically simplify the script so that it only needs to evaluate `ta.sma(close, 500)` once per execution rather than repeating the calculation 11 more times.

As we see below, the Profiler only shows results for lines 5 and 9. These are the only parts of the code requiring significant calculations since the `ta.sma()` calls on lines 10-20 are redundant in this case:

```

1 //@version=5
2 indicator("Redundant calculations demo", overlay = true)
3
4 // Plot the 100-bar SMA of `close` values one time.
5 plot(ta.sma(close, 100), "100-bar SMA", color.teal, 3)
6
7 // Plot the 500-bar SMA of `close` values 12 times. After compiler optimizations, only the first `ta.sma(close, 500)`
8 // call on line 9 requires calculation in this case.
9 plot(ta.sma(close, 500), "500-bar SMA", #001aff, 12) 23.1%
10 plot(ta.sma(close, 500), "500-bar SMA", #4d0bff, 11)
11 plot(ta.sma(close, 500), "500-bar SMA", #7306f7, 10)
12 plot(ta.sma(close, 500), "500-bar SMA", #920be9, 9)
13 plot(ta.sma(close, 500), "500-bar SMA", #ae11d5, 8)
14 plot(ta.sma(close, 500), "500-bar SMA", #c618be, 7)
15 plot(ta.sma(close, 500), "500-bar SMA", #db20a4, 6)
16 plot(ta.sma(close, 500), "500-bar SMA", #eb2c8a, 5)
17 plot(ta.sma(close, 500), "500-bar SMA", #f73d6f, 4)
18 plot(ta.sma(close, 500), "500-bar SMA", #fe5053, 3)
19 plot(ta.sma(close, 500), "500-bar SMA", #ffe534, 2)
20 plot(ta.sma(close, 500), "500-bar SMA", #ff7a00, 1)

```

Another type of repetitive code optimization occurs when a script contains two or more *user-defined functions* or *methods* with identical compiled forms. In such a case, the compiler simplifies the script by **removing** the redundant functions, and the script will treat all calls to the redundant functions as calls to the **first** defined version. Therefore, the Profiler will only show local code performance results for the *first* function since the discarded “clones” will never execute.

For instance, the script below contains two *user-defined functions*, `metallicRatio()` and `calcMetallic()`, that calculate a metallic ratio of a given order raised to a specified exponent:

```

1 //@version=5
2 indicator("Redundant functions demo")
3
4 //@variable Controls the base ratio for the `calcMetallic()` call.
5 int order1Input = input.int(1, "Order 1", 1)
6 //@variable Controls the base ratio for the `metallicRatio()` call.
7 int order2Input = input.int(2, "Order 2", 1)
8
9 //@function Calculates the value of a metallic ratio with a given `order`, ↵
10 // raised to a specified `exponent`.
11 //param order Determines the base ratio used. 1 = Golden Ratio, 2 = Silver Ratio, ↵

```

(continues on next page)

(continued from previous page)

```

11  ↵3 = Bronze Ratio, and so on.
12 // @param exponent The exponent applied to the ratio.
13 metallicRatio(int order, float exponent) =>
14     math.pow((order + math.sqrt(4.0 + order * order)) * 0.5, exponent)
15
16 // @function      A function with the same signature and body as `metallicRatio()` .
17 //                  The script discards this function and treats `calcMetallic()` as an
18 // alias for `metallicRatio()` .
19 calcMetallic(int ord, float exp) =>
20     math.pow((ord + math.sqrt(4.0 + ord * ord)) * 0.5, exp)
21
22 // Plot the results from a `calcMetallic()` and `metallicRatio()` call.
23 plot(calcMetallic(order1Input, bar_index % 5), "Ratio 1", color.orange, 3)
24 plot(metallicRatio(order2Input, bar_index % 5), "Ratio 2", color.maroon)

```

Despite the differences in the function and parameter names, the two functions are otherwise identical, which the compiler detects while translating the script. In this case, it **discards** the redundant `calcMetallic()` function, and the compiled script treats the `calcMetallic()` call as a `metallicRatio()` call.

As we see here, the Profiler shows performance information for the `calcMetallic()` and `metallicRatio()` calls on lines 21 and 22, but it does **not** show any results for the local code of the `calcMetallic()` function on line 18. Instead, the Profiler's information on line 13 within the `metallicRatio()` function reflects the local code results from **both function calls**:

```

1  // @version=5
2  indicator("Redundant functions demo")
3
4  // @variable Controls the base ratio for the `calcMetallic()` call.
5  int order1Input = input.int(1, "Order 1", 1)
6  // @variable Controls the base ratio for the `metallicRatio()` call.
7  int order2Input = input.int(2, "Order 2", 1)
8
9  // @function      Calculates the value of a metallic ratio with a given `order`, raised to a specified `exponent`.
10 // @param order   Determines the base ratio used. 1 = Golden Ratio, 2 = Silver Ratio, 3 = Bronze Ratio, and so on.
11 // @param exponent The exponent applied to the ratio.
12 metallicRatio(int order, float exponent) =>
13     math.pow((order + math.sqrt(4.0 + order * order)) * 0.5, exponent)
14
15 // @function      A function with the same signature and body as `metallicRatio()` .
16 //                  The script discards this function and treats `calcMetallic()` as an alias for `metallicRatio()` .
17 calcMetallic(int ord, float exp) =>
18     math.pow((ord + math.sqrt(4.0 + ord * ord)) * 0.5, exp)
19
20 // Plot the results from a `calcMetallic()` and `metallicRatio()` call.
21 plot(calcMetallic(order1Input, bar_index % 5), "Ratio 1", color.orange, 3)
22 plot(metallicRatio(order2Input, bar_index % 5), "Ratio 2", color.maroon)
23

```

## A look into the Profiler's inner workings

The Pine Profiler wraps all necessary code regions with specialized *internal functions* to track and collect required information across script executions. It then passes the information to additional calculations that organize and display the performance results inside the Pine Editor. This section gives users a peek into how the Profiler applies internal functions to wrap Pine code and collect performance data.

There are two main internal (**non-Pine**) functions the Profiler wraps significant code with to facilitate runtime analysis. The first function retrieves the current system time at specific points in the script's execution, and the second maps cumulative elapsed time and execution data to specific code regions. We represent these functions in this explanation as `System.currentTimeMillis()` and `registerPerf()` respectively.

When the Profiler detects code that requires analysis, it adds `System.currentTimeMillis()` above the code to get the initial time before execution. Then, it adds `registerPerf()` below the code to map and accumulate the elapsed time and

number of executions. The elapsed time added on each `registerPerf()` call is the `System.timeNow()` value *after* the execution minus the value *before* the execution.

The following *pseudocode* outlines this process for a *single line* of code, where `_startX` represents the starting time for the `lineX` line:

```
long _startX = System.timeNow()
<code_line_to_analyze>
registerPerf(System.timeNow() - _startX, lineX)
```

The process is similar for *code blocks*. The difference is that the `registerPerf()` call maps the data to a *range of lines* rather than a single line. Here, `lineX` represents the *first* line in the code block, and `lineY` represents the block's *last* line:

```
long _startX = System.timeNow()
<code_block_to_analyze>
registerPerf(System.timeNow() - _startX, lineX, lineY)
```

#### Note that:

- In the above snippets, `long`, `System.timeNow()`, and `registerPerf()` represent *internal code*, **not** Pine Script™ code.

Let's now look at how the Profiler wraps a full script and all its significant code. We will start with this script, which calculates three pseudorandom series and displays their *average* result. The script utilizes an *object* of a *user-defined type* to store a pseudorandom state, a *method* to calculate new values and update the state, and an *if...else if* structure to update each series based on generated values:

```
1 //@version=5
2 indicator("Profiler's inner workings demo")
3
4 int seedInput = input.int(12345, "Seed")
5
6 type LCG
7     float state
8
9 method generate(LCG this, int generations = 1) =>
10    float result = 0.0
11    for i = 1 to generations
12        this.state := 16807 * this.state % 2147483647
13        result += this.state / 2147483647
14    result / generations
15
16 var lcg = LCG.new(seedInput)
17
18 var float val0 = 1.0
19 var float val1 = 1.0
20 var float val2 = 1.0
21
22 if lcg.generate(10) < 0.5
23     val0 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1
24 else if lcg.generate(10) < 0.5
25     val1 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1
26 else if lcg.generate(10) < 0.5
27     val2 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1
28
29 plot(math.avg(val0, val1, val2), "Average pseudorandom result", color.purple)
```

The Profiler will wrap the entire script and all necessary code regions, excluding any *insignificant, unused, or redundant*

*code*, with the aforementioned **internal** functions to collect performance data. The *pseudocode* below demonstrates how this process applies to the above script:

```

1 long _startMain = System.timeNow() // Start time for the script's overall execution.
2
3 // <Additional internal code executes here>
4
5 //@version=5
6 indicator("Profiler's inner workings demo") // Declaration statements do not require
7 // profiling.
8
9 int seedInput = input.int(12345, "Seed") // Variable declaration without significant
10 // calculation.
11
12 type LCG           // Type declarations do not require profiling.
13     float state
14
15 method generate(LCG this, int generations = 1) => // Function signature does not
16 // affect runtime.
17     float result = 0.0 // Variable declaration without significant calculation.
18
19     long _start11 = System.timeNow() // Start time for the loop block that begins on
20 // line 11.
21     for i = 1 to generations // Loop header calculations are not independently
22 // wrapped.
23
24         long _start12 = System.timeNow() // Start time for line 12.
25         this.state := 16807 * this.state % 2147483647
26         registerPerf(System.timeNow() - _start12, line12) // Register performance
27 // info for line 12.
28
29         long _start13 = System.timeNow() // Start time for line 13.
30         result += this.state / 2147483647
31         registerPerf(System.timeNow() - _start13, line13) // Register performance
32 // info for line 13.
33
34         registerPerf(System.timeNow() - _start11, line11, line13) // Register performance
35 // info for the block (line 11 - 13).
36
37         long _start14 = System.timeNow() // Start time for line 14.
38         result / generations
39         registerPerf(System.timeNow() - _start14, line14) // Register performance info
40 // for line 14.
41
42         long _start16 = System.timeNow() // Start time for line 16.
43         var lcg = LCG.new(seedInput)
44         registerPerf(System.timeNow() - _start16, line16) // Register performance info for
45 // line 16.
46
47         var float val0 = 1.0 // Variable declarations without significant calculations.
48         var float val1 = 1.0
49         var float val2 = 1.0
50
51         long _start22 = System.timeNow() // Start time for the `if` block that begins on line
52 // 22.
53         if lcg.generate(10) < 0.5 // `if` statement is not independently wrapped.
54
55             long _start23 = System.timeNow() // Start time for line 23.

```

(continues on next page)

(continued from previous page)

```

45     val0 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1
46     registerPerf(System.currentTimeMillis() - _start23, line23) // Register performance info_
→for line 23.

47
48 else if lcg.generate(10) < 0.5 // `else if` statement is not independently wrapped.
49
50     long _start25 = System.currentTimeMillis() // Start time for line 25.
51     val1 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1
52     registerPerf(System.currentTimeMillis() - _start25, line25) // Register performance info_
→for line 25.

53
54 else if lcg.generate(10) < 0.5 // `else if` statement is not independently wrapped.
55
56     long _start27 = System.currentTimeMillis() // Start time for line 27.
57     val2 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1
58     registerPerf(System.currentTimeMillis() - _start27, line27) // Register performance info_
→for line 27.

59
60 registerPerf(System.currentTimeMillis() - _start22, line22, line28) // Register performance_
→info for the block (line 22 - 28).

61
62 long _start29 = System.currentTimeMillis() // Start time for line 29.
63 plot(math.avg(val0, val1, val2), "Average pseudorandom result", color.purple)
64 registerPerf(System.currentTimeMillis() - _start29, line29) // Register performance info for_
→line 29.

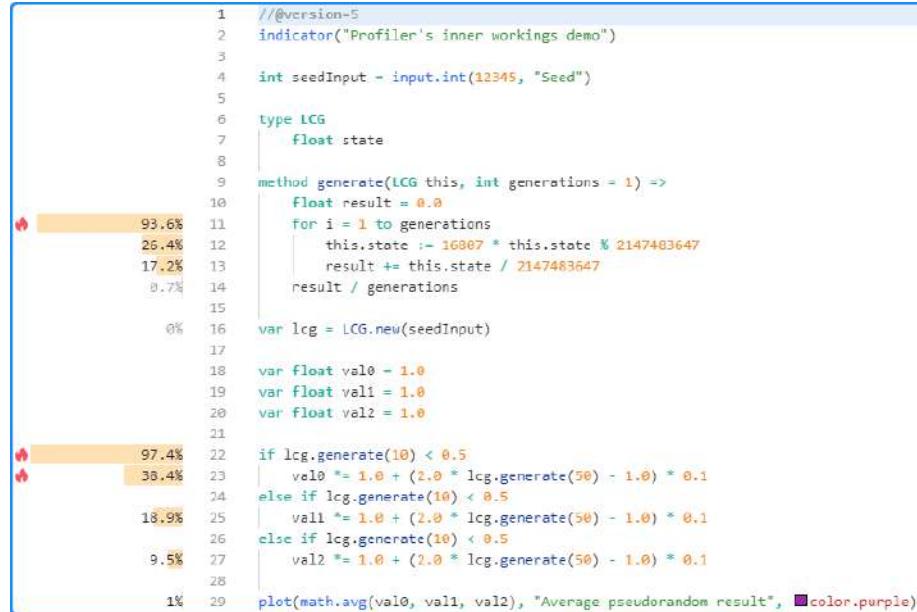
65
66 // <Additional internal code executes here>
67
68 registerPerf(System.currentTimeMillis() - _startMain, total) // Register the script's overall_
→performance info.

```

**Note that:**

- This example is **pseudocode** that provides a basic outline of the **internal calculations** the Profiler applies to collect performance data. Saving this example in the Pine Editor will result in a compilation error since `long`, `System.currentTimeMillis()`, and `registerPerf()` **do not** represent Pine Script™ code.
- These internal calculations that the Profiler wraps a script with require **additional** computational resources, which is why a script's runtime **increases** while profiling. Programmers should always interpret the results as **estimates** since they reflect a script's performance with the extra calculations included.

After running the wrapped script to collect performance data, *additional* internal calculations organize the results and display relevant information inside the Pine Editor:



The “Line time” calculation for `code blocks` also occurs at this stage, as the Profiler cannot individually wrap `loop` headers or the conditional statements in `if` or `switch` structures. This field’s value represents the *difference* between a block’s total time and the sum of its local code times, which is why the “Line time” value for a `switch` block or an `if` block with `else if` expressions represents the time spent on **all** the structure’s conditional statements, not just the block’s *initial line* of code. If a programmer requires more granular information for each conditional expression in such a block, they can reorganize the logic into a *nested if* structure, as explained [here](#).

**Note:** The Profiler **cannot** collect individual performance data for any required *internal* calculations and display their results inside the Pine Editor. Consequently, the time values the Profiler displays for all code regions in a script **will not** add up to 100% of its total runtime.

---

### Profiling across configurations

When a code’s *time complexity* is not constant or its execution pattern varies with its inputs, function arguments, or available data, it’s often wise to profile the code across *different configurations* and data feeds for a more well-rounded perspective on its general performance.

For example, this simple script uses a `for` loop to calculate the sum of squared distances between the current `close` price and `lengthInput` previous prices, then plots the `square root` of that sum on each bar. In this case, the `lengthInput` directly impacts the calculation’s runtime since it determines the number of times the loop executes its local code:

```

1 // @version=5
2 indicator("Profiling across configurations demo")
3
4 // @variable The number of previous bars in the calculation. Directly affects the
5 // number of loop iterations.
6 int lengthInput = input.int(25, "Length", 1)
7
8 // @variable The sum of squared distances from the current `close` to `lengthInput`_
9 // past `close` values.
10 float total = 0.0
11
12 // Look back across `lengthInput` bars and accumulate squared distances.

```

(continues on next page)

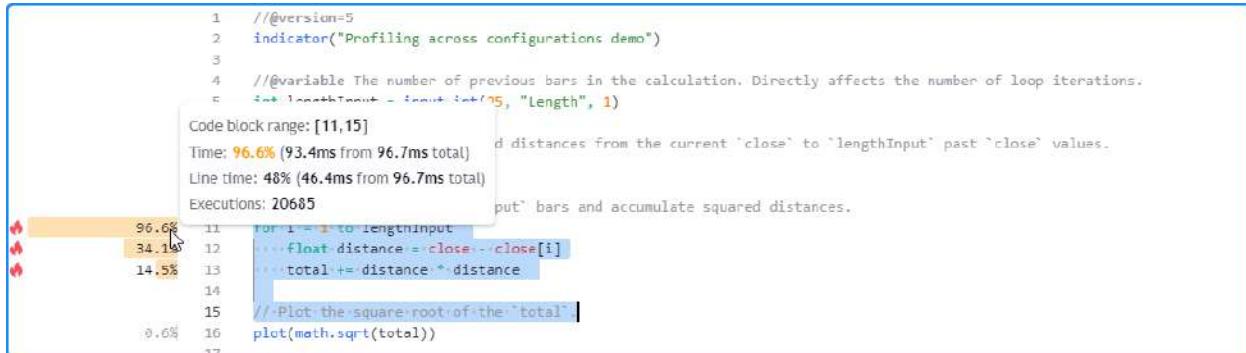
(continued from previous page)

```

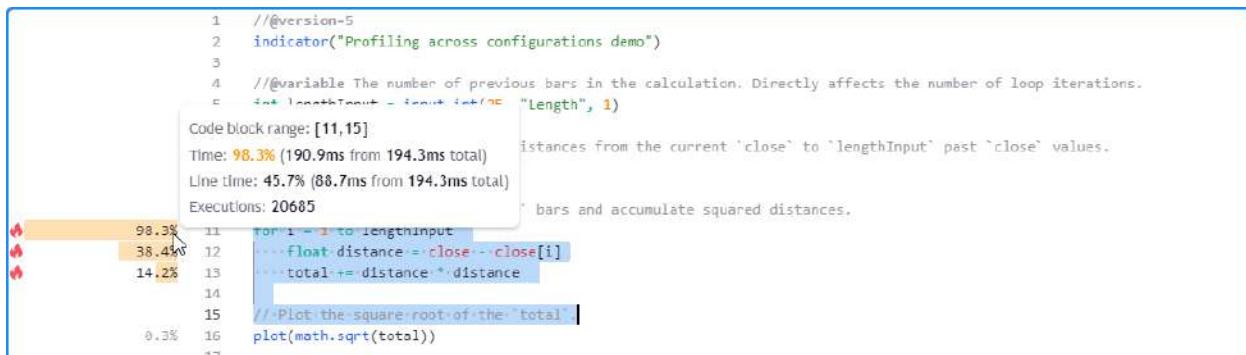
11 for i = 1 to lengthInput
12     float distance = close - close[i]
13     total += distance * distance
14
15 // Plot the square root of the `total`.
16 plot(math.sqrt(total))

```

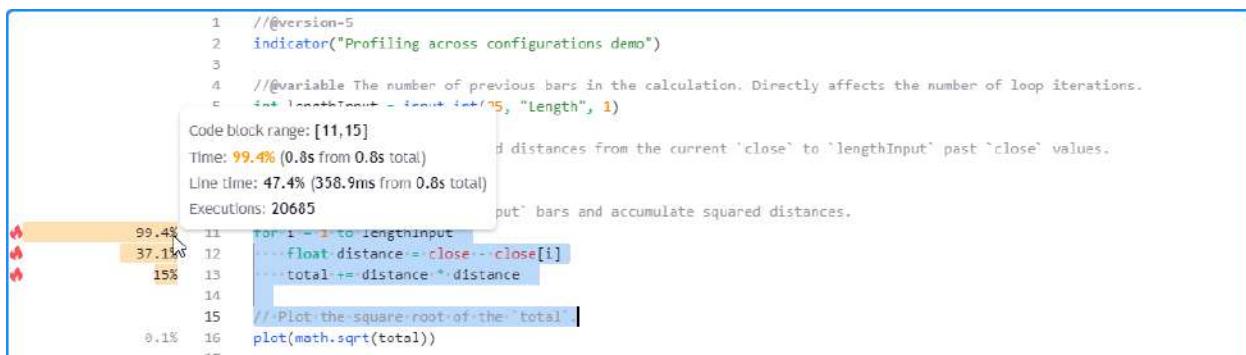
Let's try profiling this script with different `lengthInput` values. First, we'll use the default value of 25. The Profiler's results for this specific run show that the script completed 20,685 executions in about 96.7 milliseconds:



Here, we've increased the input's value to 50 in the script's settings. The results for this run show that the script's total runtime was 194.3 milliseconds, close to *twice* the time from the previous run:



In the next run, we changed the input's value to 200. This time, the Profiler's results show that the script finished all executions in approximately 0.8 seconds, around *four times* the previous run's time:



We can see from these observations that the script's runtime appears to scale *linearly* with the `lengthInput` value, excluding other factors that may affect performance, as one might expect since the bulk of the script's calculations occur within the loop and the input's value controls how many times the loop must execute.

**Note:** It's often wise to profile each configuration *more than once* to reduce the impact of outliers while assessing how a script's performance varies with its inputs or data. See the [section below](#) for more information.

---

### Repetitive profiling

The runtime resources available to a script *vary* over time. Consequently, the time it takes to evaluate a code region, even one with constant [complexity](#), *fluctuates* across executions, and the cumulative performance results shown by the Profiler **will vary** with each independent script run.

Users can enhance their analysis by *restarting* a script several times and profiling each independent run. Averaging the results from each profiled run and evaluating the dispersion of runtime results can help users establish more robust performance benchmarks and reduce the impact of *outliers* (abnormally long or short runtimes) in their conclusions.

Incorporating a *dummy input* (i.e., an input that does nothing) into a script's code is a simple technique that enables users to *restart* it while profiling. The input will not directly affect any calculations or outputs. However, as the user changes its value in the script's settings, the script restarts and the Profiler re-analyzes the executed code.

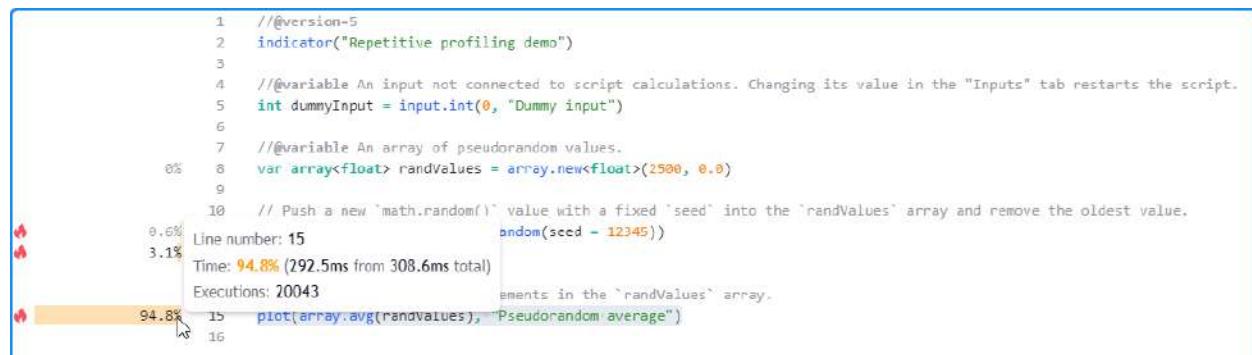
For example, this script *queues* pseudorandom values with a constant seed through an [array](#) with a fixed size, and it calculates and plots the array's [average](#) value on each bar. For profiling purposes, the script includes a `dummyInput` variable with an `input.int()` value assigned to it. The input does nothing in the code aside from allowing us to *restart* the script each time we change its value:

```

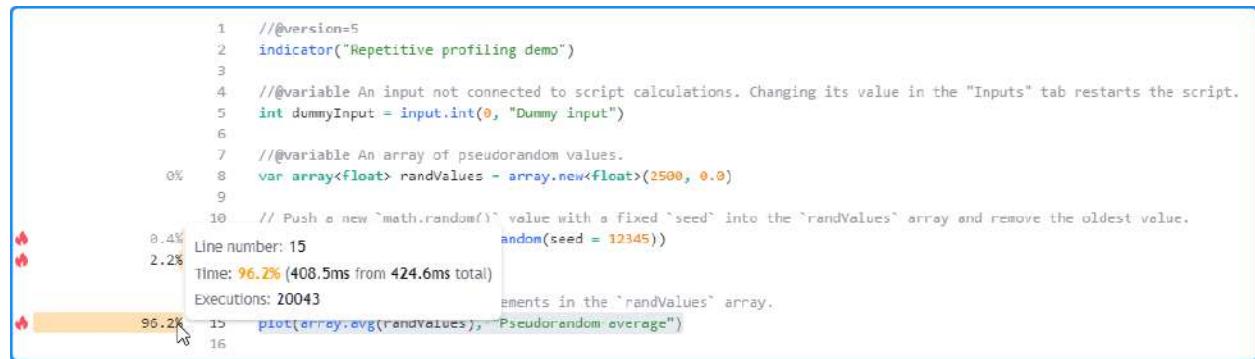
1 // @version=5
2 indicator("Repetitive profiling demo")
3
4 // @variable An input not connected to script calculations. Changing its value in the
5 // "Inputs" tab restarts the script.
6 int dummyInput = input.int(0, "Dummy input")
7
8 // @variable An array of pseudorandom values.
9 var array<float> randValues = array.new<float>(2500, 0.0)
10
11 // Push a new `math.random()` value with a fixed `seed` into the `randValues` array
12 // and remove the oldest value.
13 array.push(randValues, math.random(seed = 12345))
14 array.shift(randValues)
15
16 // Plot the average of all elements in the `randValues` array.
17 plot(array.avg(randValues), "Pseudorandom average")

```

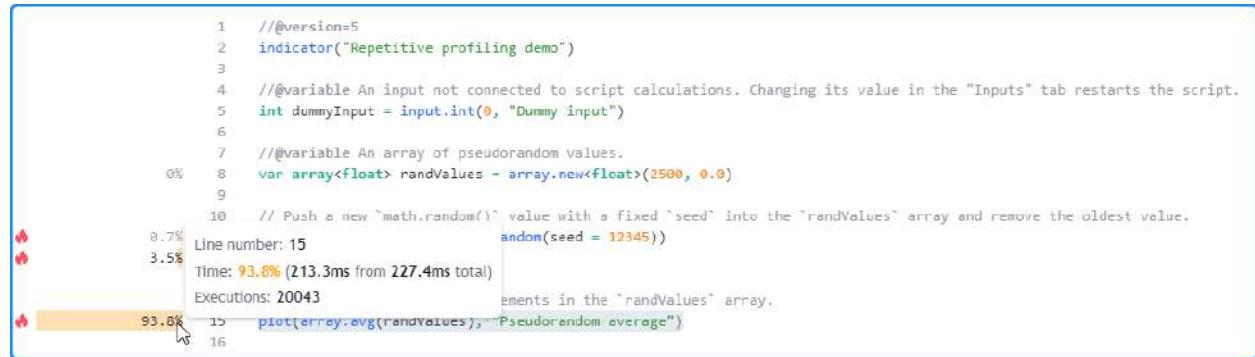
After the first script run, the Profiler shows that it took 308.6 milliseconds to execute across all of the chart's data:



Now, let's change the dummy input's value in the script's settings to restart it without changing the calculations. This time, it completed the same code executions in 424.6 milliseconds, 116 milliseconds longer than the previous run:



Restarting the script again yields another new result. On the third run, the script finished all code executions in 227.4 milliseconds, the fastest result so far:



After repeating this process several times and documenting the results from each run, one can manually calculate their *average* to estimate the script's expected total runtime:

AverageTime = (time1 + time2 + ... + timeN) / N

---

**Note:** Whether profiling a script over a single run or multiple, it's crucial to understand that **results will vary**. While averaging results across several profiled script runs can help derive more stable performance estimates, such estimates are **not** impervious to variance.

---

### 5.3.3 Optimization

*Code optimization*, not to be confused with indicator or strategy optimization, involves modifying a script's source code for improved execution speed, resource efficiency, and scalability. Programmers may use various approaches to optimize a script when they need enhanced runtime performance, depending on what a script's calculations entail.

Fundamentally, most techniques one will use to optimize Pine code involve *reducing* the number of times critical calculations occur or *replacing* significant calculations with simplified formulas or built-ins. Both of these paradigms often overlap.

The following sections explain several straightforward concepts programmers can apply to optimize their Pine Script™ code.

---

**Note:** Before looking for ways to optimize a script, *profile it* to gauge its performance and identify the **critical code**

regions that will benefit the most from optimization.

---

## Using built-ins

Pine Script™ features a variety of *built-in* functions and variables that help streamline script creation. Many of Pine's built-ins feature internal optimizations to help maximize efficiency and achieve fast execution speeds. As such, one of the simplest ways to optimize Pine code is to utilize these efficient built-ins in a script's calculations when possible.

Let's look at an example where one can replace user-defined calculations with a concise built-in call to substantially improve performance. Suppose a programmer wants to calculate the highest value of a series over a specified number of bars. Someone not familiar with all of Pine's built-ins might approach the task using a code like the following, which uses a *loop* on each bar to compare length historical values of a source series:

```
//@variable A user-defined function to calculate the highest `source` value over
// `length` bars.
pineHighest(float source, int length) =>
    float result = na
    if bar_index + 1 >= length
        result := source
    if length > 1
        for i = 1 to length - 1
            result := math.max(result, source[i])
    result
```

Alternatively, one might devise a more optimized Pine function by reducing the number of times the loop executes, as iterating over the history of the source to achieve the result is only necessary when specific conditions occur:

```
//@variable A faster user-defined function to calculate the highest `source` value
// over `length` bars.
// This version only requires a loop when the highest value is removed from
// the window, the `length`
// changes, or when the number of bars first becomes sufficient to calculate
// the result.
fasterPineHighest(float source, int length) =>
    var float result = na
    if source[length] == result or length != length[1] or bar_index + 1 == length
        result := source
    if length > 1
        for i = 1 to length - 1
            result := math.max(result, source[i])
    else
        result := math.max(result, source)
    result
```

The built-in `ta.highest()` function will outperform **both** of these implementations, as its internal calculations are highly optimized for fast computation. Below, we created a script that plots the results of calling `pineHighest()`, `fasterPineHighest()`, and `ta.highest()` to compare their performance using the *Profiler*:

```
1 // @version=5
2 indicator("Using built-ins demo")
3
4 //@variable A user-defined function to calculate the highest `source` value over
// `length` bars.
5 pineHighest(float source, int length) =>
6     float result = na
```

(continues on next page)

(continued from previous page)

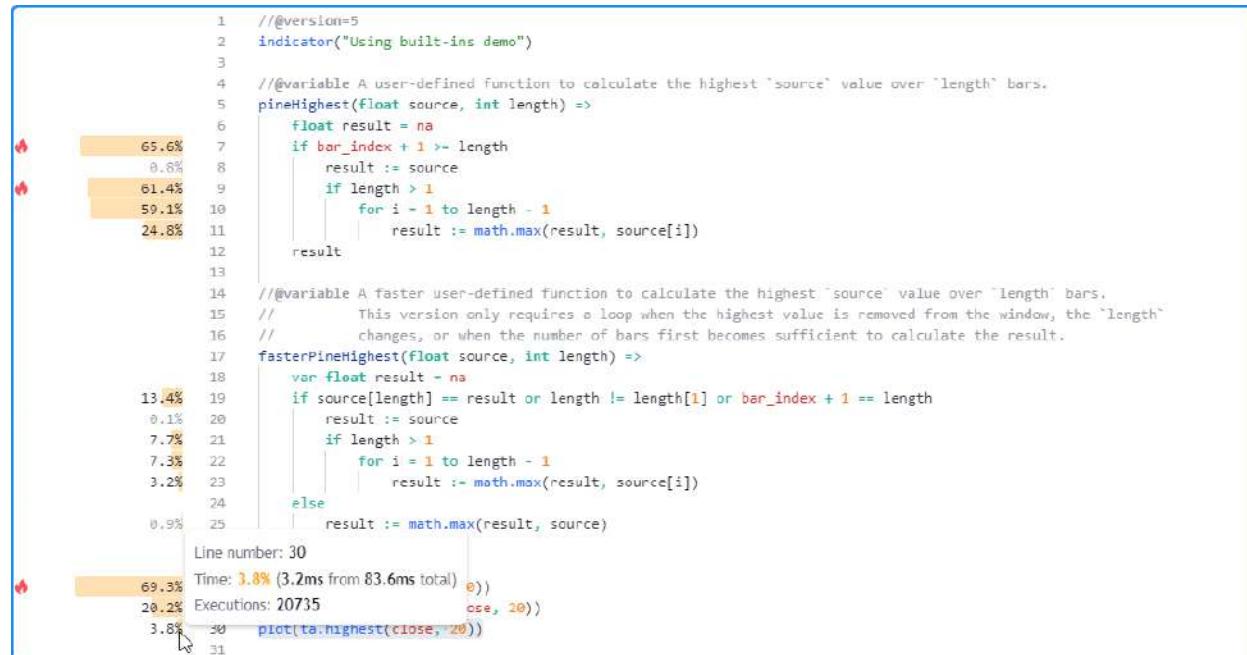
```

7  if bar_index + 1 >= length
8      result := source
9      if length > 1
10         for i = 1 to length - 1
11             result := math.max(result, source[i])
12     result
13
14 // @variable A faster user-defined function to calculate the highest `source` value
15 // over `length` bars.
16 // This version only requires a loop when the highest value is removed from
17 // the window, the `length`
18 // changes, or when the number of bars first becomes sufficient to calculate
19 // the result.
20 fasterPineHighest(float source, int length) =>
21     var float result = na
22     if source[length] == result or length != length[1] or bar_index + 1 == length
23         result := source
24     if length > 1
25         for i = 1 to length - 1
26             result := math.max(result, source[i])
27     else
28         result := math.max(result, source)
29     result
30
31 plot(pineHighest(close, 20))
32 plot(fasterPineHighest(close, 20))
33 plot(ta.highest(close, 20))

```

The *profiled results* over 20,735 script executions show the call to `pineHighest()` was by far the *slowest* to execute, with a runtime of 57.9 milliseconds, about 69.3% of the script's total runtime. The `fasterPineHighest()` call was faster, only taking about 16.9 milliseconds, approximately 20.2% of the total runtime.

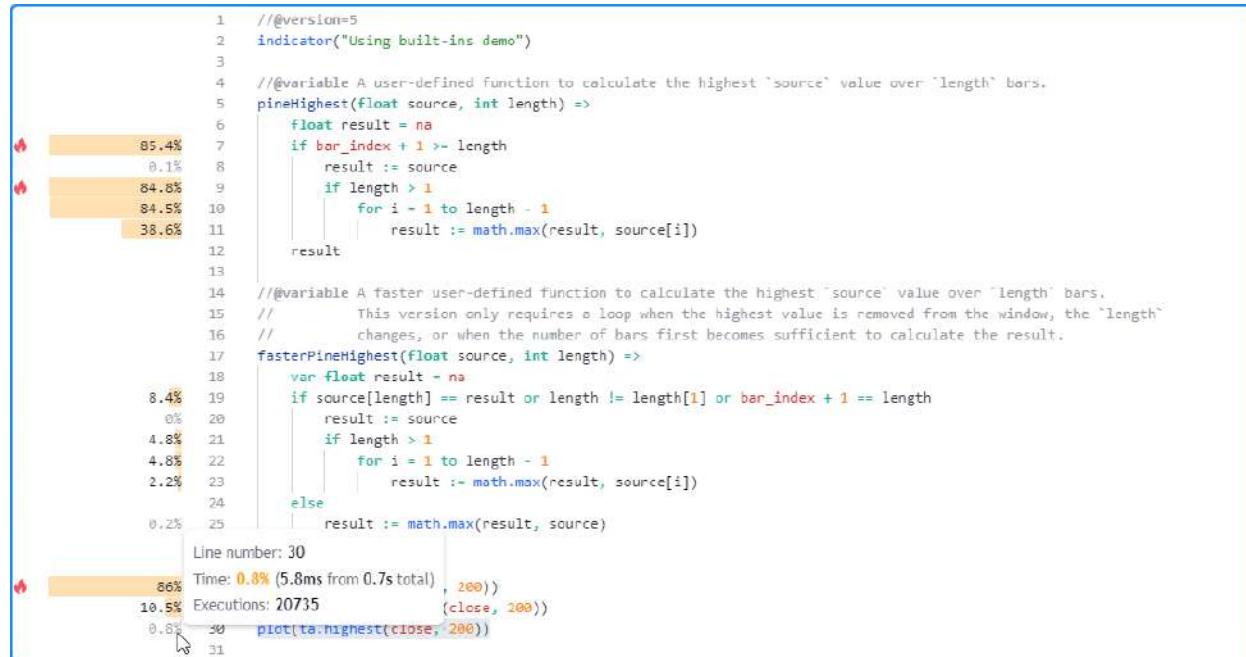
The fastest *by far*, however, was the `ta.highest()` call, which only required 3.2 milliseconds (~3.8% of the total runtime) to execute across all the chart's data and compute the same values in this run:



While these results effectively demonstrate that the built-in function outperforms our *user-defined functions* with a small length argument of 20, it's crucial to consider that the calculations required by the functions *will vary* with the argument's value. Therefore, we can profile the code while using *different arguments* to gauge how its runtime scales.

Here, we changed the `length` argument in each function call from 20 to 200 and *profiled the script* again to observe the changes in performance. The time spent on the `pineHighest()` function in this run increased to about 0.6 seconds (~86% of the total runtime), and the time spent on the `fasterPineHighest()` function increased to about 75 milliseconds. The `ta.highest()` function, on the other hand, *did not* experience a substantial runtime change. It took about 5.8 milliseconds this time, only a couple of milliseconds more than the previous run.

In other words, while our *user-defined functions* experienced significant runtime growth with a higher `length` argument in this run, the change in the built-in `ta.highest()` function's runtime was relatively marginal in this case, thus further emphasizing its performance benefits:



### Note that:

- In many scenarios, a script's runtime can benefit from using built-ins where applicable. However, the relative performance edge achieved from using built-ins depends on a script's *high-impact code* and the specific built-ins used. In any case, one should always *profile their scripts*, preferably *several times*, when exploring optimized solutions.
- The calculations performed by the functions in this example also depend on the sequence of the chart's data. Therefore, programmers can gain further insight into their general performance by profiling the script across *different datasets* as well.

## Reducing repetition

The Pine Script™ compiler can automatically simplify some types of *repetitive code* without a programmer's intervention. However, this automatic process has its limitations. If a script contains repetitive calculations that the compiler *cannot* reduce, programmers can reduce the repetition *manually* to improve their script's performance.

For example, this script contains a `valuesAbove()` *method* that counts the number of elements in an `array` above the element at a specified index. The script plots the number of values above the element at the last index of a `data` array with a calculated `plotColor`. It calculates the `plotColor` within a `switch` structure that calls `valuesAbove()` in all 10 of its conditional expressions:

```

1 // @version=5
2 indicator("Reducing repetition demo")
3
4 // @function Counts the number of elements in `this` array above the element at a
4 //      ↪specified `index`.
5 method int valuesAbove(array<float> this, int index) =>
6     int result = 0
7     float reference = this.get(index)
8     for [i, value] in this
9         if i == index
10            continue
11         if value > reference
12             result += 1
13     result
14
15 // @variable An array containing the most recent 100 `close` prices.
16 var array<float> data = array.new<float>(100)
17 data.push(close)
18 data.shift()
19
20 // @variable Returns `color.purple` with a varying transparency based on the
20 //      ↪`valuesAbove()` .
21 color plotColor = switch
22     data.valuesAbove(99) <= 10    => color.new(color.purple, 90)
23     data.valuesAbove(99) <= 20    => color.new(color.purple, 80)
24     data.valuesAbove(99) <= 30    => color.new(color.purple, 70)
25     data.valuesAbove(99) <= 40    => color.new(color.purple, 60)
26     data.valuesAbove(99) <= 50    => color.new(color.purple, 50)
27     data.valuesAbove(99) <= 60    => color.new(color.purple, 40)
28     data.valuesAbove(99) <= 70    => color.new(color.purple, 30)
29     data.valuesAbove(99) <= 80    => color.new(color.purple, 20)
30     data.valuesAbove(99) <= 90    => color.new(color.purple, 10)
31     data.valuesAbove(99) <= 100   => color.new(color.purple, 0)
32
33 // Plot the number values in the `data` array above the value at its last index.
34 plot(data.valuesAbove(99), color = plotColor, style = plot.style_area)
```

The *profiled results* for this script show that it spent about 2.5 seconds executing 21,201 times. The code regions with the highest impact on the script's runtime are the `for` loop within the `valuesAbove()` local scope starting on line 8 and the `switch` block that starts on line 21:

```

1 //version=5
2 indicator("Reducing repetition demo")
3
4 //@function Counts the number of elements in `this` array above the element at a specified `index`.
5 method valuesAbove(array<float> this, int index) =>
6     int result = 0
7     float reference = this.get(index)
8     for [i, value] in this
9         if i == index
10            continue
11            if value > reference
12                result += 1
13        result
14
15 //variable An array containing the most recent 100 `close` prices.
16 var array<float> data = array.new<float>(100)
17
18 plot(data.valuesAbove(99), color = plotColor, style = plot.style_area)
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

```

Code block range: [21,33]

Time: 84.2% (2.1s from 2.5s total)

Line time: 84.2% (2.1s from 2.5s total)

Executions: 21201

plotColor = switch

data.valuesAbove(99) <= 10 => color.new(color.purple, 90)

data.valuesAbove(99) <= 20 => color.new(color.purple, 80)

data.valuesAbove(99) <= 30 => color.new(color.purple, 70)

data.valuesAbove(99) <= 40 => color.new(color.purple, 60)

data.valuesAbove(99) <= 50 => color.new(color.purple, 50)

data.valuesAbove(99) <= 60 => color.new(color.purple, 40)

data.valuesAbove(99) <= 70 => color.new(color.purple, 30)

data.valuesAbove(99) <= 80 => color.new(color.purple, 20)

data.valuesAbove(99) <= 90 => color.new(color.purple, 10)

data.valuesAbove(99) <= 100 => color.new(color.purple, 0)

//Plot the number of values in the `data` array above the value at its last index.

plot(data.valuesAbove(99), color = plotColor, style = plot.style\_area)

Notice that the number of executions shown for the local code within `valuesAbove()` is substantially *greater* than the number shown for the code in the script's global scope, as the script calls the method up to 11 times per execution, and the results for a *function's local code* reflect the *combined* time and executions from each separate call:

```

1 //version=5
2 indicator("Reducing repetition demo")
3
4 //@function Counts the number of elements in `this` array above the element at a specified `index`.
5 method valuesAbove(array<float> this, int index) =>
6     int result = 0
7     float reference = this.get(index)
8     for [i, value] in this
9         if i == index
10            continue
11            if value > reference
12                result += 1
13        result
14
15 //variable An array containing the most recent 100 `close` prices.
16 var array<float> data = array.new<float>(100)
17
18 plot(data.valuesAbove(99), color = plotColor, style = plot.style_area)
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

```

Code block range: [8,12]

Time: 98.8% (2.5s from 2.5s total)

Line time: 49.9% (1.3s from 2.5s total)

Executions: 130506

Although each `valuesAbove()` call uses the *same* arguments and returns the *same* result, the compiler cannot automatically reduce this code for us during translation. We will need to do the job ourselves. We can optimize this script by assigning the value of `data.valuesAbove(99)` to a *variable* and *reusing* the value in all other areas requiring the result.

In the version below, we modified the script by adding a `count` variable to reference the `data.valuesAbove(99)` value. The script uses this variable in the `plotColor` calculation and the `plot()` call:

```

1 //version=5
2 indicator("Reducing repetition demo")
3
4 //@function Counts the number of elements in `this` array above the element at a
5 //specified `index`.
6 method valuesAbove(array<float> this, int index) =>
7     int result = 0
8     float reference = this.get(index)

```

(continues on next page)

(continued from previous page)

```

8     for [i, value] in this
9         if i == index
10            continue
11        if value > reference
12            result += 1
13    result
14
15 // @variable An array containing the most recent 100 `close` prices.
16 var array<float> data = array.new<float>(100)
17 data.push(close)
18 data.shift()
19
20 // @variable The number values in the `data` array above the value at its last index.
21 int count = data.valuesAbove(99)
22
23 // @variable Returns `color.purple` with a varying transparency based on the ↵
24 color plotColor = switch
25   count <= 10 => color.new(color.purple, 90)
26   count <= 20 => color.new(color.purple, 80)
27   count <= 30 => color.new(color.purple, 70)
28   count <= 40 => color.new(color.purple, 60)
29   count <= 50 => color.new(color.purple, 50)
30   count <= 60 => color.new(color.purple, 40)
31   count <= 70 => color.new(color.purple, 30)
32   count <= 80 => color.new(color.purple, 20)
33   count <= 90 => color.new(color.purple, 10)
34   count <= 100 => color.new(color.purple, 0)
35
36 // Plot the `count`.
37 plot(count, color = plotColor, style = plot.style_area)

```

With this modification, the *profiled results* show a significant improvement in performance, as the script now only needs to evaluate the `valuesAbove()` call **once** per execution rather than up to 11 separate times:

```

1 //@version=5
2 indicator("Reducing repetition demo")
3
4 //@function Counts the number of elements in `this` array above the element at a specified `index`.
5 method valuesAbove(array<float> this, int index) =>
6     int result = 0
7     float reference = this.get(index)
8     for [i, value] in this
9         if i == index
10            continue
11            if value > reference
12                result += 1
13        result
14
15 //@variable An array containing the most recent 100 `close` prices.
16 var array<float> data = array.new<float>(100)
17
18 Line number: 21
19 Time: 96.8% (413.8ms from 427.5ms total)
20 Executions: 21201
21 int count = data.valuesAbove(99)
22
23 //@variable Returns `color.purple` with a varying transparency based on the `valuesAbove()`.
24 color plotColor = switch
25     count <= 10 => □color.new(color.purple, 90)
26     count <= 20 => □color.new(color.purple, 80)
27     count <= 30 => □color.new(color.purple, 70)
28     count <= 40 => □color.new(color.purple, 60)
29     count <= 50 => □color.new(color.purple, 50)
30     count <= 60 => □color.new(color.purple, 40)
31     count <= 70 => □color.new(color.purple, 30)
32     count <= 80 => □color.new(color.purple, 20)
33     count <= 90 => □color.new(color.purple, 10)
34     count <= 100 => □color.new(color.purple, 0)
35
36 // Plot the `count`.
37 plot(count, color = plotColor, style = plot.style_area)

```

### Note that:

- Since this script only calls `valuesAbove()` once, the `method's` local code will now reflect the results from that specific call. See [this section](#) to learn more about interpreting profiled function and method call results.

### Minimizing `request.\*()` calls

The built-in functions in the `request.*()` namespace allow scripts to retrieve data from [other contexts](#). While these functions provide utility in many applications, it's important to consider that each call to these functions can have a significant impact on a script's resource usage.

A single script can contain up to 40 calls to the `request.*()` family of functions. However, users should strive to keep their scripts' `request.*()` calls well *below* this limit to keep the performance impact of their data requests as low as possible.

When a script requests the values of several expressions from the *same* context with multiple `request.security()` or `request.security_lower_tf()` calls, one effective way to optimize such requests is to *condense* them into a single `request.*()` call that uses a *tuple* as its `expression` argument. This optimization not only helps improve the runtime of the requests; it also helps reduce the script's *memory usage* and compiled size.

As a simple example, the following script requests nine `ta.percentrank()` values with different lengths from a specified symbol using nine separate calls to `request.security()`. It then *plots* all nine requested values on the chart to utilize them in the outputs:

```

1 //@version=5
2 indicator("Minimizing `request.*()` calls demo")
3
4 //@variable The symbol to request data from.
5 string symbolInput = input.symbol("BINANCE:BTCUSDT", "Symbol")
6

```

(continues on next page)

(continued from previous page)

```

7 // Request 9 `ta.percentrank()` values from the `symbolInput` context using 9
8 // `request.security()` calls.
9 float reqRank1 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
10 // 10))
11 float reqRank2 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
12 // 20))
13 float reqRank3 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
14 // 30))
15 float reqRank4 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
16 // 40))
17 float reqRank5 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
18 // 50))
19 float reqRank6 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
20 // 60))
21 float reqRank7 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
22 // 70))
23 float reqRank8 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
24 // 80))
25 float reqRank9 = request.security(symbolInput, timeframe.period, ta.percentrank(close,
26 // 90))

27 // Plot the `reqRank*` values.
28 plot(reqRank1)
29 plot(reqRank2)
30 plot(reqRank3)
31 plot(reqRank4)
32 plot(reqRank5)
33 plot(reqRank6)
34 plot(reqRank7)
35 plot(reqRank8)
36 plot(reqRank9)

```

The results from *profiling the script* show that it took the script 340.8 milliseconds to complete its requests and plot the values in this run:

```

1 //@version=5
2 indicator("Minimizing `request.*()` calls demo")
3
4 //variable The symbol to request data from.
5 string symbolInput = input.symbol("BINANCE:BTCUSDT", "Symbol")
6
7 // Request 9 `ta.percentrank()` values from the `symbolInput` context using 9 `request.security()` calls.
8 float reqRank1 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 10))
9 float reqRank2 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 20))
10 float reqRank3 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 30))
11 float reqRank4 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 40))
12 float reqRank5 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 50))
13 float reqRank6 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 60))
14 float reqRank7 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 70))
15 float reqRank8 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 80))
16 float reqRank9 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 90))

17 // Plot the `reqRank*` values.
18 plot(reqRank1)
19 plot(reqRank2)
20 plot(reqRank3)
21 plot(reqRank4)
22 plot(reqRank5)
23 plot(reqRank6)
24 plot(reqRank7)
25 plot(reqRank8)
26 plot(reqRank9)
27

```

Since all the `request.security()` calls request data from the **same context**, we can optimize the code's resource usage by merging all of them into a single `request.security()` call that uses a *tuple* as its expression argument:

```

1 //@version=5
2 indicator("Minimizing `request.*()` calls demo")
3
4 // @variable The symbol to request data from.
5 string symbolInput = input.symbol("BINANCE:BTCUSDT", "Symbol")
6
7 // Request 9 `ta.percentrank()` values from the `symbolInput` context using a single
8 // `request.security()` call.
9 [reqRank1, reqRank2, reqRank3, reqRank4, reqRank5, reqRank6, reqRank7, reqRank8,
10 reqRank9] =
11 request.security(
12     symbolInput, timeframe.period,
13         ta.percentrank(close, 10), ta.percentrank(close, 20), ta.
14     ta.percentrank(close, 30),
15         ta.percentrank(close, 40), ta.percentrank(close, 50), ta.
16     ta.percentrank(close, 60),
17         ta.percentrank(close, 70), ta.percentrank(close, 80), ta.
18     ta.percentrank(close, 90)
19     ]
20 )
21
22 // Plot the `reqRank*` values.
23 plot(reqRank1)
24 plot(reqRank2)
25 plot(reqRank3)
26 plot(reqRank4)
27 plot(reqRank5)
28 plot(reqRank6)
29 plot(reqRank7)
30 plot(reqRank8)
31 plot(reqRank9)

```

As we see below, the *profiled results* from running this version of the script show that it took 228.3 milliseconds this time, a decent improvement over the previous run:



```

1 //@version=5
2 indicator("Minimizing `request.*()` calls demo")
3
4 Line number: 8
5 Time: 50.7% (115.6ms from 228.3ms total)
6 Executions: 29330
7
8 [reqRank1, reqRank2, reqRank3, reqRank4, reqRank5, reqRank6, reqRank7, reqRank8, reqRank9] =
9 request.security(
10     symbolInput, timeframe.period,
11         ta.percentrank(close, 10), ta.percentrank(close, 20), ta.percentrank(close, 30),
12         ta.percentrank(close, 40), ta.percentrank(close, 50), ta.percentrank(close, 60),
13         ta.percentrank(close, 70), ta.percentrank(close, 80), ta.percentrank(close, 90)
14     ]
15 )
16
17 // Plot the `reqRank*` values.
18 plot(reqRank1)
19 plot(reqRank2)
20 plot(reqRank3)
21 plot(reqRank4)
22 plot(reqRank5)
23 plot(reqRank6)
24 plot(reqRank7)
25 plot(reqRank8)
26 plot(reqRank9)

```

### Note that:

- The computational resources available to a script **fluctuate** over time. As such, it's typically a good idea to

profile a script *multiple times* to help solidify performance conclusions.

- Scripts can also request multiple values from the same context with a single `request.*()` call by using an *object* of a *user-defined type (UDT)* as the *expression* argument. See *this section* of the *Other timeframes and data* page to learn more about requesting *UDTs*.
- Programmers can also reduce the total runtime of a `request.security()`, `request.security_lower_tf()`, or `request.seed()` call by passing an argument to the function's `calc_bars_count` parameter, which *restricts* the number of *historical* data points it can access from a context and execute required calculations on. In general, if calls to these `request.*()` functions retrieve *more* historical data than what a script *needs*, limiting the requests with `calc_bars_count` can help improve the script's performance.

## Avoiding redrawing

Pine Script™'s *drawing types* allow scripts to draw custom visuals on a chart that one cannot achieve through other outputs such as *plots*. While these types provide greater visual flexibility, they also have a *higher* runtime and memory cost, especially when a script unnecessarily *recreates* drawings instead of directly updating their properties to change their appearance.

Most *drawing types*, excluding *polylines*, feature built-in *setter functions* in their namespaces that allow scripts to modify a drawing *without* deleting and recreating it. Utilizing these setters is typically less computationally expensive than creating a new drawing object when only *specific properties* require modification.

For example, the script below compares deleting and redrawing *boxes* to using `box.set*()` functions. On the first bar, it declares the `redrawnBoxes` and `updatedBoxes` *arrays* and executes a *loop* to push 25 `box` elements into them.

The script uses a separate `for` loop to iterate across the *arrays* and update the drawings on each execution. It *recreates* the *boxes* in the `redrawnBoxes` array using `box.delete()` and `box.new()`, whereas it *directly modifies* the properties of the *boxes* in the `updatedBoxes` array using `box.set_lefttop()` and `box.set_rightbottom()`. Both approaches achieve the same visual result. However, the latter is more efficient:

```

1 // @version=5
2 indicator("Avoiding redrawing demo")
3
4 // @variable An array of `box` IDs deleted with `box.delete()` and redrawn with `box.
5 // →new()` on each execution.
6 var array<box> redrawnBoxes = array.new<box>()
7 // @variable An array of `box` IDs with properties that update across executions →
8 // →update via `box.set*()` functions.
9 var array<box> updatedBoxes = array.new<box>()
10
11 // Populate both arrays with 25 elements on the first bar.
12 if barstate.isfirst
13     for i = 1 to 25
14         array.push(redrawnBoxes, box(na))
15         array.push(updatedBoxes, box.new(na, na, na, na))
16
17 for i = 0 to 24
18     // Calculate coordinates.
19     int x = bar_index - i
20     float y = close[i + 1] - close
21     // Get the `box` ID from each array at the `i` index.
22     box redrawnBox = redrawnBoxes.get(i)
23     box updatedBox = updatedBoxes.get(i)
24     // Delete the `redrawnBox`, create a new `box` ID, and replace that element in →
25     // the `redrawnboxes` array.
26     box.delete(redrawnBox)

```

(continues on next page)

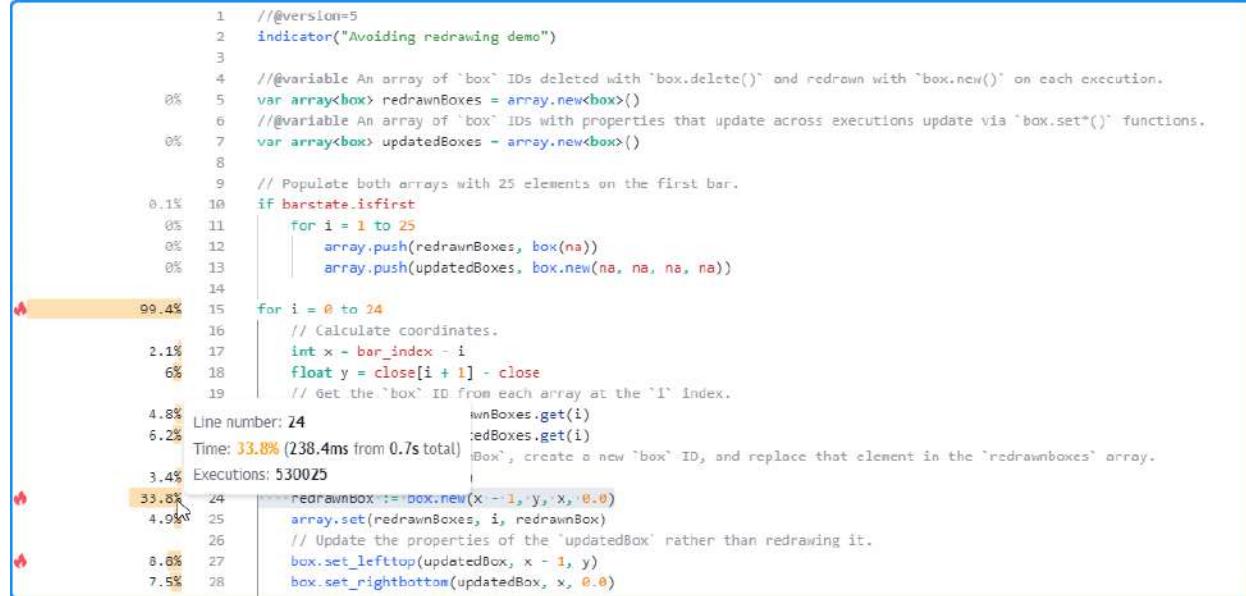
(continued from previous page)

```

24    redrawnBox := box.new(x - 1, y, x, 0.0)
25    array.set(redrawnBoxes, i, redrawnBox)
26    // Update the properties of the `updatedBox` rather than redrawing it.
27    box.set_lefttop(updatedBox, x - 1, y)
28    box.set_rightbottom(updatedBox, x, 0.0)

```

The results from *profiling this script* show that line 24, which contains the `box.new()` call, is the *heaviest* line in the *code block* that executes on each bar, with a runtime close to **double** the combined time spent on the `box.set_lefttop()` and `box.set_rightbottom()` calls on lines 27 and 28:



### Note that:

- The number of executions shown for the loop's *local code* is 25 times the number shown for the code in the script's *global scope*, as each execution of the loop statement triggers 25 executions of the local block.
- This script updates its drawings over *all bars* in the chart's history for **testing** purposes. However, it does **not** actually need to execute all these historical updates since users will only see the **final** result from the *last historical bar* and the changes across *realtime bars*. See the *next section* to learn more.

## Reducing drawing updates

When a script produces *drawing objects* that change across *historical bars*, users will only ever see their **final results** on those bars since the script completes its historical executions when it first loads on the chart. The only time one will see such drawings *evolve* across executions is during *realtime bars*, as new data flows in.

Since the evolving outputs from dynamic *drawings* on historical bars are **never visible** to a user, one can often improve a script's performance by *eliminating* the historical updates that don't impact the final results.

For example, this script creates a `table` with two columns and 21 rows to visualize the history of an `RSI` in a paginated, tabular format. The script initializes the cells of the `infoTable` on the *first bar*, and it references the history of the calculated `rsi` to update the `text` and `bgcolor` of the cells in the second column within a `for` loop on each bar:

```

1 // @version=5
2 indicator("Reducing drawing updates demo")
3

```

(continues on next page)

(continued from previous page)

```

4 // @variable The first offset shown in the paginated table.
5 int offsetInput = input.int(0, "Page", 0, 249) * 20
6
7 // @variable A table that shows the history of RSI values.
8 var table infoTable = table.new(position.top_right, 2, 21, border_color = chart.fg_
9   ↪color, border_width = 1)
10 // Initialize the table's cells on the first bar.
11 if barstate.isfirst
12   table.cell(infoTable, 0, 0, "Offset", text_color = chart.fg_color)
13   table.cell(infoTable, 1, 0, "RSI", text_color = chart.fg_color)
14   for i = 0 to 19
15     table.cell(infoTable, 0, i + 1, str.tostring(offsetInput + i))
16     table.cell(infoTable, 1, i + 1)
17
18 float rsi = ta.rsi(close, 14)
19
20 // Update the history shown in the `infoTable` on each bar.
21 for i = 0 to 19
22   float historicalRSI = rsi[offsetInput + i]
23   table.cell_set_text(infoTable, 1, i + 1, str.tostring(historicalRSI))
24   table.cell_set bgcolor(
25     infoTable, 1, i + 1, color.from_gradient(historicalRSI, 30, 70, color.red, ↪
26   ↪color.green)
27 )
28
29 plot(rsi, "RSI")

```

After *profiling* the script, we see that the code with the highest impact on performance is the `for` loop that starts on line 20, i.e., the *code block* that updates the table's cells:



This critical code region executes **excessively** across the chart's history, as users will only see the *table's final* historical result. The only time that users will see the *table* update is on the **last historical bar** and across all subsequent **realtime bars**. Therefore, we can optimize this script's resource usage by restricting the executions of this code to only the **last available bar**.

In this script version, we placed the *loop* that updates the *table* cells within an *if* structure that uses `barstate.islast` as its condition, effectively restricting the code block's executions to only the last historical bar and all realtime bars. Now, the

script *loads* more efficiently since all the table's calculations only require **one** historical execution:

```

1  //@version=5
2  indicator("Reducing drawing updates demo")
3
4  //@variable The first offset shown in the paginated table.
5  int offsetInput = input.int(0, "Page", 0, 249) * 20
6
7  //@variable A table that shows the history of RSI values.
8  var table infoTable = table.new(position.top_right, 2, 21, border_color = chart.fg_color, border_width = 1)
9  // Initialize the table's cells on the first bar.
10 if barstate.isfirst
11     table.cell(infoTable, 0, 0, "Offset", text_color = chart.fg_color)
12     table.cell(infoTable, 1, 0, "RSI", text_color = chart.fg_color)
13     for i = 0 to 19
14         table.cell(infoTable, 0, i + 1, str.tostring(offsetInput + i))
15         table.cell(infoTable, 1, i + 1)
16
17 Code block range: [21,27]
18 Time: 0.9% (209mcs from 24.2ms total)
19 Line time: 0.2% (50mcs from 24.2ms total), in the `infoTable` on the last available bar.
20 Executions: 1
21
22 for i = 0 to 19
23     float historicalRSI = rsi[offsetInput + i]
24     table.cell_set_text(infoTable, 1, i + 1, str.tostring(historicalRSI))
25     table.cell_set bgcolor(
26         infoTable, 1, i + 1, color.from_gradient(historicalRSI, 30, 70, color.red, color.green)
27     )
28
29 plot(rsi, "RSI")
30

```

```

1  //@version=5
2  indicator("Reducing drawing updates demo")
3
4  //@variable The first offset shown in the paginated table.
5  int offsetInput = input.int(0, "Page", 0, 249) * 20
6
7  //@variable A table that shows the history of RSI values.
8  var table infoTable = table.new(position.top_right, 2, 21, border_color = chart.fg_
9      color, border_width = 1)
10 // Initialize the table's cells on the first bar.
11 if barstate.isfirst
12     table.cell(infoTable, 0, 0, "Offset", text_color = chart.fg_color)
13     table.cell(infoTable, 1, 0, "RSI", text_color = chart.fg_color)
14     for i = 0 to 19
15         table.cell(infoTable, 0, i + 1, str.tostring(offsetInput + i))
16         table.cell(infoTable, 1, i + 1)
17
18 float rsi = ta.rsi(close, 14)
19
20 // Update the history shown in the `infoTable` on the last available bar.
21 if barstate.islast
22     for i = 0 to 19
23         float historicalRSI = rsi[offsetInput + i]
24         table.cell_set_text(infoTable, 1, i + 1, str.tostring(historicalRSI))
25         table.cell_set bgcolor(
26             infoTable, 1, i + 1, color.from_gradient(historicalRSI, 30, 70, color.
27                 red, color.green)
28         )
29
30 plot(rsi, "RSI")

```

### Note that:

- The script will still update the cells when new **realtime** updates come in, as users can observe those changes on the chart, unlike the changes that the script used to execute across historical bars.

## Storing calculated values

When a script performs a critical calculation that changes *infrequently* throughout all executions, one can reduce its runtime by **saving the result** to a variable declared with the `var` or `varip` keywords and **only** updating the value if the calculation changes. If the script calculates *multiple* values excessively, one can store them within *collections* (*arrays*, *matrices*, and *maps*) or *objects* of *user-defined types*.

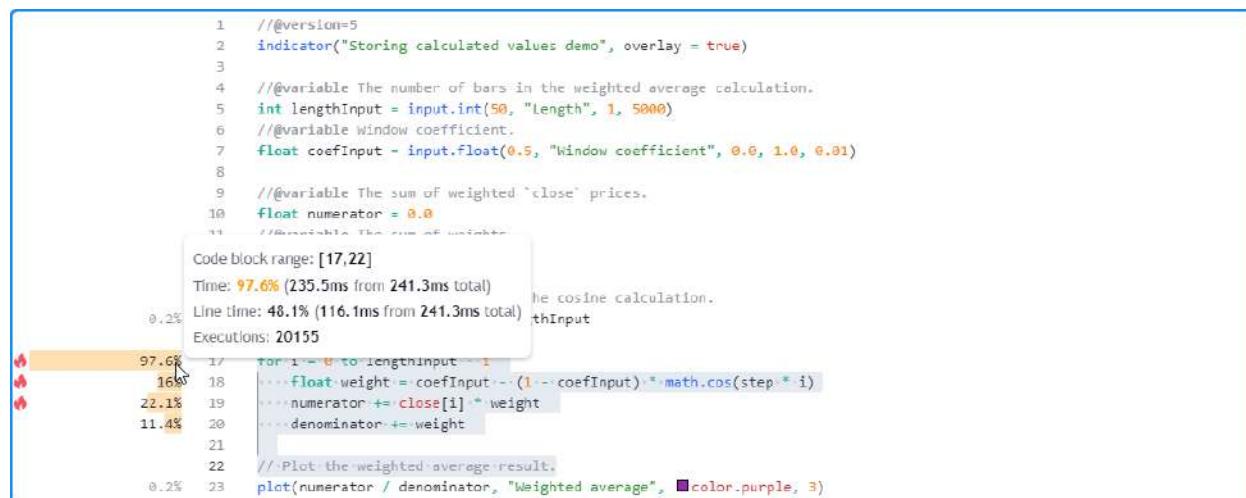
Let's look at an example. This script calculates a weighted moving average with custom weights based on a generalized **window function**. The numerator is the sum of weighted `close` values, and the denominator is the sum of the calculated weights. The script uses a `for` loop that iterates `lengthInput` times to calculate these sums, then it plots their ratio, i.e., the resulting average:

```

1 // @version=5
2 indicator("Storing calculated values demo", overlay = true)
3
4 //@variable The number of bars in the weighted average calculation.
5 int lengthInput = input.int(50, "Length", 1, 5000)
6 //@variable Window coefficient.
7 float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01)
8
9 //@variable The sum of weighted `close` prices.
10 float numerator = 0.0
11 //@variable The sum of weights.
12 float denominator = 0.0
13
14 //@variable The angular step in the cosine calculation.
15 float step = 2.0 * math.pi / lengthInput
16 // Accumulate weighted sums.
17 for i = 0 to lengthInput - 1
18     float weight = coefInput - (1 - coefInput) * math.cos(step * i)
19     numerator += close[i] * weight
20     denominator += weight
21
22 // Plot the weighted average result.
23 plot(numerator / denominator, "Weighted average", color.purple, 3)

```

After *profiling* the script's performance over our chart's data, we see that it took about 241.3 milliseconds to calculate the default 50-bar average across 20,155 chart updates, and the critical code with the *highest impact* on the script's performance is the *loop block* that starts on line 17:



Since the number of loop iterations *depends* on the `lengthInput` value, let's test how its runtime scales with *another*

*configuration* requiring heavier looping. Here, we set the value to 2500. This time, the script took about 12 seconds to complete all its executions:

```

1 //@version=5
2 indicator("Storing calculated values demo", overlay = true)
3
4 //@variable The number of bars in the weighted average calculation.
5 int lengthInput = input.int(50, "Length", 1, 5000)
6 //@variable Window coefficient.
7 float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01)
8
9 //@variable The sum of weighted `close` prices.
10 float numerator = 0.0
11 //@variable The sum of weights.
12
13
14
15
16
17
18
19
20
21
22
23

```

Code block range: [17,22]  
Time: 99.9% (12s from 12s total)  
Line time: 46.7% (5.6s from 12s total)  
Executions: 20155

99.9%	17	for i = 0 to lengthInput - 1
16.9%	18	// float weight = coefInput - (1 - coefInput) * math.cos(step * i)
24.6%	19	// numerator += close[i] * weight
11.6%	20	// denominator += weight
0%	21	
0%	22	// Plot the weighted average result.
0%	23	plot(numerator / denominator, "Weighted average", color.purple, 3)

Now that we've pinpointed the script's *high-impact* code and established a benchmark to improve, we can inspect the critical code block to identify optimization opportunities. After examining the calculations, we can observe the following:

- The only value that causes the weight calculation on line 18 to vary across loop iterations is the *loop index*. All other values in its calculation remain consistent. Consequently, the weight calculated on each loop iteration **does not vary** across chart bars. Therefore, rather than calculating the weights **on every update**, we can calculate them **once**, on the first bar, and **store them** in a *collection* for future access across subsequent script executions.
- Since the weights never change, the resulting denominator never changes. Therefore, we can add the `var` keyword to the *variable declaration* and only calculate its value **once** to reduce the number of executed addition *assignment* operations.
- Unlike the denominator, we **cannot** store the numerator value to simplify its calculation since it consistently *changes over time*.

In the modified script below, we've added a `weights` variable to reference an `array` that stores each calculated `weight`. This variable and the `denominator` both include the `var` keyword in their declarations, meaning the values assigned to them will *persist* throughout all script executions until explicitly reassigned. The script calculates their values using a `for` loop that only executes on the *first chart bar*. Across all other bars, it calculates the `numerator` using a `for...in` loop that references the *saved values* from the `weights` array:

```

1 //@version=5
2 indicator("Storing calculated values demo", overlay = true)
3
4 //@variable The number of bars in the weighted average calculation.
5 int lengthInput = input.int(50, "Length", 1, 5000)
6 //@variable Window coefficient.
7 float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01)
8
9 //@variable An array that stores the `weight` values calculated on the first chart
10 //bar.
11 var array<float> weights = array.new<float>()
12
13 //@variable The sum of weighted `close` prices.
14 float numerator = 0.0
15 //@variable The sum of weights. The script now only calculates this value on the
16 //first bar.
17 var float denominator = 0.0

```

(continues on next page)

(continued from previous page)

```

16 // @variable The angular step in the cosine calculation.
17 float step = 2.0 * math.pi / lengthInput
18
19 // Populate the `weights` array and calculate the `denominator` only on the first bar.
20 if barstate.isfirst
21     for i = 0 to lengthInput - 1
22         float weight = coefInput - (1 - coefInput) * math.cos(step * i)
23         array.push(weights, weight)
24         denominator += weight
25
26 // Calculate the `numerator` on each bar using the stored `weights`.
27 for [i, w] in weights
28     numerator += close[i] * w
29
30 // Plot the weighted average result.
31 plot(numerator / denominator, "Weighted average", color.purple, 3)

```

With this optimized structure, the *profiled results* show that our modified script with a high `lengthInput` value of 2500 took about 5.9 seconds to calculate across the same data, about *half* the time of our previous version:

```

1 // @version=5
2 indicator("Storing calculated values demo", overlay = true)
3
4 // @variable The number of bars in the weighted average calculation.
5 int lengthInput = input.int(50, "Length", 1, 5000)
6 // @variable Window coefficient.
7 float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01)
8
9 // @variable An array that stores the "weight" values calculated on the first chart bar.
10 var array<float> weights = array.new<float>()
11
12 // @variable The sum of weighted "close" prices.
13 float numerator = 0.0
14 // @variable The sum of weights. The script now only calculates this value on the first bar.
15 var float denominator = 0.0
16
17 // @variable The angular step in the cosine calculation.
18 float step = 2.0 * math.pi / lengthInput
19
20 // Populate the `weights` array and calculate the `denominator` only on the first bar.
21 if barstate.isfirst
22     for i = 1 to lengthInput - 1
23         float weight = coefInput - (1 - coefInput) * math.cos(step * i)
24         array.push(weights, weight)
25         denominator += weight
26
27 for [i, w] in weights
28     numerator += close[i] * w
29
30 // Plot the weighted average result.
31 plot(numerator / denominator, "Weighted average", color.purple, 3)

```

### Note that:

- Although we've significantly improved this script's performance by saving its *execution-invariant* values to variables, it does still involve a higher computational cost with **large** `lengthInput` values due to the remaining loop calculations that execute on each bar.
- Another, more *advanced* way one can further speed up this script is by storing the weights in a *single-row matrix* on the first bar, using an `array` as a *queue* to hold recent `close` values, then replacing the `for...in` loop with a call to `matrix.mult()`. See the *Matrices* page to learn more about working with `matrix.*()` functions.

## Eliminating loops

*Loops* allow Pine scripts to perform *iterative* calculations on each execution. Each time a loop activates, its local code may execute *several times*, often leading to a *substantial increase* in resource usage.

Pine loops are necessary for *some* calculations, such as manipulating elements within *collections* or looking backward through a dataset's history to calculate values *only* obtainable on the current bar. However, in many other cases, programmers use loops when they **don't need to**, leading to suboptimal runtime performance. In such cases, one may eliminate unnecessary loops in any of the following ways, depending on what their calculations entail:

- Identifying simplified, **loop-free expressions** that achieve the same result without iteration
- Replacing a loop with optimized *built-ins* where possible
- Distributing a loop's iterations *across bars* when feasible rather than evaluating them all at once

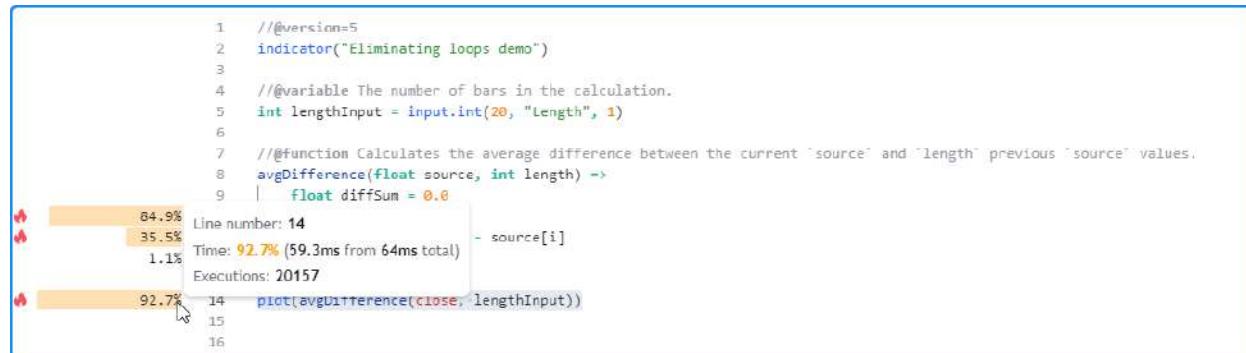
This simple example contains an `avgDifference()` function that calculates the average difference between the current bar's `source` value and all the values from `length` previous bars. The script calls this function to calculate the average difference between the current `close` price and `lengthInput` previous prices, then it *plots* the result on the chart:

```

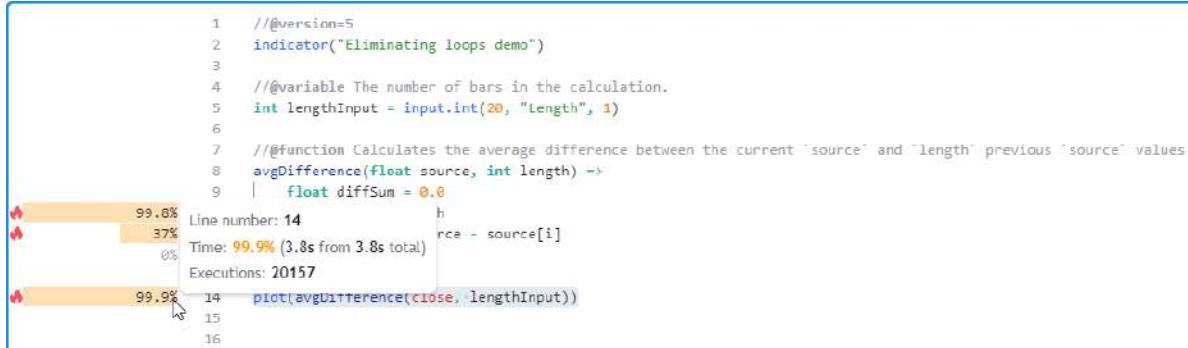
1 // @version=5
2 indicator("Eliminating loops demo")
3
4 //@variable The number of bars in the calculation.
5 int lengthInput = input.int(20, "Length", 1)
6
7 //@function Calculates the average difference between the current `source` and_
8 // `length` previous `source` values.
9 avgDifference(float source, int length) =>
10    float diffSum = 0.0
11    for i = 1 to length
12        diffSum += source - source[i]
13    diffSum / length
14
15 plot(avgDifference(close, lengthInput))

```

After inspecting the script's *profiled results* with the default settings, we see that it took about 64 milliseconds to execute 20,157 times:



Since we use the `lengthInput` as the `length` argument in the `avgDifference()` call and that argument controls how many times the loop inside the function must iterate, our script's runtime will **grow** with the `lengthInput` value. Here, we set the input's value to 2000 in the script's settings. This time, the script completed its executions in about 3.8 seconds:



As we see from these results, the `avgDifference()` function can be costly to call, depending on the specified `lengthInput` value, due to its `for` loop that executes on each bar. However, *loops* are **not** necessary to achieve the output. To understand why, let's take a closer look at the loop's calculations. We can represent them with the following expression:

```
(source - source[1]) + (source - source[2]) + ... + (source - source[length])
```

Notice that it adds the *current* `source` value `length` times. These iterative additions are not necessary. We can simplify that part of the expression to `source * length`, which reduces it to the following:

```
source * length - source[1] - source[2] - ... - source[length]
```

or equivalently:

```
source * length - (source[1] + source[2] + ... + source[length])
```

After simplifying and rearranging this representation of the loop's calculations, we see that we can compute the result in a simpler way and **eliminate** the loop by subtracting the previous bar's `rolling sum` of `source` values from the `source * length` value, i.e.:

```
source * length - math.sum(source, length)[1]
```

The `fastAvgDifference()` function below is a **loop-free** alternative to the original `avgDifference()` function that uses the above expression to calculate the sum of `source` differences, then divides the expression by the `length` to return the average difference:

```
//@function A faster way to calculate the `avgDifference()` result.
//           Eliminates the `for` loop using the relationship:
//           `(x - x[1]) + (x - x[2]) + ... + (x - x[n]) = x * n - math.sum(x, n)[1]`.
fastAvgDifference(float source, int length) =>
    (source * length - math.sum(source, length)[1]) / length
```

Now that we've identified a potential optimized solution, we can compare the performance of `fastAvgDifference()` to the original `avgDifference()` function. The script below is a modified form of the previous version that plots the results from calling both functions with the `lengthInput` as the `length` argument:

```
1  //@version=5
2  indicator("Eliminating loops demo")
3
4  //@variable The number of bars in the calculation.
5  int lengthInput = input.int(20, "Length", 1)
6
7  //@function Calculates the average difference between the current `source` and
   ↴`length` previous `source` values.
```

(continues on next page)

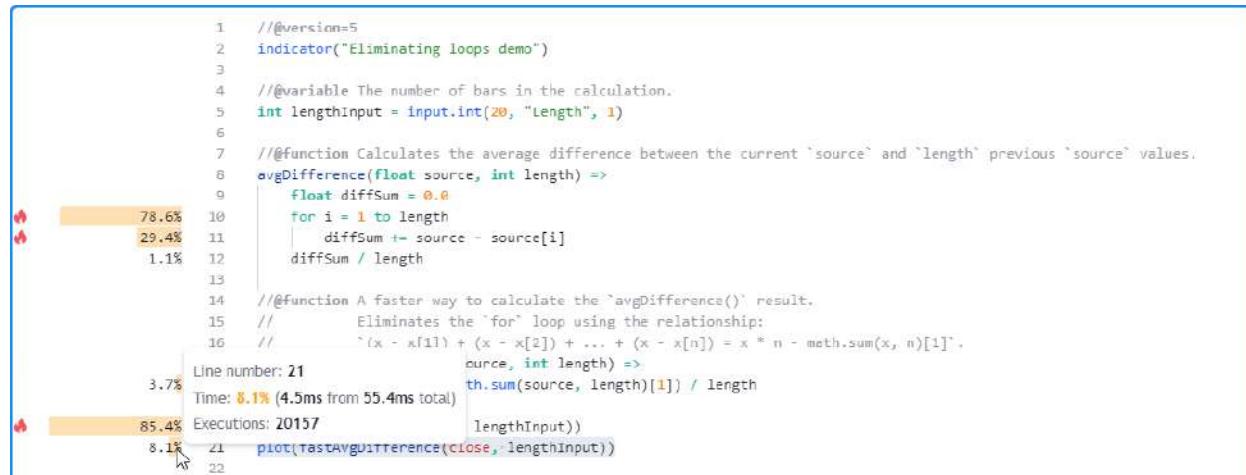
(continued from previous page)

```

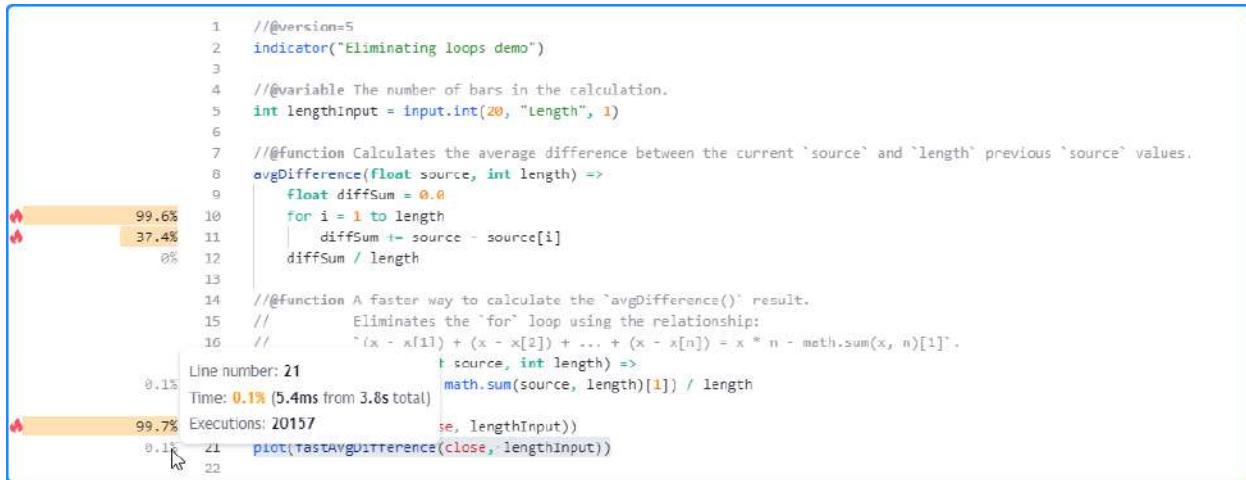
8 avgDifference(float source, int length) =>
9     float diffSum = 0.0
10    for i = 1 to length
11        diffSum += source - source[i]
12    diffSum / length
13
14 // @function A faster way to calculate the `avgDifference()` result.
15 //     Eliminates the `for` loop using the relationship:
16 //     `(x - x[1]) + (x - x[2]) + ... + (x - x[n]) = x * n - math.sum(x, n)[1]`.
17 fastAvgDifference(float source, int length) =>
18     (source * length - math.sum(source, length)[1]) / length
19
20 plot(avgDifference(close, lengthInput))
21 plot(fastAvgDifference(close, lengthInput))

```

The *profiled results* for the script with the default `lengthInput` of 20 show a substantial difference in runtime spent on the two function calls. The call to the original function took about 47.3 milliseconds to execute 20,157 times on this run, whereas our optimized function only took 4.5 milliseconds:



Now, let's compare the performance with the *heavier* `lengthInput` value of 2000. As before, the runtime spent on the `avgDifference()` function increased significantly. However, the time spent executing the `fastAvgDifference()` call remained very close to the result from the previous *configuration*. In other words, while our original function's runtime scales directly with its `length` argument, our optimized function demonstrates relatively *consistent* performance since it does not require a loop:



**Note:** Not all iterative calculations will necessarily have loop-free alternatives. In the case where a script can **only** achieve its results through iteration, programmers can identify possible ways to optimize loops to improve performance. See the [next section](#) for more information.

## Optimizing loops

Although Pine's *execution model* and the available built-ins often *eliminate* the need for *loops* in many cases, there are still instances where a script **will** require *loops* for some types of tasks, including:

- Manipulating *collections* or executing calculations over a collection's elements when the available built-ins **will not suffice**
- Performing calculations across historical bars that one **cannot** achieve with simplified *loop-free* expressions or optimized *built-ins*
- Calculating values that are **only** obtainable through iteration

When a script uses *loops* that a programmer cannot *eliminate*, there are **several techniques** one can use to reduce their performance impact. This section explains two of the most common, useful techniques that can help improve a required loop's efficiency.

**Note:** Before identifying ways to *optimize* a loop, we recommend searching for ways to *eliminate* it first. If **no solution** exists that makes the loop unnecessary, then proceed with attempting to reduce its overhead.

## Reducing loop calculations

The code executed within a *loop's* local scope can have a **multiplicative** impact on its overall runtime, as each time a loop statement executes, it will typically trigger *several* iterations of the local code. Therefore, programmers should strive to keep a loop's calculations as simple as possible by eliminating unnecessary structures, function calls, and operations to minimize the performance impact, especially when the script must evaluate its loops *numerous times* throughout all its executions.

For example, this script contains a `filter()` function that filters elements out of an `array` based on the `true` elements in a mask `array` of the same size. The function uses a `for...in` loop to iterate through both `arrays` and construct a new filtered array. The script uses this function to filter out random elements of a `prices` array and plots the `array.avg()` of the result on the chart:

```
1 // @version=5
2 indicator("Reducing loop calculations demo", overlay = true)
3
4 // @function Creates a filtered version of an array.
5 // @param this The array of "bool" values to filter.
6 // @param mask An array of "bool" values used to filter `this` array.
7 filter(array<float> this, array<bool> mask) =>
8     array<float> result = array.new<float>()
9     if this.size() != mask.size()
10        runtime.error("Cannot call `filter()` with two arrays of different sizes.")
11    for item in this
12        int index = array.indexof(this, item)
13        if array.get(mask, index)
14            result.push(item)
15    result
16
17 // @variable An array containing the most recent 100 `close` prices.
18 var array<float> prices = array.new<float>(100, close)
19 // @variable An array containing 100 pseudorandom `bool` values to filter the `price` ↴
20 // array.
21 var array<bool> randMask = array.new<bool>(100, true)
22
23 // Push the first element from `randMask` to the end and queue a new pseudorandom ↴
24 // value.
25 randMask.push(randMask.shift())
26 randMask.push(math.random(seed = 12345) < 0.5)
27 randMask.shift()
28 // Queue the `close` value into the `prices` array.
29 prices.push(close)
30 prices.shift()
31
32 // Plot the average of the elements in the filtered `prices` array.
33 plot(array.avg(filter(prices, randMask)))
```

After [profiling the script](#), we see it took 1.7 seconds to execute 20,207 times. The code with the highest performance impact is the expression on line 31, which calls the `filter()` function and `array.avg()`. Within the `filter()` function's scope, the `for...in` loop has the highest impact, with the first line in the loop's scope (line 12) contributing the most to the loop's time:

```

1  //@version=5
2  indicator("Reducing loop calculations demo", overlay = true)
3
4  //@function    Creates a filtered version of an array.
5  //@param this   The array of "bool" values to filter.
6  //@param mask   An array of "bool" values used to filter `this` array.
7  filter(array<float> this, array<bool> mask) ->
8      array<float> result = array.new<float>()
9      if this.size() != mask.size()
10         runtime.error("Cannot call `filter()` with two arrays of different sizes.")
11     for item in this
12         int index = array.indexof(this, item)
13         if array.get(mask, index)
14             result.push(item)
15     result
16
17 // @variable An array containing the most recent 100 `close` prices.
18 var array<float> prices = array.new<float>(100, close)
19 // @variable An array containing 100 pseudorandom `bool` values to filter the `price` array.
20 var array<bool> randMask = array.new<bool>(100, true)
21
22 // Push the first element from `randMask` to the end and queue a new pseudorandom value.
23 randMask.push(randMask.shift())
24 randMask.push(math.random(seed = 12345) < 0.5)
25 randMask.shift()
26 // Queue the `close` value into the `prices` array,
27
28 Line number: 31
29 Time: 95.1% (1.6s from 1.7s total)
30 Executions: 20207
31 plot(array.avg(filter(prices, randMask)))
32

```

The above code demonstrates suboptimal usage of a `for...in` loop, as we **do not** need to call `array.indexof()` to retrieve the `index` in this case. The `array.indexof()` function can be *costly* to call within a loop since it must search through the `array's` contents and locate the corresponding element's index *each time* the script calls it.

To eliminate this costly call from our `for...in` loop, we can use the *second form* of the structure, which produces a *tuple* containing the **index** and the element's value on each iteration:

```
for [index, item] in this
```

In this version of the script, we removed the `array.indexof()` call from the code since it is **not** necessary, and we changed the `for...in` loop to use the alternative form:

```

1  // @version=5
2  indicator("Reducing loop calculations demo", overlay = true)
3
4  //@function    Creates a filtered version of an array.
5  //@param this   The array of "bool" values to filter.
6  //@param mask   An array of "bool" values used to filter `this` array.
7  filter(array<float> this, array<bool> mask) ->
8      array<float> result = array.new<float>()
9      if this.size() != mask.size()
10         runtime.error("Cannot call `filter()` with two arrays of different sizes.")
11     for [index, item] in this
12         if array.get(mask, index)
13             result.push(item)
14     result
15
16 // @variable An array containing the most recent 100 `close` prices.
17 var array<float> prices = array.new<float>(100, close)
18 // @variable An array containing 100 pseudorandom `bool` values to filter the `price` array.
19 var array<bool> randMask = array.new<bool>(100, true)
20
21 // Push the first element from `randMask` to the end and queue a new pseudorandom

```

(continues on next page)

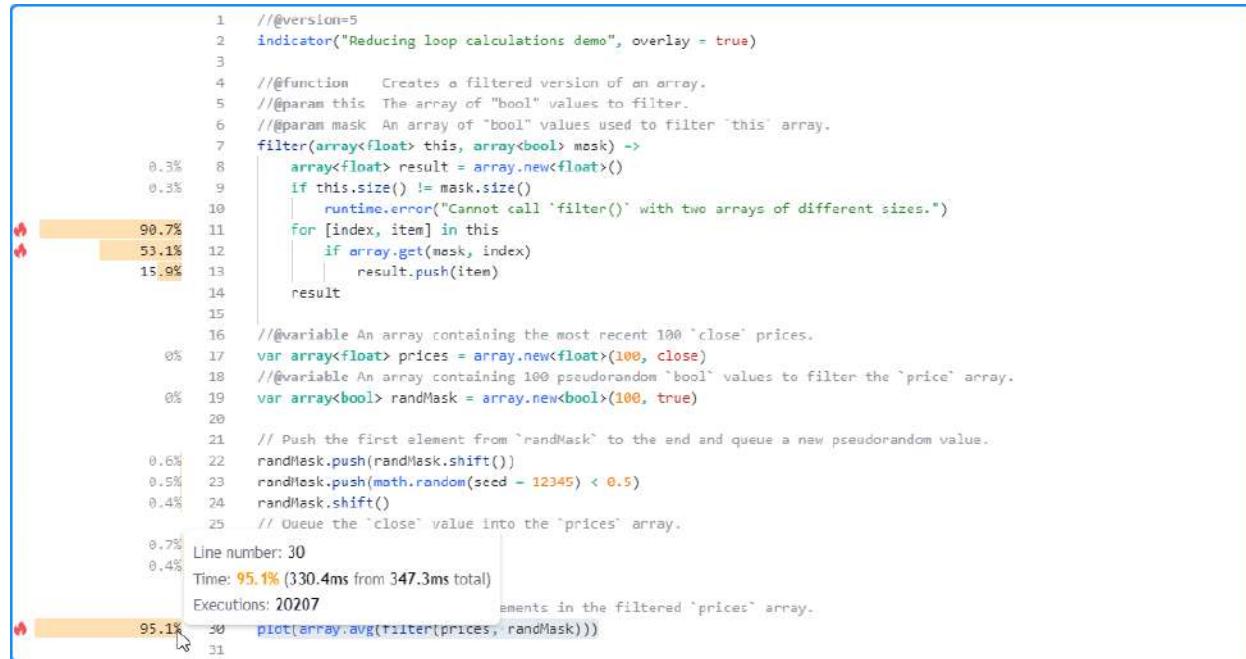
(continued from previous page)

```

1  ↵value.
2 randMask.push(randMask.shift())
3 randMask.push(math.random(seed = 12345) < 0.5)
4 randMask.shift()
5 // Queue the `close` value into the `prices` array.
6 prices.push(close)
7 prices.shift()
8
9 // Plot the average of the elements in the filtered `prices` array.
10 plot(array.avg(filter(prices, randMask)))

```

With this structure, our loop is much more efficient, as it no longer needs to redundantly search through the `array` on each iteration to keep track of the index. The *profiled results* from this script run show that it took only 347.3 milliseconds, a significant improvement from the previous version's result:



## Loop-invariant code motion

*Loop-invariant code* is any code region within a `loop`'s scope that produces an **unchanging** result on each iteration. When a script's `loops` contain loop-invariant code, it can substantially impact performance in some cases due to excessive, **unnecessary** calculations.

Programmers can optimize a loop with invariant code by *moving* the unchanging calculations **outside** the loop's scope so the script only needs to evaluate them once per execution rather than repetitively.

The following example contains a `featureScale()` function that creates a rescaled version of an `array`. Within the function's `for...in` loop, it scales each element by calculating its distance from the `array.min()` and dividing the value by the `array.range()`. The script uses this function to create a rescaled version of a `prices` array and *plots* the difference between the `rescaled.first()` and `rescaled.avg()` values on the chart:

```

1 // @version=5
2 indicator("Loop-invariant code motion demo")
3

```

(continues on next page)

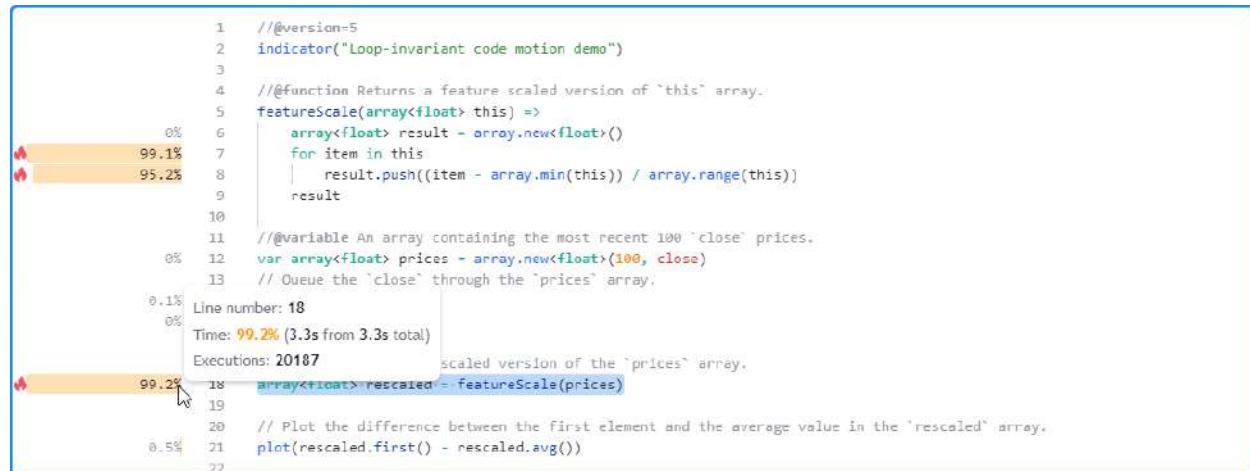
(continued from previous page)

```

4 // @function Returns a feature scaled version of `this` array.
5 featureScale(array<float> this) =>
6     array<float> result = array.new<float>()
7     for item in this
8         result.push((item - array.min(this)) / array.range(this))
9     result
10
11 // @variable An array containing the most recent 100 `close` prices.
12 var array<float> prices = array.new<float>(100, close)
13 // Queue the `close` through the `prices` array.
14 prices.unshift(close)
15 prices.pop()
16
17 // @variable A feature scaled version of the `prices` array.
18 array<float> rescaled = featureScale(prices)
19
20 // Plot the difference between the first element and the average value in the ↵
21 plot(rescaled.first() - rescaled.avg())

```

As we see below, the *profiled results* for this script after 20,187 executions show it completed its run in about 3.3 seconds. The code with the highest impact on performance is the line containing the `featureScale()` function call, and the function's critical code is the `for...in` loop block starting on line 7:



Upon examining the loop's calculations, we can see that the `array.min()` and `array.range()` calls on line 8 are **loop-invariant**, as they will always produce the **same result** across each iteration. We can make our loop much more efficient by assigning the results from these calls to variables **outside** its scope and referencing them as needed.

The `featureScale()` function in the script below assigns the `array.min()` and `array.range()` values to `minValue` and `rangeValue` variables *before* executing the `for...in` loop. Inside the loop's local scope, it *references* the variables across its iterations rather than repetitively calling these `array.*()` functions:

```

1 // @version=5
2 indicator("Loop-invariant code motion demo")
3
4 // @function Returns a feature scaled version of `this` array.
5 featureScale(array<float> this) =>
6     array<float> result = array.new<float>()
7     float minValue      = array.min(this)
8     float rangeValue    = array.range(this)

```

(continues on next page)

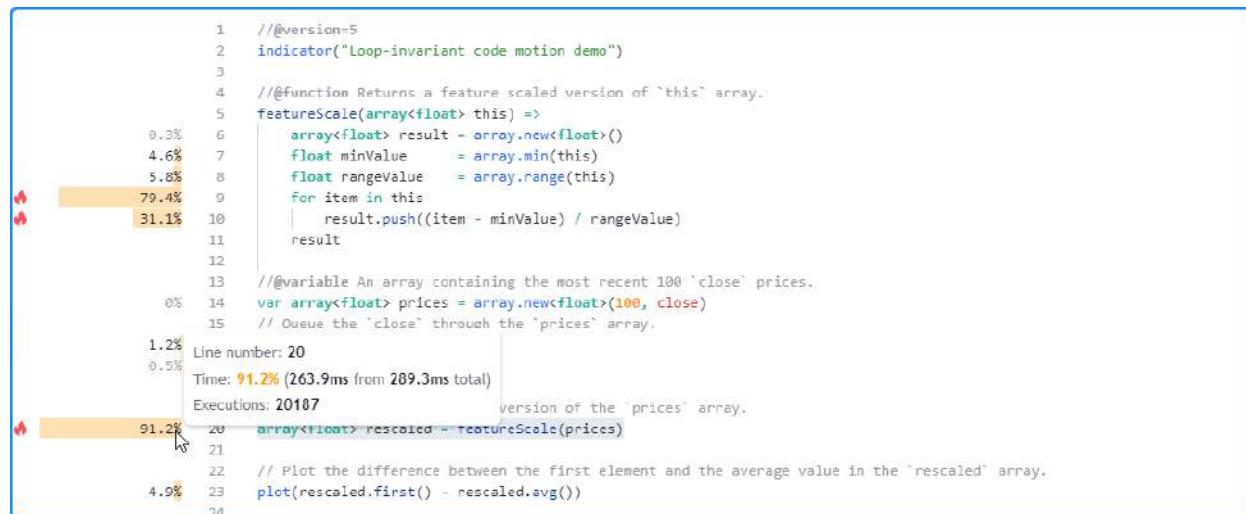
(continued from previous page)

```

9   for item in this
10    result.push((item - minValue) / rangeValue)
11  result
12
13 // @variable An array containing the most recent 100 `close` prices.
14 var array<float> prices = array.new<float>(100, close)
15 // Queue the `close` through the `prices` array.
16 prices.unshift(close)
17 prices.pop()
18
19 // @variable A feature scaled version of the `prices` array.
20 array<float> rescaled = featureScale(prices)
21
22 // Plot the difference between the first element and the average value in the ↵
23 // `rescaled` array.
24 plot(rescaled.first() - rescaled.avg())

```

As we see from the script's *profiled results*, moving the *loop-invariant* calculations outside the loop leads to a substantial performance improvement. This time, the script completed all its executions in only 289.3 milliseconds:



## Minimizing historical buffer calculations

Pine scripts create *historical buffers* for all variables and function calls their outputs depend on. Each buffer contains information about the range of historical values the script can access with the history-referencing operator `[]`.

A script *automatically* determines the required buffer size for all its variables and function calls by analyzing the historical references executed during the **first 244 bars** in a dataset. When a script only references the history of a calculated value *after* those initial bars, it will **restart** its executions repetitively across previous bars with successively larger historical buffers until it either determines the appropriate size or raises a runtime error. Those repetitive executions can significantly increase a script's runtime in some cases.

When a script *excessively* executes across a dataset to calculate historical buffers, one effective way to improve its performance is *explicitly* defining suitable buffer sizes using the `max_bars_back()` function. With appropriate buffer sizes declared explicitly, the script does not need to re-execute across past data to determine the sizes.

For example, the script below uses a *polyline* to draw a basic histogram representing the distribution of calculated source values over 500 bars. On the `last` available bar, the script uses a `for` loop to look back through historical values of

the calculated source series and determine the *chart points* used by the polyline drawing. It also *plots* the value of `bar_index + 1` to verify the number of bars it executed across:

```

1 // @version=5
2 indicator("Minimizing historical buffer calculations demo", overlay = true)
3
4 // @variable A polyline with points that form a histogram of `source` values.
5 var polyline display = na
6 // @variable The difference Q3 of `high` prices and Q1 of `low` prices over 500 bars.
7 float innerRange = ta.percentile_nearest_rank(high, 500, 75) - ta.percentile_nearest_
8 rank(low, 500, 25)
9 // Calculate the highest and lowest prices, and the total price range, over 500 bars.
10 float highest = ta.highest(500)
11 float lowest = ta.lowest(500)
12 float totalRange = highest - lowest
13
14 // @variable The source series for histogram calculation. Its value is the midpoint_
15 // between the `open` and `close`.
16 float source = math.avg(open, close)
17
18 if barstate.islast
19     polyline.delete(display)
20     // Calculate the number of histogram bins and their size.
21     int bins = int(math.round(5 * totalRange / innerRange))
22     float binSize = totalRange / bins
23     // @variable An array of chart points for the polyline.
24     array<chart.point> points = array.new<chart.point>(bins, chart.point.new(na, na,_
25     na))
26     // Loop to build the histogram.
27     for i = 0 to 499
28         // @variable The histogram bin number. Uses past values of the `source` for_
29         // its calculation.
30         //
31         // The script must execute across all previous bars AGAIN to_
32         // determine the historical buffer for
33         // `source`, as initial references to the calculated series occur_
34         // AFTER the first 244 bars.
35         int index = int((source[i] - lowest) / binSize)
36         if na(index)
37             continue
38         chart.point currentPoint = points.get(index)
39         if na(currentPoint.index)
40             points.set(index, chart.point.from_index(bar_index + 1, (index + 0.5) *_
41             binSize + lowest))
42             continue
43             currentPoint.index += 1
44             // Add final points to the `points` array and draw the new `display` polyline.
45             points.unshift(chart.point.now(lowest))
46             points.push(chart.point.now(highest))
47             display := polyline.new(points, closed = true)
48
49 plot(bar_index + 1, "Number of bars", display = display.data_window)

```

Since the script *only* references past source values on the *last bar*, it will **not** construct a suitable historical buffer for the series within the first 244 bars on a larger dataset. Consequently, it will **re-execute** across all historical bars to identify the appropriate buffer size.

As we see from the *profiled results* after running the script across 20,320 bars, the number of *global* code executions was 162,560, which is **eight times** the number of chart bars. In other words, the script had to *repeat* the historical executions **seven more times** to determine the appropriate buffer for the source series in this case:

```

1 //@version=5
2 indicator("Minimizing historical buffer calculations demo", overlay = true)
Line number: 7
Time: 0.25% (1.1s from 1.3s total) na
Executions: 162560
with points that form a histogram of `source` values.
8 float innerRange = ta.percentile_nearest_rank(high, 500, 75) - ta.percentile_nearest_rank(low, 500, 25)
9 // calculate the highest and lowest prices, and the total price range, over 500 bars.
10 float highest = ta.highest(500)
11 float lowest = ta.lowest(500)
12 float totalRange = highest - lowest
13 //variable The source series for histogram calculation. Its value is the midpoint between the `open` and `close`.
14 float source = math.avg(open, close)
15
16 if barstate.islast
17     polyline.delete(display)
18     // Calculate the number of histogram bins and their size.
19     int bins = int(math.round(5 * totalRange / innerRange))
20     float binSize = totalRange / bins
21     //variable An array of chart points for the polyline.
22     array<chart.point> points = array.new<chart.point>(bins, chart.point.new(na, na, na))
23     // loop to build the histogram.
24     for i = 0 to 499
25         //variable The histogram bin number. Uses past values of the `source` for its calculation.
26         // The script must execute across all previous bars AGAIN to determine the historical buffer for
27         // `source`, as initial references to the calculated series occur AFTER the first 244 bars.
28         int index = int((source[i] - lowest) / binSize)
29         if na(index)

```

This script will only reference the most recent 500 source values on the last historical bar and all realtime bars. Therefore, we can help it establish the correct buffer *without* re-execution by defining a 500-bar referencing length with `max_bars_back()`.

In the following script version, we added `max_bars_back(source, 500)` after the variable declaration to explicitly specify that the script will access up to 500 historical source values throughout its executions:

```

1 //@version=5
2 indicator("Minimizing historical buffer calculations demo", overlay = true)
3
4 //@variable A polyline with points that form a histogram of `source` values.
5 var polyline display = na
6 //@variable The difference Q3 of `high` prices and Q1 of `low` prices over 500 bars.
7 float innerRange = ta.percentile_nearest_rank(high, 500, 75) - ta.percentile_nearest_
8 rank(low, 500, 25)
// Calculate the highest and lowest prices, and the total price range, over 500 bars.
9 float highest = ta.highest(500)
10 float lowest = ta.lowest(500)
11 float totalRange = highest - lowest
12
13 //@variable The source series for histogram calculation. Its value is the midpoint_
14 //between the `open` and `close`.
15 float source = math.avg(open, close)
16 // Explicitly define a 500-bar historical buffer for the `source` to prevent_
17 //recalculation.
18 max_bars_back(source, 500)
19
20 if barstate.islast
21     polyline.delete(display)
22     // Calculate the number of histogram bins and their size.
23     int bins = int(math.round(5 * totalRange / innerRange))
24     float binSize = totalRange / bins
25     //variable An array of chart points for the polyline.
26     array<chart.point> points = array.new<chart.point>(bins, chart.point.new(na, na,_
27 na))
28     // Loop to build the histogram.
29     for i = 0 to 499
         //variable The histogram bin number. Uses past values of the `source` for_
        //its calculation.
         // Since the `source` now has an appropriate predefined buffer, the_

```

(continues on next page)

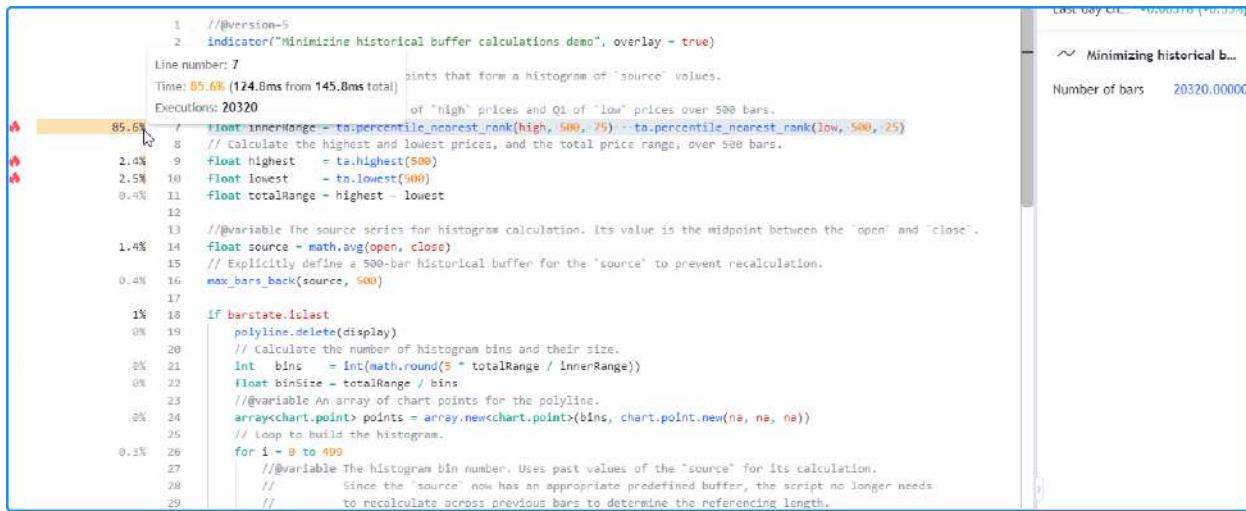
(continued from previous page)

```

29     ↵script no longer needs
30         //           to recalculate across previous bars to determine the referencing_
31         ↵length.
32             int index = int((source[i] - lowest) / binSize)
33             if na(index)
34                 continue
35             chart.point currentPoint = points.get(index)
36             if na(currentPoint.index)
37                 points.set(index, chart.point.from_index(bar_index + 1, (index + 0.5) *_
38             ↵binSize + lowest))
39                 continue
40             currentPoint.index += 1
41             // Add final points to the `points` array and draw the new `display` polyline.
42             points.unshift(chart.point.now(lowest))
43             points.push(chart.point.now(highest))
44             display := polyline.new(points, closed = true)
45
46 plot(bar_index + 1, "Number of bars", display = display.data_window)

```

With this change, our script no longer needs to re-execute across all the historical data to determine the buffer size. As we see in the [profiled results](#) below, the number of global code executions now aligns with the number of chart bars, and the script took substantially less time to complete all of its historical executions:



### Note that:

- This script only requires up to the most recent 501 historical bars to calculate its drawing output. In this case, another way to optimize resource usage is to include `calc_bars_count = 501` in the `indicator()` function, which reduces unnecessary script executions by restricting the historical data the script can calculate across to 501 bars.

**Note:** When explicitly defining a buffer size for a problematic historical reference with `max_bars_back()`, it's imperative to ensure that the script **will not** use more data than specified later in its executions, as the script will still re-execute on historical bars and try to calculate the buffer if the user-specified size is insufficient.

Another consideration when explicitly defining buffer sizes is that the larger the buffer, the larger the *memory cost*. As such, programmers should aim to keep the explicit buffer length limited to **only** the maximum number of historical values the script will reference and **not more**. For example, defining a 5000-bar buffer when a script only requires 500 historical values will result in an unnecessary waste of memory.

## 5.3.4 Tips

### Working around Profiler overhead

Since the *Pine Profiler* must perform *extra calculations* to collect performance data, as explained in [this section](#), the time it takes to execute a script **increases** while profiling.

Most scripts will run as expected with the Profiler's overhead included. However, when a complex script's runtime approaches a plan's limit, using the *Profiler* on it may cause its runtime to *exceed* the limit. Such a case indicates that the script likely needs *optimization*, but it can be challenging to know where to start without being able to [profile the code](#). The most effective workaround in this scenario is reducing the number of bars the script must execute on. Users can achieve this reduction in any of the following ways:

- Selecting a dataset that has fewer data points in its history, e.g., a higher timeframe or a symbol with limited data
- Using conditional logic to limit code executions to a specific time or bar range
- Including a `calc_bars_count` argument in the script's declaration statement to specify how many recent historical bars it can use

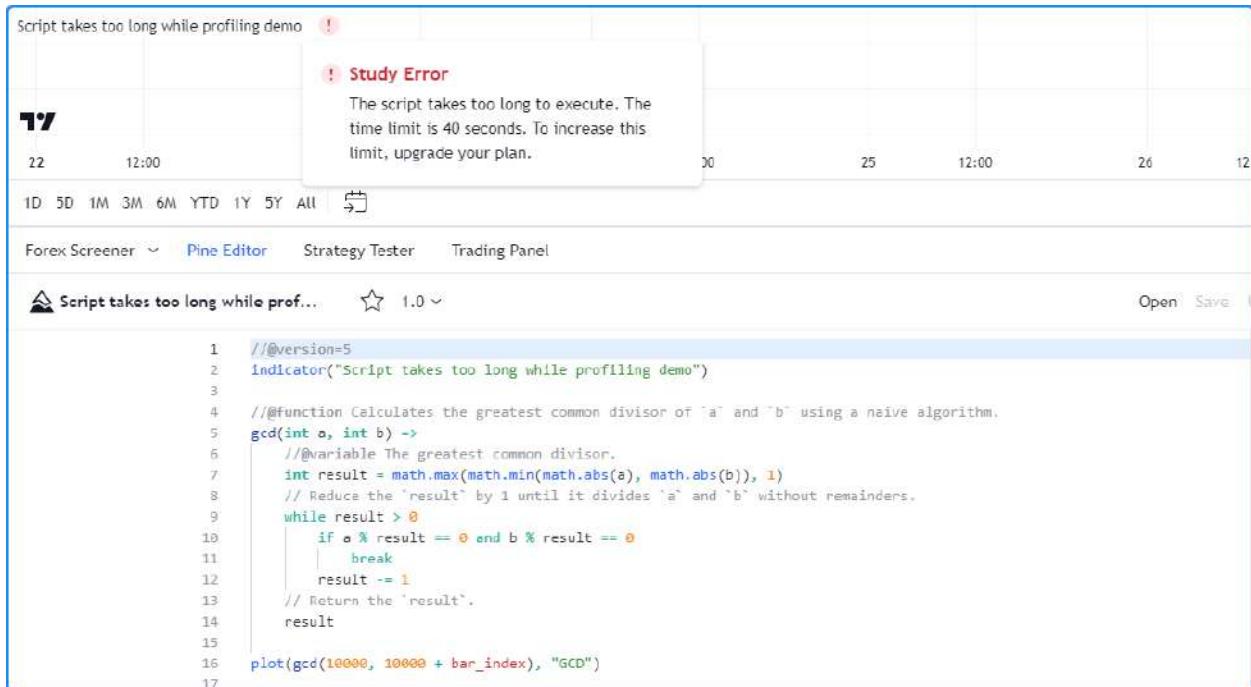
Reducing the number of data points works in most cases because it directly decreases the number of times the script must execute, typically resulting in less accumulated runtime.

As a demonstration, this script contains a `gcd()` function that uses a *naive* algorithm to calculate the *greatest common divisor* of two integers. The function initializes its `result` using the smallest absolute value of the two numbers. Then, it reduces the value of the `result` by one within a `while` loop until it can divide both numbers without remainders. This structure entails that the loop will iterate up to  $N$  times, where  $N$  is the smallest of the two arguments.

In this example, the script plots the value of `gcd(10000, 10000 + bar_index)`. The smallest of the two arguments is always 10,000 in this case, meaning the `while` loop within the function will require up to 10,000 iterations per script execution, depending on the `bar_index` value:

```
1 // @version=5
2 indicator("Script takes too long while profiling demo")
3
4 // @function Calculates the greatest common divisor of `a` and `b` using a naive
5 //      ↪algorithm.
6 gcd(int a, int b) =>
7     // @variable The greatest common divisor.
8     int result = math.max(math.min(math.abs(a), math.abs(b)), 1)
9     // Reduce the `result` by 1 until it divides `a` and `b` without remainders.
10    while result > 0
11        if a % result == 0 and b % result == 0
12            break
13        result -= 1
14    // Return the `result`.
15    result
16 plot(gcd(10000, 10000 + bar_index), "GCD")
```

When we add the script to our chart, it executes slowly, but it does not raise an error. However, *after* enabling the *Profiler*, the script raises a runtime error stating that it exceeded the Premium plan's *runtime limit* (40 seconds):



Our current chart has over 20,000 historical bars, which may be too many for the script to handle within the allotted time while the *Profiler* is active. We can try limiting the number of historical executions to work around the issue in this case.

Below, we included `calc_bars_count = 10000` in the `indicator()` function, which limits the script's available history to the most recent 10,000 historical bars. After restricting the script's historical executions, it no longer exceeds the Premium plan's limit while profiling, so we can now inspect its performance results:



```

1 //version=5
2 indicator("Script takes too long while profiling demo", calc_bars_count = 10000)
3

```

(continues on next page)

(continued from previous page)

```

4 // @function Calculates the greatest common divisor of `a` and `b` using a naive
5 // ↪ algorithm.
6 gcd(int a, int b) =>
7     // @variable The greatest common divisor.
8     int result = math.max(math.min(math.abs(a), math.abs(b)), 1)
9     // Reduce the `result` by 1 until it divides `a` and `b` without remainders.
10    while result > 0
11        if a % result == 0 and b % result == 0
12            break
13        result -= 1
14    // Return the `result`.
15    result
16
17 plot(gcd(10000, 10000 + bar_index), "GCD")

```

**Note:** This process may require trial and error, as determining the number of executions that a computationally heavy script can handle before timing out is not necessarily straightforward. If a script takes too long to execute after enabling the *Profiler*, experiment with different ways to limit its executions until you can profile it successfully.



## 5.4 Publishing scripts

- *Script visibility and access*
- *Preparing a publication*
- *Publishing a script*
- *Updating a publication*

Programmers who wish to share their Pine scripts with other traders can publish them.

**Note:** If you write scripts for your personal use, there is no need to publish them; you can save them in the Pine Editor and use the “Add to Chart” button to add your script to your chart.

### 5.4.1 Script visibility and access

When you publish a script, you control its **visibility** and **access**:

- **Visibility** is controlled by choosing to publish **publicly** or **privately**. See [How do private ideas and scripts differ from public ones?](#) in the Help Center for more details. Publish publicly when you have written a script you think can be useful to TradingViewers. Public scripts are subject to moderation. To avoid moderation, ensure your publication complies with our [House Rules](#) and [Script Publishing Rules](#). Publish privately when you don't want your script visible to all other users, but want to share it with a few friends.
- **Access** determines if users will see your source code, and how they will be able to use your script. There are three access types: *open*, *protected* (reserved to paid accounts) or *invite-only* (reserved to Premium accounts). See [What are the different types of published scripts?](#) in the Help Center for more details.

#### When you publish a script

- The publication's title is determined by the argument used for the `title` parameter in the script's `indicator()` or `strategy()` declaration statement. That title is also used when TradingViewers search for script names.
- The name of your script on the chart will be the argument used for the `shorttitle` parameter in the script's `indicator()` or `strategy()` declaration statement, or the `title` argument in `library()`.
- Your script must have a description explaining what your script does and how to use it.
- The chart you are using when you publish will become visible in your publication, including any other scripts or drawings on it. Remove unrelated scripts or drawings from your chart before publishing your script.
- Your script's code can later be updated. Each update can include *release notes* which will appear, dated, under your original description.
- Scripts can be liked, shared, commented on or reported by other users.
- Your published scripts appear under the “SCRIPTS” tab of your user profile.
- A *script widget* and a *script page* are created for your script. The script widget is your script's placeholder showing in script feeds on the platform. It contains your script's title, chart and the first few lines of your description. When users click on your script **widget**, the script's **page** opens. It contains all the information relating to your script.

#### Visibility

##### Public

When you publish a public script:

- Your script will be included in our [Community Scripts](#) where it becomes visible to the millions of TradingViewers on all internationalized versions of the site.
- Your publication must comply with [House Rules](#) and [Script Publishing Rules](#).
- If your script is an invite-only script, you must comply with our [Vendor Requirements](#).
- It becomes accessible through the search functions for scripts.
- You will not be able to edit your original description or its title, nor change its public/private visibility, nor its access type (open-source, protected, invite-only).
- You will not be able to delete your publication.

### Private

When you publish a private script:

- It will not be visible to other users unless you share its url with them.
- It is visible to you from your user profile's "SCRIPTS" tab.
- Private scripts are identifiable by the "X" and "lock" icons in the top-right of their widget. The "X" is used to delete it.
- It is not moderated, unless you sell access to it or make it available publicly, as it is then no longer "private".
- You can update its original description and title.
- You cannot link to or mention it from any public TradingView content (ideas, script descriptions, comments, chats, etc.).
- It is not accessible through the search functions for scripts.

### Access

Public or private scripts can be published using one of three access types: open, protected or invite-only. The access type you can select from will vary with the type of account you hold.

#### Open

The Pine Script™ code of scripts published **open** is visible to all users. Open-source scripts on TradingView use the Mozilla license by default, but you may choose any license you want. You can find information on licensing at [GitHub](#).

#### Protected

The code of **protected** scripts is hidden from view and no one but its author can access it. While the script's code is not accessible, protected scripts can be used freely by any user. Only Pro, Pro+ or Premium accounts may publish public protected scripts.

#### Invite-only

The **invite-only** access type protects both the script's code and its use. The publisher of an invite-only script must explicitly grant access to individual users. Invite-only scripts are mostly used by script vendors providing paid access to their scripts. Only Premium accounts can publish invite-only scripts, and they must comply with our [Vendor Requirements](#).

TradingView does not benefit from script sales. Transactions concerning invite-only scripts are strictly between users and vendors; they do not involve TradingView.

Public invite-only scripts are the only scripts for which vendors are allowed to ask for payment on TradingView.

On their invite-only script's page, authors will see a "Manage Access" button. The "Manage Access" window allows authors to control who has access to their script.

The screenshot shows the Pine Editor interface with the Aroon script open. At the top, there's a chart area with a candlestick pattern. Below the chart, a text box contains the script code:

```

1 //@version=4
2 study("Aroon with dynamic colors", "Aroon", false, format.percent, 2)
3 i_length = input(25, minval = 1)
4 float upper = 100 * (highestbars(high, i_length + 1) + i_length) / i_length
5 float lower = 100 * (lowestbars(low, i_length + 1) + i_length) / i_length
6 plot(upper, "Upper", upper > lower ? #00FF00FF : #00FF0060)
7 plot(lower, "Lower", lower > upper ? #FF0000FF : #FF000060)
8

```

Below the code, there are several interface elements:

- Invite-only script**: A note indicating the source code is protected.
- Author's instructions**: A note asking for a private message to request access.
- Want to use this script on a chart?**: A note with a warning to read before requesting access.
- Add to favorite indicators** and **Manage Access** buttons. An orange arrow points from the text above to the **Manage Access** button.
- Tools and Ideas for all Pine coders**, **Pine FAQ & Code**, and **Pine news broadcasts** links.
- Comments**: A section for leaving comments, with a note to leave helpful or encouraging comments.

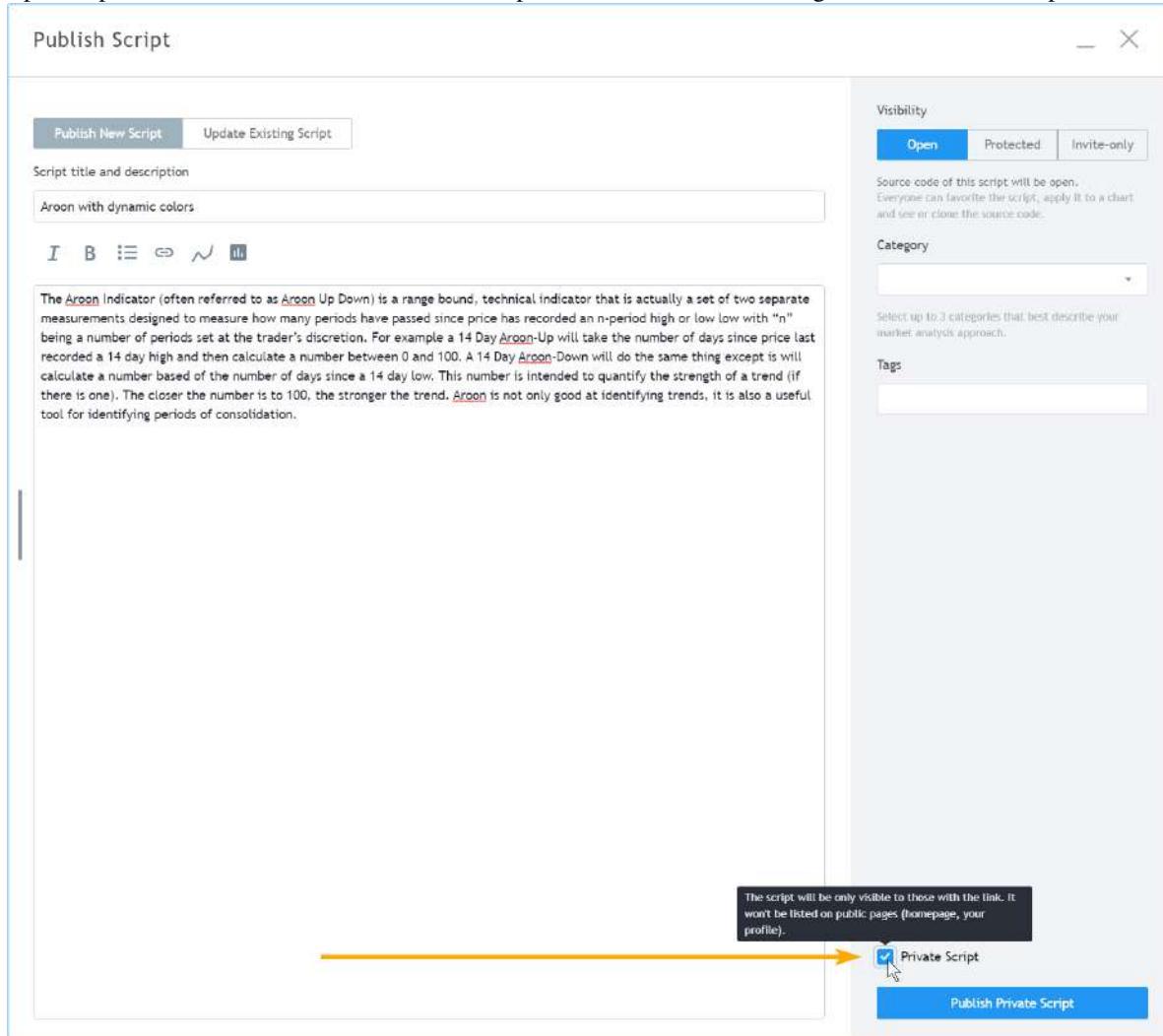
## 5.4.2 Preparing a publication

- Even if you intend to publish publicly, it is always best to start with a private publication because you can use it to validate what your final publication will look like. You can edit the title, description, code or chart of private publications, and contrary to public scripts, you can delete private scripts when you don't need them anymore, so they are the perfect way to practice before sharing a script publicly. You can read more about preparing script descriptions in the [How We Write and Format Script Descriptions](#) publication.
- Prepare your chart. Load your script on the chart and remove other scripts or drawings that won't help users understand your script. Your script's plots should be easy to identify on the chart that will be published with it.
- Load your code in the Pine Editor if it isn't already. In the Editor, click the “Publish Script” button:

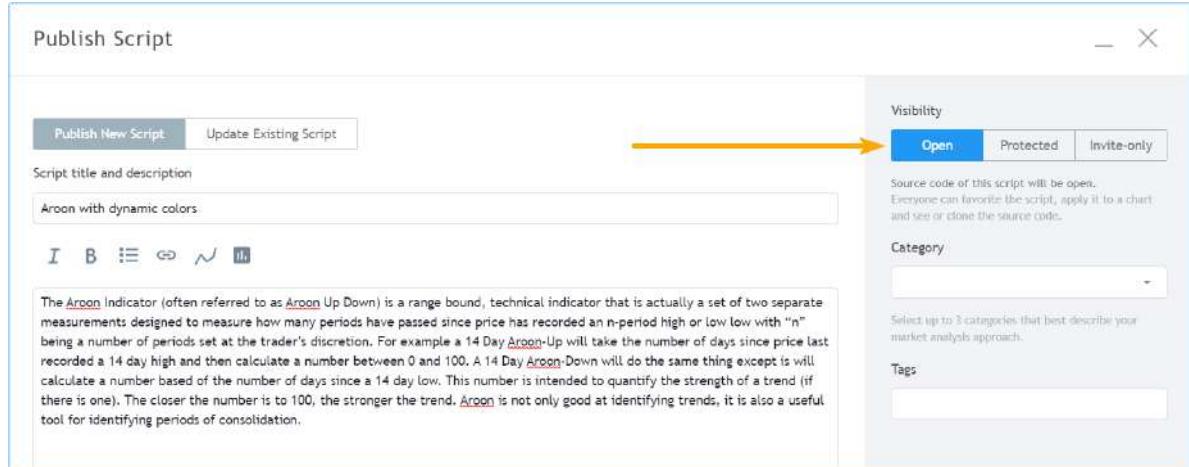


- A popup appears to remind you that if you publish publicly, it's important that your publication comply with House Rules. Once you're through the popup, place your description in the field below the script's title. The default title proposed for your publication is the `title` field from your script's code. It is always best to use that title; it makes it easier for users to search for your script if it is public. Select the visibility of your publication. We want to publish

a private publication, so we check the “Private Script” checkbox at the bottom-right of the “Publish Script” window:

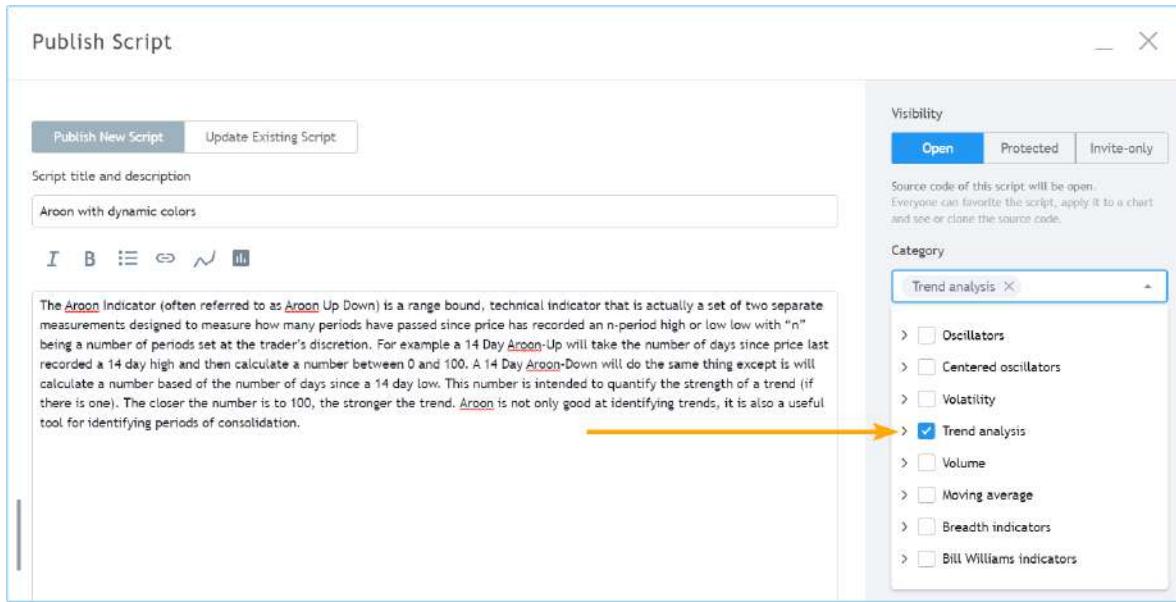


5. Select the access type you want for your script: Open, Protected or Invite-only. We have selected “Open” for open-



source.

6. Select the appropriate categories for your script (at least one is mandatory) and enter optional custom tags.



- Click the “Publish Private Script” button in the lower-right of the window. When the publication is complete, your published script’s page will appear. You are done! You can confirm the publication by going to your User Profile and viewing your “SCRIPTS” tab. From there, you will be able to open your script’s page and edit your private publication by using the “Edit” button in the top-right of your script’s page. Note that you can also update private publications, just like you can public ones. If you want to share your private publication with a friend, privately send her the url from your script’s page. Remember you are not allowed to share links to private publications in public TradingView content.

### 5.4.3 Publishing a script

Whether you intend to publish privately or publicly, first follow the steps in the previous section. If you intend to publish privately, you will be done. If you intend to publish publicly and are satisfied with the preparatory process of validating your private publication, follow the same steps as above but do not check the “Private Script” checkbox and click the “Publish Public Script” button at the bottom-right of the “Publish Script” page.

When you publish a new public script, you have a 15-minute window to make changes to your description or delete the publication. After that you will no longer be able to change your publication’s title, description, visibility or access type. If you make an error, send a message to the [PineCoders](#) moderator account; they moderate script publications and will help.

### 5.4.4 Updating a publication

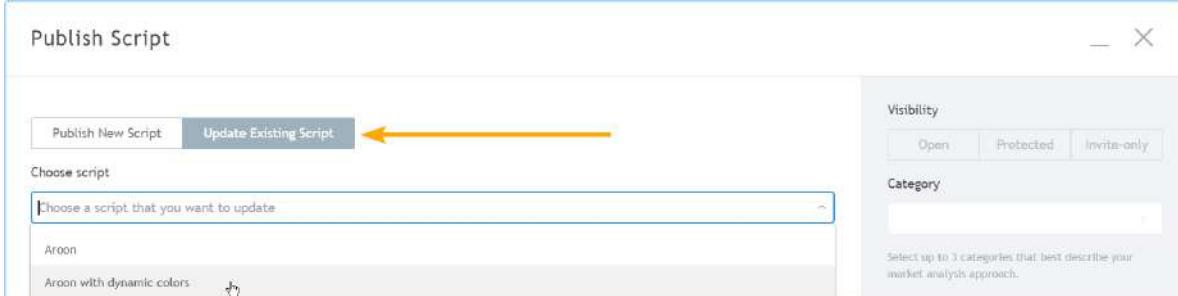
You can update both public or private script publications. When you update a script, its code must be different than the previously published version’s code. You can add release notes with your update. They will appear after your script’s original description in the script’s page.

By default, the chart used when you update will replace the previous chart in your script’s page. You can choose not to update your script page’s chart, however. Note that while you can update the chart displayed in the script’s page, the chart from the script’s widget will not update.

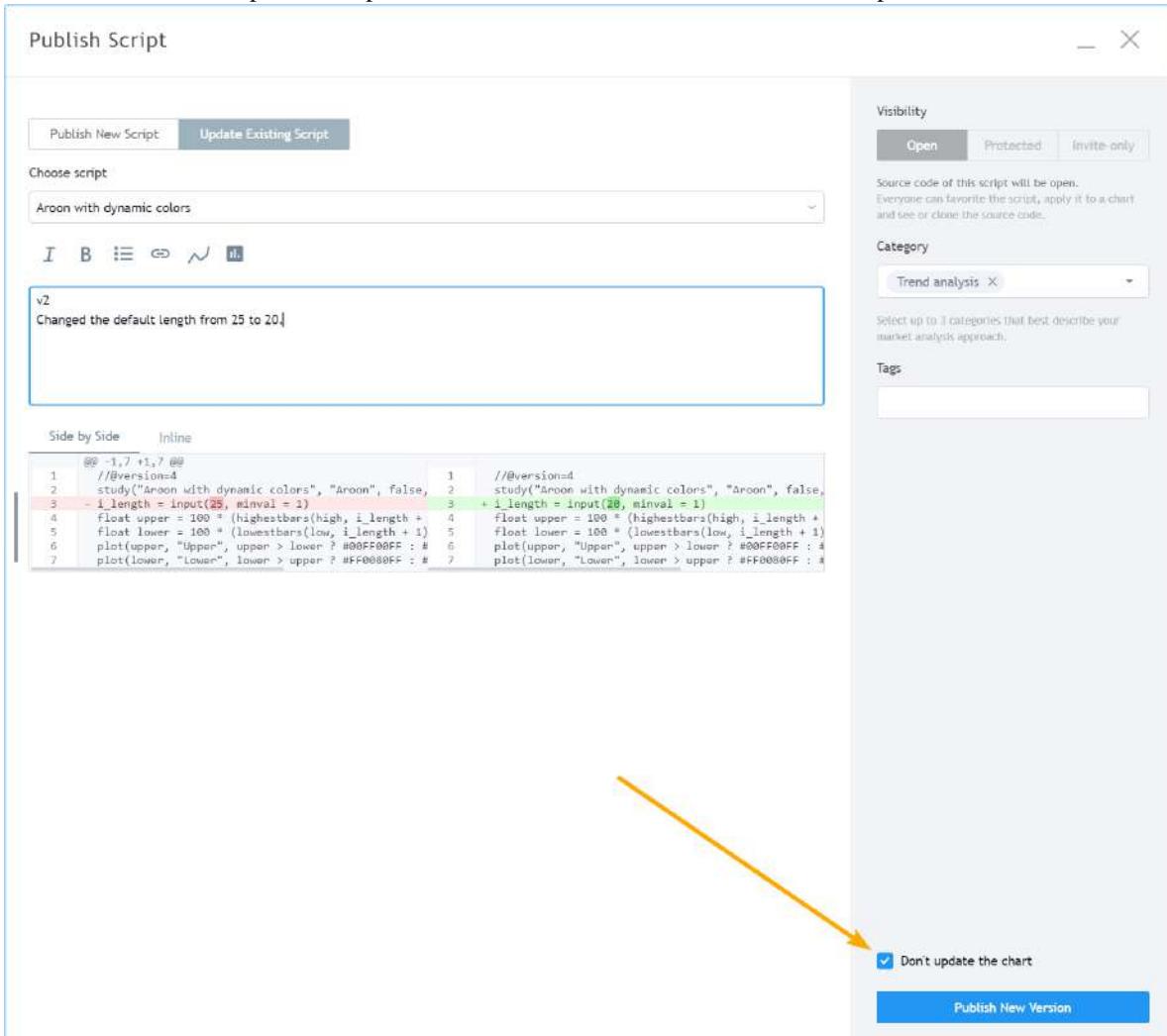
In the same way you can validate a public publication by first publishing a private script, you can also validate an update on a private publication before proceeding with it on your public one. The process of updating a published script is the same for public and private scripts.

If you intend to update both the code and chart of your published script, prepare your chart the same way you would for a new publication. In the following example, we will **not** be updating the publication's chart:

1. As you would for a new publication, load your script in the Editor and click the “Publish Script” button.
2. Once in the “Publish Script” window, select the “Update Existing Script” button. Then select the script to update from the “Choose script” dropdown menu:



3. Enter your release notes in the text field. The differences in your code are highlighted below your release notes.
4. We do not want to update the publication's chart, so we check the “Don't update the chart” checkbox:



5. Click the “Publish New Version” button. You’re done.



## 5.5 Limitations

- *Introduction*
- *Time*
- *Chart visuals*
- *`request.\*()` calls*
- *Script size and memory*
- *Other limitations*

### 5.5.1 Introduction

As is mentioned in our [Welcome](#) page:

*Because each script uses computational resources in the cloud, we must impose limits in order to share these resources fairly among our users. We strive to set as few limits as possible, but will of course have to implement as many as needed for the platform to run smoothly. Limitations apply to the amount of data requested from additional symbols, execution time, memory usage and script size.*

If you develop complex scripts using Pine Script™, sooner or later you will run into some of the limitations we impose. This section provides you with an overview of the limitations that you may encounter. There are currently no means for Pine Script™ programmers to get data on the resources consumed by their scripts. We hope this will change in the future.

In the meantime, when you are considering large projects, it is safest to make a proof of concept in order to assess the probability of your script running into limitations later in your project.

Below, we describe the limits imposed in the Pine Script™ environment.

### 5.5.2 Time

#### Script compilation

Scripts must compile before they are executed on charts. Compilation occurs when you save a script from the Pine Editor or when you add a script to the chart. A two-minute limit is imposed on compilation time, which will depend on the size and complexity of your script, and whether or not a cached version of a previous compilation is available. When a compile exceeds the two-minute limit, a warning is issued. Heed that warning by shortening your script because after three consecutive warnings a one-hour ban on compilation attempts is enforced. The first thing to consider when optimizing code is to avoid repetitions by using functions to encapsulate oft-used segments, and call functions instead of repeating code.

## Script execution

Once a script is compiled it can be executed. See the [Events triggering the execution of a script](#) for a list of the events triggering the execution of a script. The time allotted for the script to execute on all bars of a dataset varies with account types. The limit is 20 seconds for basic accounts, 40 for others.

## Loop execution

The execution time for any loop on any single bar is limited to 500 milliseconds. The outer loop of embedded loops counts as one loop, so it will time out first. Keep in mind that even though a loop may execute under the 500 ms time limit on a given bar, the time it takes to execute on all the dataset's bars may nonetheless cause your script to exceed the total execution time limit. For example, the limit on total execution time will make it impossible for you script to execute a 400 ms loop on each bar of a 20,000-bar dataset because your script would then need 8000 seconds to execute.

### 5.5.3 Chart visuals

#### Plot limits

A maximum of 64 plot counts are allowed per script. The functions that generate plot counts are:

- `plot()`
- `plotarrow()`
- `plotbar()`
- `plotcandle()`
- `plotchar()`
- `plotshape()`
- `alertcondition()`
- `bgcolor()`
- `fill()`, but only if its `color` is of the `series` form.

The following functions do not generate plot counts:

- `hline()`
- `line.new()`
- `label.new()`
- `table.new()`
- `box.new()`

One function call can generate up to seven plot counts, depending on the function and how it is called. When your script exceeds the maximum of 64 plot counts, the runtime error message will display the plot count generated by your script. Once you reach that point, you can determine how many plot counts a function call generates by commenting it out in a script. As long as your script still throws an error, you will be able to see how the actual plot count decreases after you have commented out a line.

The following example shows different function calls and the number of plot counts each one will generate:

```

1 //@version=5
2 indicator("Plot count example")
3
4 bool isUp = close > open
5 color isUpColor = isUp ? color.green : color.red
6 bool isDn = not isUp
7 color isDnColor = isDn ? color.red : color.green
8
9 // Uses one plot count each.
10 p1 = plot(close, color = color.white)
11 p2 = plot(open, color = na)
12
13 // Uses two plot counts for the `close` and `color` series.
14 plot(close, color = isUpColor)
15
16 // Uses one plot count for the `close` series.
17 plotarrow(close, colorup = color.green, colordown = color.red)
18
19 // Uses two plot counts for the `close` and `colorup` series.
20 plotarrow(close, colorup = isUpColor)
21
22 // Uses three plot counts for the `close`, `colorup`, and the `colordown` series.
23 plotarrow(close - open, colorup = isUpColor, colordown = isDnColor)
24
25 // Uses four plot counts for the `open`, `high`, `low`, and `close` series.
26 plotbar(open, high, low, close, color = color.white)
27
28 // Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series.
29 plotbar(open, high, low, close, color = isUpColor)
30
31 // Uses four plot counts for the `open`, `high`, `low`, and `close` series.
32 plotcandle(open, high, low, close, color = color.white, wickcolor = color.white, ↵
33 bordercolor = color.purple)
34
35 // Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series.
36 plotcandle(open, high, low, close, color = isUpColor, wickcolor = color.white, ↵
37 bordercolor = color.purple)
38
39 // Uses six plot counts for the `open`, `high`, `low`, `close`, `color`, and ↵
39 wickcolor` series.
40 plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor, ↵
41 bordercolor = color.purple)
42
43 // Uses seven plot counts for the `open`, `high`, `low`, `close`, `color`, ↵
43 wickcolor` , and `bordercolor` series.
44 plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor, ↵
45 bordercolor = isUp ? color.lime : color.maroon)
46
47 // Uses one plot count for the `close` series.
48 plotchar(close, color = color.white, text = "|", textcolor = color.white)
49
50 // Uses two plot counts for the `close` and `color` series.
51 plotchar(close, color = isUpColor, text = "-", textcolor = color.white)
52
53 // Uses three plot counts for the `close`, `color`, and `textcolor` series.
54 plotchar(close, color = isUpColor, text = "O", textcolor = isUp ? color.yellow : ↵
55 color.white)

```

(continues on next page)

(continued from previous page)

```

51 // Uses one plot count for the `close` series.
52 plotshape(close, color = color.white, textcolor = color.white)
53
54 // Uses two plot counts for the `close` and `color` series.
55 plotshape(close, color = isUpColor, textcolor = color.white)
56
57 // Uses three plot counts for the `close`, `color`, and `textcolor` series.
58 plotshape(close, color = isUpColor, textcolor = isUp ? color.yellow : color.white)
59
60 // Uses one plot count.
61 alertcondition(close > open, "close > open", "Up bar alert")
62
63 // Uses one plot count.
64 bgcolor(isUp ? color.yellow : color.white)
65
66 // Uses one plot count for the `color` series.
67 fill(p1, p2, color = isUpColor)
68

```

This example generates a plot count of 56. If we were to add two more instances of the last call to `plotcandle()`, the script would throw an error stating that the script now uses 70 plot counts, as each additional call to `plotcandle()` generates seven plot counts, and  $56 + (7 * 2)$  is 70.

### Line, box, polyline, and label limits

Contrary to `plots`, which can cover the chart's entire dataset, scripts will only show the last 50 `lines`, `boxes`, `polylines`, and `labels` on the chart by default. One can increase the maximum number for each of these `drawing types` via the `max_lines_count`, `max_boxes_count`, `max_polylines_count`, and `max_labels_count` parameters of the script's `indicator()` or `strategy()` declaration statement. The maximum number of `line`, `box`, and `label` IDs is 500, and the maximum number of `polyline` IDs is 100.

In this example, we set the maximum number of recent labels shown on the chart to 100:

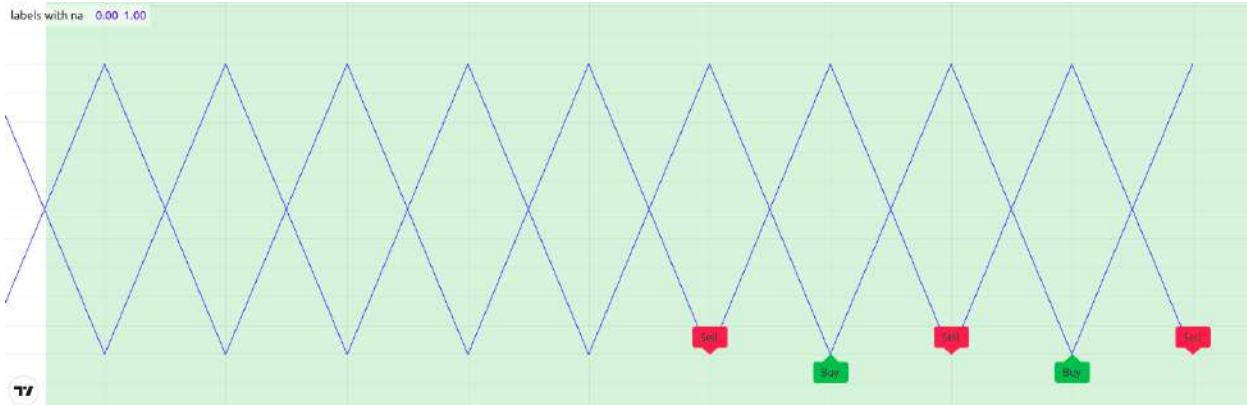
```

1 // @version=5
2 indicator("Label limits example", max_labels_count = 100, overlay = true)
3 label.new(bar_index, high, str.tostring(high, format.mintick))

```

It's important to note when setting any of a drawing object's properties to `na` that its ID still exists and thus contributes to a script's drawing totals. To demonstrate this behavior, the following script draws a "Buy" and "Sell" `label` on each bar, with `x` values determined by the `longCondition` and `shortCondition` variables.

The "Buy" label's `x` value is `na` when the bar index is even, and the "Sell" label's `x` value is `na` when the bar index is odd. Although the `max_labels_count` is 10 in this example, we can see that the script displays fewer than 10 `labels` on the chart since the ones with `na` values also count toward the total:

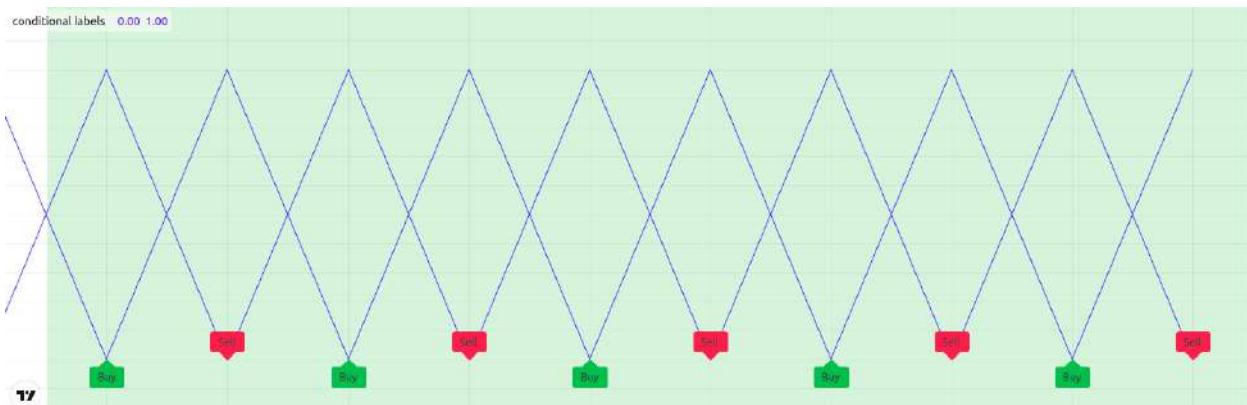


```

1 // @version=5
2
3 // Approximate maximum number of label drawings
4 MAX_LABELS = 10
5
6 indicator("labels with na", overlay = false, max_labels_count = MAX_LABELS)
7
8 // Add background color for the last MAX_LABELS bars.
9 bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na)
10
11 longCondition = bar_index % 2 != 0
12 shortCondition = bar_index % 2 == 0
13
14 // Add "Buy" and "Sell" labels on each new bar.
15 label.new(longCondition ? bar_index : na, 0, text = "Buy", color = color.new(color.
16   ↪green, 0), style = label.style_label_up)
16 label.new(shortCondition ? bar_index : na, 0, text = "Sell", color = color.new(color.
17   ↪red, 0), style = label.style_label_down)
17
18 plot(longCondition ? 1 : 0)
19 plot(shortCondition ? 1 : 0)

```

To display the desired number of labels, we must eliminate label drawings we don't want to show rather than setting their properties to `na`. The example below uses an `if` structure to conditionally draw the “Buy” and “Sell” labels, preventing the script from creating new label IDs when it isn’t necessary:



```

1 // @version=5
2

```

(continues on next page)

(continued from previous page)

```

3 // Approximate maximum number of label drawings
4 MAX_LABELS = 10
5
6 indicator("conditional labels", overlay = false, max_labels_count = MAX_LABELS)
7
8 // Add background color for the last MAX_LABELS bars.
9 bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na)
10
11 longCondition = bar_index % 2 != 0
12 shortCondition = bar_index % 2 == 0
13
14 // Add a "Buy" label when `longCondition` is true.
15 if longCondition
16     label.new(bar_index, 0, text = "Buy", color = color.new(color.green, 0), style =_
17         ↪label.style_label_up)
18 // Add a "Sell" label when `shortCondition` is true.
19 if shortCondition
20     label.new(bar_index, 0, text = "Sell", color = color.new(color.red, 0), style =_
21         ↪label.style_label_down)
22
23 plot(longCondition ? 1 : 0)
24 plot(shortCondition ? 1 : 0)

```

## Table limits

Scripts can display a maximum of nine *tables* on the chart, one for each of the possible locations: `position.bottom_center`, `position.bottom_left`, `position.bottom_right`, `position.middle_center`, `position.middle_left`, `position.middle_right`, `position.top_center`, `position.top_left`, and `position.top_right`. When attempting to place two tables in the same location, only the newest instance will show on the chart.

## 5.5.4 `request.\*()` calls

### Number of calls

A script cannot contain more than 40 calls to functions in the `request.` namespace. All instances of these functions count toward this limit, even when contained within local blocks of *user-defined functions* that aren't utilized by the script's main logic. This limitation applies to all functions discussed in the *Other timeframes and data* page, including:

- `request.security()`
- `request.security_lower_tf()`
- `request.currency_rate()`
- `request.dividends()`
- `request.splits()`
- `request.earnings()`
- `request.quandl()`
- `request.financial()`
- `request.economic()`
- `request.seed()`

## Intrabars

Scripts can retrieve up to the most recent 100,000 *intrabars* (lower-timeframe bars) via the `request.security()` or `request.security_lower_tf()` functions.

The number of bars on the chart's timeframe covered by 100,000 intrabars varies with the number of intrabars each chart bar contains. For example, requesting data from the 1-minute timeframe while running the script on a 60-minute chart means each chart bar can contain up to 60 intrabars. In this case, the minimum number of chart bars covered by the intrabar request is 1,666, as  $100,000 / 60 = 1,666.67$ . It's important to note, however, that a provider may not report data for *every* minute within an hour. Therefore, such a request may cover more chart bars, depending on the available data.

## Tuple element limit

All the `request.*()` function calls in a script taken together cannot return more than 127 tuple elements. When the combined tuple size of all `request.*()` calls will exceed 127 elements, one can instead utilize *user-defined types (UDTs)* to request a greater number of values.

The example below outlines this limitation and the way to work around it. The first `request.security()` call represents using a tuple with 128 elements as the `expression` argument. Since the number of elements is greater than 127, it would result in an error.

To avoid the error, we can use those same values as *fields* within an *object* of a *UDT* and pass its ID to the `expression` instead:

```

1 // @version=5
2 indicator("Tuple element limit")
3
4 s1 = close
5 s2 = close * 2
6 ...
7 s128 = close * 128
8
9 // Causes an error.
10 [v1, v2, v3, ..., v128] = request.security(syminfo.tickerid, "1D", [s1, s2, s3, ...,  
→s128])
11
12 // Works fine:
13 type myType
14     float v1
15     float v2
16     float v3
17     ...
18     float v128
19
20 myObj = request.security(syminfo.tickerid, "1D", myType.new(s1, s2, s3, ..., s128))

```

### Note that:

- This example outlines a scenario where the script tries to evaluate 128 tuple elements in a single `request.security()` call. The same limitation applies if we were to split the tuple request across *multiple* calls. For example, two `request.security()` calls that each retrieve a tuple with 64 elements will also cause an error.

## 5.5.5 Script size and memory

### Compiled tokens

Before the execution of a script, the compiler translates it into a tokenized *Intermediate Language* (IL). Using an IL allows Pine Script™ to accommodate larger scripts by applying various memory and performance optimizations. The compiler determines the size of a script based on the *number of tokens* in its IL form, **not** the number of characters or lines in the code viewable in the Pine Editor.

The compiled form of each indicator, strategy, and library script is limited to 68,000 tokens. When a script imports libraries, the total number of tokens from all imported libraries cannot exceed 1 million. There is no way to inspect a script's compiled form, nor its IL token count. As such, you will only know your script exceeds the size limit when the compiler reaches it.

In most cases, a script's compiled size will likely not reach the limit. However, if a compiled script does reach the token limit, the most effective ways to decrease compiled tokens are to reduce repetitive code, encapsulate redundant calls within functions, and utilize *libraries* when possible.

It's important to note that the compilation process omits any *unused* variables, functions, types, etc. from the final IL form, where "unused" refers to anything that *does not* affect the script's outputs. This optimization prevents superfluous elements in the code from contributing to the script's IL token count.

For example, the script below declares a *user-defined type* and a *user-defined method* and defines a sequence of calls using them:

```
1 // @version=5
2 indicator("My Script")
3 plot(close)
4
5 type myType
6     float field = 10.0
7
8 method m(array<myType> a, myType v) =>
9     a.push(v)
10
11 var arr = array.new<myType>()
12 arr.push(myType.new(25))
13 arr.m(myType.new())
```

Despite the inclusion of `array.new<myType>()`, `myType.new()`, and `arr.m()` calls in the script, the only thing actually **output** by the script is `plot(close)`. The rest of the code does not affect the output. Therefore, the compiled form of this script will have the *same* number of tokens as:

```
1 // @version=5
2 indicator("My Script")
3 plot(close)
```

## Variables per scope

Scripts can contain up to 1,000 variables in each of its scopes. Pine scripts always contain one global scope, represented by non-indented code, and they may contain zero or more local scopes. Local scopes are sections of indented code representing procedures executed within *functions* and *methods*, as well as *if*, *switch*, *for*, *for...in*, and *while* structures, which allow for one or more local blocks. Each local block counts as one local scope.

The branches of a conditional expression using the `?:` ternary operator do not count as local blocks.

## Scope count

The total number of scopes in a script, including its global scope and each local scope from the *user-defined functions*, *methods*, *conditional structures*, or *loops* it uses, cannot exceed 500.

It's important to note that the `request.security()`, `request.security_lower_tf()`, and `request.seed()` functions *duplicate* the scopes required to evaluate the values of their `expression` argument in another context. The scopes produced by each call to these `request.*()` functions also count toward the script's scope limit.

For example, suppose we created a script with a global variable that depends on the local scopes of 250 `if` structures. The total scope count for this script is 251 (1 global scope + 250 local scopes):

```

1 // @version=5
2 indicator("Scopes demo")
3
4 var x = 0
5
6 if close > 0
7     x += 0
8 if close > 1
9     x += 1
10 // ... Repeat this `if close > n` pattern until `n = 249`.
11 if close > 249
12     x += 249
13
14 plot(x)

```

Since the total number of scopes is within the limit, it will compile successfully. Now, suppose we call `request.security()` to evaluate the value of `x` from another context and `plot` its value as well. In this case, it will effectively *double* the script's scope count since the value of `x` depends on *all* the script's scopes:

```

1 // @version=5
2 indicator("Scopes demo")
3
4 var x = 0
5
6 if close > 0
7     x += 0
8 if close > 1
9     x += 1
10 // ... Repeat this `if close > n` pattern until `n = 249`.
11 if close > 249
12     x += 249
13
14 plot(x)
15 plot(request.security(syminfo.tickerid, "1D", x)) // Causes compilation error since
    ↵the scope count is now 502.

```

We can resolve this issue by encapsulating the `if` blocks within a *user-defined function*, as the scope of a function counts as one embedded scope:

```

1 // @version=5
2 indicator("Scopes demo")
3
4 f () =>
5     var x = 0
6
7     if close > 0
8         x += 0
9     if close > 1
10        x += 1
11    // ... Repeat this `if close > n` pattern until `n = 249`.
12    if close > 249
13        x += 249
14
15 plot(f())
16 plot(request.security(syminfo.tickerid, "1D", f())) // No compilation error.

```

## Collections

Pine Script™ collections (*arrays*, *matrices*, and *maps*) can have a maximum of 100,000 elements. Each key-value pair in a map contains two elements, meaning *maps* can contain a maximum of 50,000 key-value pairs.

### 5.5.6 Other limitations

#### Maximum bars back

References to past values using the `[]` history-referencing operator are dependent on the size of the historical buffer maintained by the Pine Script™ runtime, which is limited to a maximum of 5000 bars. This Help Center page discusses the historical buffer and how to change its size using either the `max_bars_back` parameter or the `max_bars_back()` function.

#### Maximum bars forward

When positioning drawings using `xloc.bar_index`, it is possible to use bar index values greater than that of the current bar as *x* coordinates. A maximum of 500 bars in the future can be referenced.

This example shows how we use the `maxval` parameter in our `input.int()` function call to cap the user-defined number of bars forward we draw a projection line so that it never exceeds the limit:

```

1 // @version=5
2 indicator("Max bars forward example", overlay = true)
3
4 // This function draws a `line` using bar index x-coordinates.
5 drawLine(bar1, y1, bar2, y2) =>
6     // Only execute this code on the last bar.
7     if barstate.islast
8         // Create the line only the first time this function is executed on the last
9         // bar.
10        var line lin = line.new(bar1, y1, bar2, y2, xloc.bar_index)
11        // Change the line's properties on all script executions on the last bar.

```

(continues on next page)

(continued from previous page)

```

11     line.set_xy1(lin, bar1, y1)
12     line.set_xy2(lin, bar2, y2)
13
14 // Input determining how many bars forward we draw the `line`.
15 int forwardBarsInput = input.int(10, "Forward Bars to Display", minval = 1, maxval =_
16   ↴500)
17
18 // Calculate the line's left and right points.
19 int    leftBar  = bar_index[2]
20 float  leftY    = high[2]
21 int    rightBar = leftBar + forwardBarsInput
22 float  rightY   = leftY + (ta.change(high)[1] * forwardBarsInput)
23
24 // This function call is executed on all bars, but it only draws the `line` on the_
25   ↴last bar.
26 drawLine(leftBar, leftY, rightBar, rightY)

```

## Chart bars

The number of bars appearing on charts is dependent on the amount of historical data available for the chart's symbol and timeframe, and on the type of account you hold. When the required historical date is available, the minimum number of chart bars is:

- 40,000 bars for the Ultimate plan.
- 30,000 bars for the Elite plan.
- 25,000 bars for the Expert plan.
- 20,000 bars for the Premium plan.
- 10,000 bars for Essential and Plus plans.
- 5000 bars for the Basic plan.

## Trade orders in backtesting

A maximum of 9000 orders can be placed when backtesting strategies. When using Deep Backtesting, the limit is 200,000.





- Get real OHLC price on a Heikin Ashi chart
- Get non-standard OHLC values on a standard chart
- Plot arrows on the chart
- Plot a dynamic horizontal line
- Plot a vertical line on condition
- Access the previous value
- Get a 5-days high
- Count bars in a dataset
- Enumerate bars in a day
- Find the highest and lowest values for the entire dataset
- Query the last non-na value

## 6.1 Get real OHLC price on a Heikin Ashi chart

Suppose, we have a Heikin Ashi chart (or Renko, Kagi, PriceBreak etc) and we've added a Pine script on it:

```

1 // @version=5
2 indicator("Visible OHLC", overlay=true)
3 c = close
4 plot(c)
```

You may see that variable `c` is a Heikin Ashi *close* price which is not the same as real OHLC price. Because `close` built-in variable is always a value that corresponds to a visible bar (or candle) on the chart.

So, how do we get the real OHLC prices in Pine Script™ code, if current chart type is non-standard? We should use `request.security` function in combination with `ticker.new` function. Here is an example:

```

1 // @version=5
2 indicator("Real OHLC", overlay = true)
3 t = ticker.new(syminfo.prefix, syminfo.ticker)
4 realC = request.security(t, timeframe.period, close)
5 plot(realC)
```

In a similar way we may get other OHLC prices: *open*, *high* and *low*.

## 6.2 Get non-standard OHLC values on a standard chart

Backtesting on non-standard chart types (e.g. Heikin Ashi or Renko) is not recommended because the bars on these kinds of charts do not represent real price movement that you would encounter while trading. If you want your strategy to enter and exit on real prices but still use Heikin Ashi-based signals, you can use the same method to get Heikin Ashi values on a regular candlestick chart:

```

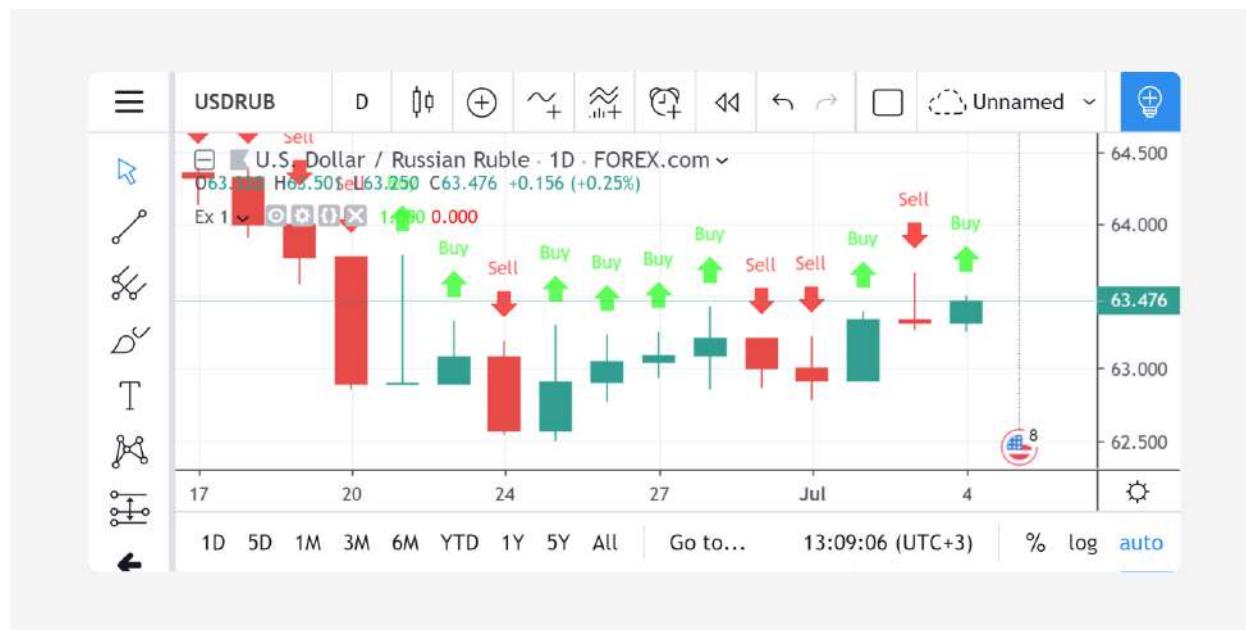
1 //@version=5
2 strategy("BarUpDn Strategy", overlay = true, default_qty_type = strategy.percent_of_
3   ↪equity, default_qty_value = 10)
4 maxIdLossPcntInput = input.float(1, "Max Intraday Loss (%)")
5 strategy.risk.max_intraday_loss(maxIdLossPcntInput, strategy.percent_of_equity)
6 needTrade() => close > open and open > close[1] ? 1 : close < open and open <_
7   ↪close[1] ? -1 : 0
8 trade = request.security(ticker.heikinashi(syminfo.tickerid), timeframe.period,_
9   ↪needTrade())
10 if trade == 1
11   strategy.entry("BarUp", strategy.long)
12 if trade == -1
13   strategy.entry("BarDn", strategy.short)
```

## 6.3 Plot arrows on the chart

You may use plotshape with style shape.arrowup and shape.arrowdown:

```

1 //@version=5
2 indicator('Ex 1', overlay = true)
3 condition = close >= open
4 plotshape(condition, color = color.lime, style = shape.arrowup, text = "Buy")
5 plotshape(not condition, color = color.red, style = shape.arrowdown, text = "Sell")
```

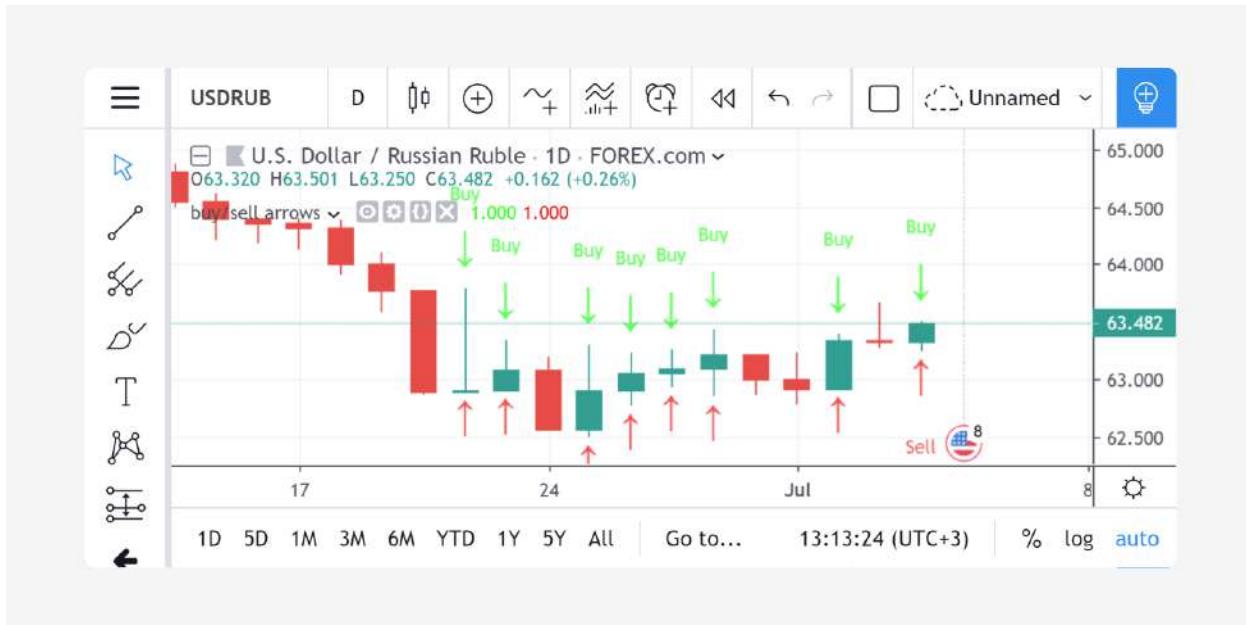


You may use the plotchar function with any unicode character:

```

1 //@version=5
2 indicator('buy/sell arrows', overlay = true)
3 condition = close >= open
4 plotchar(not condition, char='↓', color = color.lime, text = "Buy")
5 plotchar(condition, char='↑', location = location.belowbar, color = color.red, text =
  ↪"Sell")

```



## 6.4 Plot a dynamic horizontal line

There is the function `hline` in Pine Script™, but it is limited to only plot a constant value. Here is a simple script with a workaround to plot a changing `hline`:

```

1 //@version=5
2 indicator("Horizontal line", overlay = true)
3 plot(close[10], trackprice = true, offset = -9999)
4 // `trackprice = true` plots horizontal line on close[10]
5 // `offset = -9999` hides the plot
6 plot(close, color = #FFFFFF) // forces display

```

## 6.5 Plot a vertical line on condition

```

1 //@version=5
2 indicator("Vertical line", overlay = true, scale = scale.none)
3 // scale.none means do not resize the chart to fit this plot
4 // if the bar being evaluated is the last baron the chart (the most recent bar), then
  ↪cond is true
5 cond = barstate.islast
6 // when cond is true, plot a histogram with a line with height value of 100,000,000,
  ↪000,000,000,000.00
7 // (10 to the power of 20)

```

(continues on next page)

(continued from previous page)

```

8 // when cond is false, plot no numeric value (nothing is plotted)
9 // use the style of histogram, a vertical bar
10 plot(cond ? 10e20 : na, style = plot.style_histogram)

```

## 6.6 Access the previous value

```

1 //@version=5
2 ...
3 s = 0.0
4 s := nz(s[1]) // Accessing previous values
5 if (condition)
6     s := s + 1

```

## 6.7 Get a 5-days high

Lookback 5 days from the current bar, find the highest bar, plot a star character at that price level above the current bar



```

1 //@version=5
2 indicator("High of last 5 days", overlay = true)
3
4 // Milliseconds in 5 days: millisecs * secs * mins * hours * days
5 MS_IN_5DAYS = 1000 * 60 * 60 * 24 * 5
6
7 // The range check begins 5 days from the current time.
8 leftBorder = timenow - time < MS_IN_5DAYS
9 // The range ends on the last bar of the chart.
10 rightBorder = barstate.islast
11
12 // ----- Keep track of highest `high` during the range.

```

(continues on next page)

(continued from previous page)

```

13 // Initialize `maxHi` with `var` on bar zero only.
14 // This way, its value is preserved, bar to bar.
15 var float maxHi = na
16 if leftBorder
17     if not leftBorder[1]
18         // Range's first bar.
19         maxHi := high
20     else if not rightBorder
21         // On other bars in the range, track highest `high`.
22         maxHi := math.max(maxHi, high)
23
24 // Plot level of the highest `high` on the last bar.
25 plotchar(rightBorder ? maxHi : na, "Level", "-", location.absolute, size = size,
26           ↵normal)
26 // When in range, color the background.
27 bgcolor(leftBorder and not rightBorder ? color.new(color.aqua, 70) : na)

```

## 6.8 Count bars in a dataset

Get a count of all the bars in the loaded dataset. Might be useful for calculating flexible lookback periods based on number of bars.

```

1 //@version=5
2 indicator("Bar Count", overlay = true, scale = scale.none)
3 plot(bar_index + 1, style = plot.style_histogram)

```

## 6.9 Enumerate bars in a day

```

1 //@version=5
2 indicator("My Script", overlay = true, scale = scale.none)
3
4 isNewDay() =>
5     d = dayofweek
6     na(d[1]) or d != d[1]
7
8 plot(ta.barssince(isNewDay()), style = plot.style_cross)

```

## 6.10 Find the highest and lowest values for the entire dataset

```

1 //@version=5
2 indicator("", "", true)
3
4 allTimetHi(source) =>
5     var atHi = source
6     atHi := math.max(atHi, source)
7
8 allTimetLo(source) =>
9     var atLo = source

```

(continues on next page)

(continued from previous page)

```
10 atLo := math.min(atLo, source)
11
12 plot(allTimetHi(close), "ATH", color.green)
13 plot(allTimetLo(close), "ATL", color.red)
```

## 6.11 Query the last non-na value

You can use the script below to avoid gaps in a series:

```
1 //@version=5
2 indicator("")
3 series = close >= open ? close : na
4 vw = fixnan(series)
5 plot(series, style = plot.style_linebr, color = color.red) // series has na values
6 plot(vw) // all na values are replaced with the last non-empty value
```



## ERROR MESSAGES

- *The if statement is too long*
- *Script requesting too many securities*
- *Script could not be translated from: null*
- *line 2: no viable alternative at character '\$'*
- *Mismatched input <...> expecting <??>*
- *Loop is too long (> 500 ms)*
- *Script has too many local variables*
- *Pine Script™ cannot determine the referencing length of a series. Try using max\_bars\_back in the indicator or strategy function*

### 7.1 The if statement is too long

This error occurs when the indented code inside an `if statement` is too large for the compiler. Because of how the compiler works, you won't receive a message telling you exactly how many lines of code you are over the limit. The only solution now is to break up your `if statement` into smaller parts (functions or smaller `if statements`). The example below shows a reasonably lengthy `if statement`; theoretically, this would throw line 4: `if statement is too long`:

To fix this code, you could move these lines into their own function:

### 7.2 Script requesting too many securities

The maximum number of securities in script is limited to 40. If you declare a variable as a `request.security` function call and then use that variable as input for other variables and calculations, it will not result in multiple `request.security` calls. But if you will declare a function that calls `request.security` — every call to this function will count as a `request.security` call.

It is not easy to say how many securities will be called looking at the source code. Following example have exactly 3 calls to `request.security` after compilation:

```
1 // @version=5
2 indicator("Securities count")
3 a = request.security(syminfo.tickerid, '42', close) // (1) first unique security call
4 b = request.security(syminfo.tickerid, '42', close) // same call as above, will not
```

(continues on next page)

(continued from previous page)

```

5   ↪produce new security call after optimizations
6
7 plot(a)
8 plot(a + 2)
9 plot(b)
10
11 sym(p) => // no security call on this line
12     request.security(syminfo.tickerid, p, close)
13 plot(sym('D')) // (2) one indirect call to security
14 plot(sym('W')) // (3) another indirect call to security
15 request.security(syminfo.tickerid, timeframe.period, open) // result of this line is
   ↪never used, and will be optimized out

```

## 7.3 Script could not be translated from: null

```
1 study($)
```

Usually this error occurs in version 1 Pine scripts, and means that code is incorrect. Pine Script™ of version 2 (and higher) is better at explaining errors of this kind. So you can try to switch to version 2 by adding a special attribute in the first line. You'll get line 2: no viable alternative at character '\$':

```
1 // @version=2
2 study($)
```

## 7.4 line 2: no viable alternative at character ‘\$’

This error message gives a hint on what is wrong. \$ stands in place of string with script title. For example:

```
1 // @version=2
2 study("title")
```

## 7.5 Mismatched input <...> expecting <???>

Same as no viable alternative, but it is known what should be at that place. Example:

line 3: mismatched input 'plot' expecting 'end of line without line continuation'

To fix this you should start line with plot on a new line without an indent:

## 7.6 Loop is too long (> 500 ms)

We limit the computation time of loop on every historical bar and realtime tick to protect our servers from infinite or very long loops. This limit also fail-fast indicators that will take too long to compute. For example, if you'll have 5000 bars, and indicator takes 500 milliseconds to compute on each of bars, it would have result in more than 16 minutes of loading:

```

1 // @version=5
2 indicator("Loop is too long", max_bars_back = 101)
3 s = 0
4 for i = 1 to 1e3 // to make it longer
5     for j = 0 to 100
6         if timestamp(2017, 02, 23, 00, 00) <= time[j] and time[j] < timestamp(2017,
7             ↵02, 23, 23, 59)
8             s := s + 1
9 plot(s)

```

It might be possible to optimize algorithm to overcome this error. In this case, algorithm may be optimized like this:

```

1 // @version=5
2 indicator("Loop is too long", max_bars_back = 101)
3 bar_back_at(t) =>
4     i = 0
5     step = 51
6     for j = 1 to 100
7         if i < 0
8             i := 0
9             break
10        if step == 0
11            break
12        if time[i] >= t
13            i := i + step
14            i
15        else
16            i := i - step
17            i
18        step := step / 2
19        step
20        i
21
22 s = 0
23 for i = 1 to 1e3 // to make it longer
24     s := s - bar_back_at(timestamp(2017, 02, 23, 23, 59)) +
25         bar_back_at(timestamp(2017, 02, 23, 00, 00))
26     s
27 plot(s)

```

## 7.7 Script has too many local variables

This error appears if the script is too large to be compiled. A statement `var=expression` creates a local variable for `var`. Apart from this, it is important to note, that auxiliary variables can be implicitly created during the process of a script compilation. The limit applies to variables created both explicitly and implicitly. The limitation of 1000 variables is applied to each function individually. In fact, the code placed in a *global* scope of a script also implicitly wrapped up into the main function and the limit of 1000 variables becomes applicable to it. There are few refactorings you can try to avoid this issue:

```
1 var1 = expr1
2 var2 = expr2
3 var3 = var1 + var2
```

can be converted into:

```
1 var3 = expr1 + expr2
```

## 7.8 Pine Script™ cannot determine the referencing length of a series. Try using max\_bars\_back in the indicator or strategy function

The error appears in cases where Pine Script™ wrongly autodetects the required maximum length of series used in a script. This happens when a script's flow of execution does not allow Pine Script™ to inspect the use of series in branches of conditional statements (`if`, `iff` or `?`), and Pine Script™ cannot automatically detect how far back the series is referenced. Here is an example of a script causing this problem:

```
1 //>@version=5
2 indicator("Requires max_bars_back")
3 test = 0.0
4 if bar_index > 1000
5     test := ta.roc(close, 20)
6 plot(test)
```

In order to help Pine Script™ with detection, you should add the `max_bars_back` parameter to the script's indicator or strategy function:

```
1 //>@version=5
2 indicator("Requires max_bars_back", max_bars_back = 20)
3 test = 0.0
4 if bar_index > 1000
5     test := ta.roc(close, 20)
6 plot(test)
```

You may also resolve the issue by taking the problematic expression out of the conditional branch, in which case the `max_bars_back` parameter is not required:

```
1 //>@version=5
2 indicator("My Script")
3 test = 0.0
4 roc20 = ta.roc(close, 20)
5 if bar_index > 1000
6     test := roc20
7 plot(test)
```

In cases where the problem is caused by a **variable** rather than a built-in **function** (vwma in our example), you may use the `max_bars_back` function to explicitly define the referencing length for that variable only. This has the advantage of requiring less runtime resources, but entails that you identify the problematic variable, e.g., variable `s` in the following example:

```

1 //@version=5
2 indicator("My Script")
3 f(off) =>
4     t = 0.0
5     s = close
6     if bar_index > 242
7         t := s[off]
8     t
9 plot(f(301))

```

This situation can be resolved using the `max_bars_back` **function** to define the referencing length of variable `s` only, rather than for all the script's variables:

```

1 //@version=5
2 indicator("My Script")
3 f(off) =>
4     t = 0.0
5     s = close
6     max_bars_back(s, 301)
7     if bar_index > 242
8         t := s[off]
9     t
10 plot(f(301))

```

When using drawings that refer to previous bars through `bar_index[n]` and `xloc = xloc.bar_index`, the time series received from this bar will be used to position the drawings on the time axis. Therefore, if it is impossible to determine the correct size of the buffer, this error may occur. To avoid this, you need to use `max_bars_back(time, n)`. This behavior is described in more detail in the section about [drawings](#).





## RELEASE NOTES

- [2024](#)
- [2023](#)
- [2022](#)
- [2021](#)
- [2020](#)
- [2019](#)
- [2018](#)
- [2017](#)
- [2016](#)
- [2015](#)
- [2014](#)
- [2013](#)

This page contains release notes of notable changes in Pine Script™.

## 8.1 2024

### 8.1.1 May 2024

The `strategy.*` namespace features several new built-in variables:

- `strategy.avg_trade` - Returns the average amount of money gained or lost per trade. Calculated as the sum of all profits and losses divided by the number of closed trades.
- `strategy.avg_trade_percent` - Returns the average percentage gain or loss per trade. Calculated as the sum of all profit and loss percentages divided by the number of closed trades.
- `strategy.avg_winning_trade` - Returns the average amount of money gained per winning trade. Calculated as the sum of profits divided by the number of winning trades.
- `strategy.avg_winning_trade_percent` - Returns the average percentage gain per winning trade. Calculated as the sum of profit percentages divided by the number of winning trades.
- `strategy.avg_losing_trade` - Returns the average amount of money lost per losing trade. Calculated as the sum of losses divided by the number of losing trades.

- `strategy.avg_losing_trade_percent` - Returns the average percentage loss per losing trade. Calculated as the sum of loss percentages divided by the number of losing trades.

### Pine Profiler

Our new *Pine Profiler* is a powerful utility that analyzes the executions of all significant code in a script and displays helpful performance information next to the code lines *inside* the Pine Editor. The *Profiler*'s information provides insight into a script's runtime, the distribution of runtime across significant code regions, and the number of times each code region executes. With these insights, programmers can effectively pinpoint performance *bottlenecks* and ensure they focus on *optimizing* their code where it truly matters when they need to improve execution speeds.

See the new *Profiling and optimization* page to learn more about the Profiler, how it works, and how to use it to analyze a script's performance and identify optimization opportunities.

### 8.1.2 April 2024

We've added a new parameter to the `plot()`, `plotchar()`, `plotcandle()`, `plotbar()`, `plotarrow()`, `plotshape()`, and `bgcolor()` functions:

- `force_overlay` - If true, the output will display on the main chart pane, even when the script occupies a separate pane.

### 8.1.3 March 2024

The `syminfo.*` namespace features a new built-in variable:

- `syminfo.expiration_date` - On non-continuous futures symbols, returns a UNIX timestamp representing the start of the last day of the current contract.

The `time()` and `time_close()` functions have a new parameter:

- `bars_back` - If specified, the function will calculate the timestamp from the bar N bars back relative to the current bar on its timeframe. It can also calculate the expected time of a future bar up to 500 bars away if the argument is a negative value. Optional. The default is 0.

### 8.1.4 February 2024

We've added two new functions for working with strings:

- `str.repeat()` - Constructs a new string containing the source string repeated a specified number of times with a separator injected between each repeated instance.
- `str.trim()` - Constructs a new string with all consecutive whitespaces and other control characters removed from the left and right of the source string.

The `request.financial()` function now accepts "D" as a `period` argument, allowing scripts to request available daily financial data.

For example:

```
1 //@version=5
2 indicator("Daily financial data demo")
3
4 //@variable The daily Premium/Discount to Net Asset Value for "AMEX:SPY"
```

(continues on next page)

(continued from previous page)

```

5 float f1 = request.financial("AMEX:SPY", "NAV", "D")
6 plot(f1)

```

The `strategy.*` namespace features a new variable for monitoring available capital in a strategy's simulation:

- `strategy.opentrades.capital_held` - Returns the capital amount currently held by open trades.

## 8.1.5 January 2024

The `syminfo.*` namespace features new built-in variables:

Syminfo:

- `syminfo.employees` - The number of employees the company has.
- `syminfo.shareholders` - The number of shareholders the company has.
- `syminfo.shares_outstanding_float` - The total number of shares outstanding a company has available, excluding any of its restricted shares.
- `syminfo.shares_outstanding_total` - The total number of shares outstanding a company has available, including restricted shares held by insiders, major shareholders, and employees.

Target price:

- `syminfo.target_price_average` - The average of the last yearly price targets for the symbol predicted by analysts.
- `syminfo.target_price_date` - The starting date of the last price target prediction for the current symbol.
- `syminfo.target_price_estimates` - The latest total number of price target predictions for the current symbol.
- `syminfo.target_price_high` - The last highest yearly price target for the symbol predicted by analysts.
- `syminfo.target_price_low` - The last lowest yearly price target for the symbol predicted by analysts.
- `syminfo.target_price_median` - The median of the last yearly price targets for the symbol predicted by analysts.

Recommendations:

- `syminfo.recommendations_buy` - The number of analysts who gave the current symbol a “Buy” rating.
- `syminfo.recommendations_buy_strong` - The number of analysts who gave the current symbol a “Strong Buy” rating.
- `syminfo.recommendations_date` - The starting date of the last set of recommendations for the current symbol.
- `syminfo.recommendations_hold` - The number of analysts who gave the current symbol a “Hold” rating.
- `syminfo.recommendations_total` - The total number of recommendations for the current symbol.
- `ssyminfo.recommendations_sell` - The number of analysts who gave the current symbol a “Sell” rating.
- `syminfo.recommendations_sell_strong` - The number of analysts who gave the current symbol a “Strong Sell” rating.

## 8.2 2023

### 8.2.1 December 2023

We've added `format` and `precision` parameters to all `plot*`() functions, allowing indicators and strategies to selectively apply formatting and decimal precision settings to plotted results in the chart pane's y-axis, the script's status line, and the Data Window. The arguments passed to these parameters supersede the values in the `indicator()` and `strategy()` functions. Both are optional. The defaults for these parameters are the same as the values specified in the script's declaration statement.

For example:

```
1 // @version=5
2 indicator("My script", format = format.percent, precision = 4)
3
4 plot(close, format = format.price)           // Price format with 4-digit precision.
5 plot(100 * bar_index / close, precision = 2) // Percent format with 2-digit precision.
```

### 8.2.2 November 2023

We've added the following variables and functions to the `strategy.*` namespace:

- `strategy.grossloss_percent` - The total gross loss value of all completed losing trades, expressed as a percentage of the initial capital.
- `strategy.grossprofit_percent` - The total gross profit value of all completed winning trades, expressed as a percentage of the initial capital.
- `strategy.max_runup_percent` - The maximum rise from a trough in the equity curve, expressed as a percentage of the trough value.
- `strategy.max_drawdown_percent` - The maximum drop from a peak in the equity curve, expressed as a percentage of the peak value.
- `strategy.netprofit_percent` - The total value of all completed trades, expressed as a percentage of the initial capital.
- `strategy.openprofit_percent` - The current unrealized profit or loss for all open positions, expressed as a percentage of realized equity.
- `strategy.closedtrades.max_drawdown_percent()` - Returns the maximum drawdown of the closed trade, i.e., the maximum possible loss during the trade, expressed as a percentage.
- `strategy.closedtrades.max_runup_percent()` - Returns the maximum run-up of the closed trade, i.e., the maximum possible profit during the trade, expressed as a percentage.
- `strategy.closedtrades.profit_percent()` - Returns the profit/loss value of the closed trade, expressed as a percentage. Losses are expressed as negative values.
- `strategy.opentrades.max_drawdown_percent()` - Returns the maximum drawdown of the open trade, i.e., the maximum possible loss during the trade, expressed as a percentage.
- `strategy.opentrades.max_runup_percent()` - Returns the maximum run-up of the open trade, i.e., the maximum possible profit during the trade, expressed as a percentage.
- `strategy.opentrades.profit_percent()` - Returns the profit/loss of the open trade, expressed as a percentage. Losses are expressed as negative values.

## 8.2.3 October 2023

### Pine Script™ Polylines

Polyline drawings are drawings that sequentially connect the coordinates from an `array` of up to 10,000 `chart points` using straight or `curved` line segments, allowing scripts to draw custom formations that are difficult or impossible to achieve using `line` or `box` objects. To learn more about this new drawing type, see the [Polyline](#) section of our User Manual's page on [Lines and boxes](#).

## 8.2.4 September 2023

New functions were added:

- `strategy.default_entry_qty()` - Calculates the default quantity, in units, of an entry order from `strategy.entry()` or `strategy.order()` if it were to fill at the specified `fill_price` value.
- `chart.point.new()` - Creates a new `chart.point` object with the specified `time`, `index`, and `price`.
- `request.seed()` - Requests data from a user-maintained GitHub repository and returns it as a series. An in-depth tutorial on how to add new data can be found [here](#).
- `ticker.inherit()` - Constructs a ticker ID for the specified `symbol` with additional parameters inherited from the ticker ID passed into the function call, allowing the script to request a symbol's data using the same modifiers that the `from_tickerid` has, including extended session, dividend adjustment, currency conversion, non-standard chart types, back-adjustment, settlement-as-close, etc.
- `timeframe.from_seconds()` - Converts a specified number of `seconds` into a valid timeframe string based on our [timeframe specification format](#).

The `dividends.*` namespace now includes variables for retrieving future dividend information:

- `dividends.future_amount` - Returns the payment amount of the upcoming dividend in the currency of the current instrument, or `na` if this data isn't available.
- `dividends.future_ex_date` - Returns the Ex-dividend date (Ex-date) of the current instrument's next dividend payment, or `na` if this data isn't available.
- `dividends.future_pay_date` - Returns the Payment date (Pay date) of the current instrument's next dividend payment, or `na` if this data isn't available.

The `request.security_lower_tf()` function has a new parameter:

- `ignore_invalid_timeframe` - Determines how the function behaves when the chart's timeframe is smaller than the `timeframe` value in the function call. If `false`, the function will raise a runtime error and halt the script's execution. If `true`, the function will return `na` without raising an error.

Users can now explicitly declare variables with the `const`, `simple`, and `series` type qualifiers, allowing more precise control over the types of variables in their scripts. For example:

## 8.2.5 August 2023

Added the following alert placeholders:

- `{syminfo.currency}` - Returns the currency code of the current symbol (“EUR”, “USD”, etc.).
- `{syminfo.basecurrency}` - Returns the base currency code of the current symbol if the symbol refers to a currency pair. Otherwise, it returns `na`. For example, it returns “EUR” when the symbol is “EURUSD”.

## Pine Script™ Maps

Maps are collections that hold elements in the form of *key-value pairs*. They associate unique keys of a *fundamental type* with values of a *built-in* or *user-defined* type. Unlike *arrays* and *matrices*, these collections are *unordered* and do not utilize an internal lookup index. Instead, scripts access the values of maps by referencing the *keys* from the key-value pairs put into them. For more information on these new collections, see our [User Manual's page on Maps](#).

## 8.2.6 July 2023

Fixed an issue that caused strategies to occasionally calculate the sizes of limit orders incorrectly due to improper tick rounding of the `limit` price.

Added a new built-in variable to the `strategy.*` namespace:

- `strategy.margin_liquidation_price` - When a strategy uses margin, returns the price value after which a margin call will occur.

## 8.2.7 June 2023

New `syminfo.*` built-in variables were added:

- `syminfo.sector` - Returns the sector of the symbol.
- `syminfo.industry` - Returns the industry of the symbol.
- `syminfo.country` - Returns the two-letter code of the country where the symbol is traded.

## 8.2.8 May 2023

New parameter added to the `strategy.entry()`, `strategy.order()`, `strategy.close()`, `strategy.close_all()`, and `strategy.exit()` functions:

- `disable_alert` - Disables order fill alerts for any orders placed by the function.

Our “Indicator on indicator” feature, which allows a script to pass another indicator’s plot as a source value via the `input.source()` function, now supports multiple external inputs. Scripts can use a multitude of external inputs originating from up to 10 different indicators.

We’ve added the following array functions:

- `array.every()` - Returns `true` if all elements of the `id` array are `true`, `false` otherwise.
- `array.some()` - Returns `true` if at least one element of the `id` array is `true`, `false` otherwise.

These functions also work with arrays of `int` and `float` types, in which case zero values are considered `false`, and all others `true`.

## 8.2.9 April 2023

Fixed an issue with trailing stops in `strategy.exit()` being filled on high/low prices rather than on intrabar prices.

Fixed behavior of `array.mode()`, `matrix.mode()` and `ta.mode()`. Now these functions will return the smallest value when the data has no most frequent value.

## 8.2.10 March 2023

It is now possible to use seconds-based timeframe strings for the `timeframe` parameter in `request.security()` and `request.security_lower_tf()`.

A new function was added:

- `request.currency_rate()` - provides a daily rate to convert a value expressed in the `from` currency to another in the `to` currency.

## 8.2.11 February 2023

### Pine Script™ Methods

Pine Script™ methods are specialized functions associated with specific instances of built-in or user-defined types. They offer a more convenient syntax than standard functions, as users can access methods in the same way as object fields using the handy dot notation syntax. Pine Script™ includes built-in methods for `array`, `matrix`, `line`, `linefill`, `label`, `box`, and `table` types and facilitates user-defined methods with the new `method` keyword. For more details on this new feature, see our [User Manual's page on methods](#).

## 8.2.12 January 2023

New array functions were added:

- `array.first()` - Returns the array's first element.
- `array.last()` - Returns the array's last element.

## 8.3 2022

### 8.3.1 December 2022

#### Pine Objects

Pine objects are instantiations of the new user-defined composite types (UDTs) declared using the `type` keyword. Experienced programmers can think of UDTs as method-less classes. They allow users to create custom types that organize different values under one logical entity. A detailed rundown of the new functionality can be found in our [User Manual's page on objects](#).

A new function was added:

- `ticker.standard()` - Creates a ticker to request data from a standard chart that is unaffected by modifiers like extended session, dividend adjustment, currency conversion, and the calculations of non-standard chart types: Heikin Ashi, Renko, etc.

New `strategy.*` functions were added:

- `strategy.opentrades.entry_comment()` - The function returns the comment message of the open trade's entry.
- `strategy.closedtrades.entry_comment()` - The function returns the comment message of the closed trade's entry.
- `strategy.closedtrades.exit_comment()` - The function returns the comment message of the closed trade's exit.

### 8.3.2 November 2022

Fixed behaviour of `math.round_to_mintick()` function. For 'na' values it returns 'na'.

### 8.3.3 October 2022

Pine Script™ now has a new, more powerful and better-integrated editor. Read [our blog](#) to find out everything to know about all the new features and upgrades.

New overload for the `fill()` function was added. Now it can create vertical gradients. More info about it in the [blog post](#).

A new function was added:

- `str.format_time()` - Converts a timestamp to a formatted string using the specified format and time zone.

### 8.3.4 September 2022

The `text_font_family` parameter now allows the selection of a monospace font in `label.new()`, `box.new()` and `table.cell()` function calls, which makes it easier to align text vertically. Its arguments can be:

- `font.family_default` - Specifies the default font.
- `font.family_monospace` - Specifies a monospace font.

The accompanying setter functions are:

- `label.set_text_font_family()` - The function sets the font family of the text inside the label.
- `box.set_text_font_family()` - The function sets the font family of the text inside the box.
- `table.cell_set_text_font_family()` - The function sets the font family of the text inside the cell.

### 8.3.5 August 2022

A new label style `label.style_text_outline` was added.

A new parameter for the `ta.pivot_point_levels()` function was added:

- `developing` - If `false`, the values are those calculated the last time the anchor condition was true. They remain constant until the anchor condition becomes true again. If `true`, the pivots are developing, i.e., they constantly recalculate on the data developing between the point of the last anchor (or bar zero if the anchor condition was never true) and the current bar. Cannot be `true` when `type` is set to "Woodie".

A new parameter for the `box.new()` function was added:

- `text_wrap` - It defines whether the text is presented in a single line, extending past the width of the box if necessary, or wrapped so every line is no wider than the box itself.

This parameter supports two arguments:

- `text.wrap_none` - Disabled wrapping mode for `box.new` and `box.set_text_wrap` functions.
- `text.wrap_auto` - Automatic wrapping mode for `box.new` and `box.set_text_wrap` functions.

New built-in functions were added:

- `ta.min()` - Returns the all-time low value of `source` from the beginning of the chart up to the current bar.
- `ta.max()` - Returns the all-time high value of `source` from the beginning of the chart up to the current bar.

A new annotation `//@strategy_alert_message` was added. If the annotation is added to the strategy, the text written after it will be automatically set as the default alert message in the *Create Alert* window.

```
1 // @version=5
2 // @strategy_alert_message My Default Alert Message
3 strategy("My Strategy")
4 plot(close)
```

### 8.3.6 July 2022

It is now possible to fine-tune where a script's plot values are displayed through the introduction of new arguments for the `display` parameter of the `plot()`, `plotchar()`, `plotshape()`, `plotarrow()`, `plotcandle()`, and `plotbar()` functions.

Four new arguments were added, complementing the previously available `display.all` and `display.none`:

- `display.data_window` displays the plot values in the Data Window, one of the items available from the chart's right sidebar.
- `display.pane` displays the plot in the pane where the script resides, as defined in with the `overlay` parameter of the script's `indicator()`, `strategy()`, or `library()` declaration statement.
- `display.price_scale` controls the display of the plot's label and price in the price scale, if the chart's settings allow them.
- `display.status_line` displays the plot values in the script's status line, next to the script's name on the chart, if the chart's settings allow them.

The `display` parameter supports the addition and subtraction of its arguments:

- `display.all - display.status_line` will display the plot's information everywhere except in the script's status line.
- `display.price_scale + display.status_line` will display the plot in the price scale and status line only.

### 8.3.7 June 2022

The behavior of the argument used with the `qty_percent` parameter of `strategy.exit()` has changed. Previously, the percentages used on successive exit orders of the same position were calculated from the remaining position at any given time. Instead, the percentages now always apply to the initial position size. When executing the following strategy, for example:

```
1 // @version=5
2 strategy("strategy.exit() example", overlay = true)
3 strategy.entry("Long", strategy.long, qty = 100)
4 strategy.exit("Exit Long1", "Long", trail_points = 50, trail_offset = 0, qty_percent ←
← 20)
5 strategy.exit("Exit Long2", "Long", trail_points = 100, trail_offset = 0, qty_percent ←
← 20)
```

20% of the initial position will be closed on each `strategy.exit()` call. Before, the first call would exit 20% of the initial position, and the second would exit 20% of the remaining 80% of the position, so only 16% of the initial position.

Two new parameters for the built-in `ta.vwap()` function were added:

- `anchor` - Specifies the condition that triggers the reset of VWAP calculations. When `true`, calculations reset; when `false`, calculations proceed using the values accumulated since the previous reset.
- `stdev_mult` - If specified, the `ta.vwap()` calculates the standard deviation bands based on the main VWAP series and returns a `[vwap, upper_band, lower_band]` tuple.

New overloaded versions of the `strategy.close()` and `strategy.close_all()` functions with the `immediately` parameter. When `immediately` is set to `true`, the closing order will be executed on the tick where it has been placed, ignoring the strategy parameters that restrict the order execution to the open of the next bar.

New built-in functions were added:

- `timeframe.change()` - Returns `true` on the first bar of a new timeframe, `false` otherwise.
- `ta.pivot_point_levels()` - Returns a float array with numerical values representing 11 pivot point levels: `[P, R1, S1, R2, S2, R3, S3, R4, S4, R5, S5]`. Levels absent from the specified type return `na` values.

New built-in variables were added:

- `session.isfirstbar` - returns `true` if the current bar is the first bar of the day's session, `false` otherwise.
- `session.islastbar` - returns `true` if the current bar is the last bar of the day's session, `false` otherwise.
- `session.isfirstbar_regular` - returns `true` on the first regular session bar of the day, `false` otherwise.
- `session.islastbar_regular` - returns `true` on the last regular session bar of the day, `false` otherwise.
- `chart.left_visible_bar_time` - returns the `time` of the leftmost bar currently visible on the chart.
- `chart.right_visible_bar_time` - returns the `time` of the rightmost bar currently visible on the chart.

### 8.3.8 May 2022

`Matrix` support has been added to the `request.security()` function.

The historical states of `arrays` and `matrices` can now be referenced with the `[]` operator. In the example below, we reference the historic state of a matrix 10 bars ago:

```
1 // @version=5
2 indicator("matrix.new<float> example")
3 m = matrix.new<float>(1, 1, close)
4 float x = na
5 if bar_index > 10
6     x := matrix.get(m[10], 0, 0)
7 plot(x)
8 plot(close)
```

The `ta.change()` function now can take values of `int` and `bool` types as its `source` parameter and return the difference in the respective type.

New built-in variables were added:

- `chart.bg_color` - Returns the color of the chart's background from the "Chart settings/Appearance/Background" field.
- `chart_fg_color` - Returns a color providing optimal contrast with `chart.bg_color`.
- `chart.is_standard` - Returns true if the chart type is bars, candles, hollow candles, line, area or baseline, false otherwise.
- `currency.USDT` - A constant for the Tether currency code.

New functions were added:

- `syminfo.prefix()` - returns the exchange prefix of the `symbol` passed to it, e.g. “NASDAQ” for “NASDAQ:AAPL”.
- `syminfo.ticker()` - returns the ticker of the `symbol` passed to it without the exchange prefix, e.g. “AAPL” for “NASDAQ:AAPL”.
- `request.security_lower_tf()` - requests data from a lower timeframe than the chart’s.

Added `use_bar_magnifier` parameter for the `strategy()` function. When `true`, the `Broker Emulator` uses lower timeframe data during history backtesting to achieve more realistic results.

Fixed behaviour of `strategy.exit()` function when stop loss triggered at prices outside the bars price range.

Added new `comment` and `alert` message parameters for the `strategy.exit()` function:

- `comment_profit` - additional notes on the order if the exit was triggered by crossing `profit` or `limit` specifically.
- `comment_loss` - additional notes on the order if the exit was triggered by crossing `stop` or `loss` specifically.
- `comment_trailing` - additional notes on the order if the exit was triggered by crossing `trail_offset` specifically.
- `alert_profit` - text that will replace the '`\{\{strategy.order.alert_message\}\}`' placeholder if the exit was triggered by crossing `profit` or `limit` specifically.
- `alert_loss` - text that will replace the '`\{\{strategy.order.alert_message\}\}`' placeholder if the exit was triggered by crossing `stop` or `loss` specifically.
- `alert_trailing` - text that will replace the '`\{\{strategy.order.alert_message\}\}`' placeholder if the exit was triggered by crossing `trail_offset` specifically.

### 8.3.9 April 2022

Added the `display` parameter to the following functions: `barcolor`, `bgcolor`, `fill`, `hline`.

A new function was added:

- `request.economic()` - Economic data includes information such as the state of a country’s economy or of a particular industry.

New built-in variables were added:

- `strategy.max_runup` - Returns the maximum equity run-up value for the whole trading interval.
- `syminfo.volumetype` - Returns the volume type of the current symbol.
- `chart.is_heikinashi` - Returns true if the chart type is Heikin Ashi, false otherwise.
- `chart.is_kagi` - Returns true if the chart type is Kagi, false otherwise.
- `chart.is_linebreak` - Returns true if the chart type is Line break, false otherwise.
- `chart.is_pnf` - Returns true if the chart type is Point & figure, false otherwise.
- `chart.is_range` - Returns true if the chart type is Range, false otherwise.
- `chart.is_renko` - Returns true if the chart type is Renko, false otherwise.

New matrix functions were added:

- `matrix.new<type>` - Creates a new matrix object. A matrix is a two-dimensional data structure containing rows and columns. All elements in the matrix must be of the type specified in the type template (“`<type>`”).
- `matrix.row()` - Creates a one-dimensional array from the elements of a matrix row.

- `matrix.col()` - Creates a one-dimensional array from the elements of a matrix column.
- `matrix.get()` - Returns the element with the specified index of the matrix.
- `matrix.set()` - Assigns value to the element at the `column` and `row` index of the matrix.
- `matrix.rows()` - Returns the number of rows in the matrix.
- `matrix.columns()` - Returns the number of columns in the matrix.
- `matrix.elements_count()` - Returns the total number of matrix elements.
- `matrix.add_row()` - Adds a row to the matrix. The row can consist of `na` values, or an array can be used to provide values.
- `matrix.add_col()` - Adds a column to the matrix. The column can consist of `na` values, or an array can be used to provide values.
- `matrix.remove_row()` - Removes the row of the matrix and returns an array containing the removed row's values.
- `matrix.remove_col()` - Removes the column of the matrix and returns an array containing the removed column's values.
- `matrix.swap_rows()` - Swaps the rows in the matrix.
- `matrix.swap_columns()` - Swaps the columns in the matrix.
- `matrix.fill()` - Fills a rectangular area of the matrix defined by the indices `from_column` to `to_column`.
- `matrix.copy()` - Creates a new matrix which is a copy of the original.
- `matrix.submatrix()` - Extracts a submatrix within the specified indices.
- `matrix.reverse()` - Reverses the order of rows and columns in the matrix. The first row and first column become the last, and the last become the first.
- `matrix.reshape()` - Rebuilds the matrix to `rows x cols` dimensions.
- `matrix.concat()` - Append one matrix to another.
- `matrix.sum()` - Returns a new matrix resulting from the sum of two matrices, or of a matrix and a scalar (a numerical value).
- `matrix.diff()` - Returns a new matrix resulting from the subtraction between matrices, or of matrix and a scalar (a numerical value).
- `matrix.mult()` - Returns a new matrix resulting from the product between the matrices, or between a matrix and a scalar (a numerical value), or between a matrix and a vector (an array of values).
- `matrix.sort()` - Rearranges the rows in the `id` matrix following the sorted order of the values in the `column`.
- `matrix.avg()` - Calculates the average of all elements in the matrix.
- `matrix.max()` - Returns the largest value from the matrix elements.
- `matrix.min()` - Returns the smallest value from the matrix elements.
- `matrix.median()` - Calculates the median ("the middle" value) of matrix elements.
- `matrix.mode()` - Calculates the mode of the matrix, which is the most frequently occurring value from the matrix elements. When there are multiple values occurring equally frequently, the function returns the smallest of those values.
- `matrix.pow()` - Calculates the product of the matrix by itself `power` times.
- `matrix.det()` - Returns the determinant of a square matrix.

- `matrix.transpose()` - Creates a new, transposed version of the matrix by interchanging the row and column index of each element.
- `matrix.pinv()` - Returns the pseudoinverse of a matrix.
- `matrix.inv()` - Returns the inverse of a square matrix.
- `matrix.rank()` - Calculates the rank of the matrix.
- `matrix.trace()` - Calculates the trace of a matrix (the sum of the main diagonal's elements).
- `matrix.eigenvalues()` - Returns an array containing the eigenvalues of a square matrix.
- `matrix.eigenvectors()` - Returns a matrix of eigenvectors, in which each column is an eigenvector of the matrix.
- `matrix.kron()` - Returns the Kronecker product for the two matrices.
- `matrix.is_zero()` - Determines if all elements of the matrix are zero.
- `matrix.is_identity()` - Determines if a matrix is an identity matrix (elements with ones on the main diagonal and zeros elsewhere).
- `matrix.is_binary()` - Determines if the matrix is binary (when all elements of the matrix are 0 or 1).
- `matrix.is_symmetric()` - Determines if a square matrix is symmetric (elements are symmetric with respect to the main diagonal).
- `matrix.is_antisymmetric()` - Determines if a matrix is antisymmetric (its transpose equals its negative).
- `matrix.is_diagonal()` - Determines if the matrix is diagonal (all elements outside the main diagonal are zero).
- `matrix.is_antidiagonal()` - Determines if the matrix is anti-diagonal (all elements outside the secondary diagonal are zero).
- `matrix.is_triangular()` - Determines if the matrix is triangular (if all elements above or below the main diagonal are zero).
- `matrix.is_stochastic()` - Determines if the matrix is stochastic.
- `matrix.is_square()` - Determines if the matrix is square (it has the same number of rows and columns).

Added a new parameter for the `strategy()` function:

- `risk_free_rate` - The risk-free rate of return is the annual percentage change in the value of an investment with minimal or zero risk, used to calculate the Sharpe and Sortino ratios.

### 8.3.10 March 2022

New array functions were added:

- `array.sort_indices()` - returns an array of indices which, when used to index the original array, will access its elements in their sorted order.
- `array.percentrank()` - returns the percentile rank of a value in the array.
- `array.percentile_nearest_rank()` - returns the value for which the specified percentage of array values (percentile) are less than or equal to it, using the nearest-rank method.
- `array.percentile_linear_interpolation()` - returns the value for which the specified percentage of array values (percentile) are less than or equal to it, using linear interpolation.
- `array.abs()` - returns an array containing the absolute value of each element in the original array.
- `array.binary_search()` - returns the index of the value, or -1 if the value is not found.

- `array.binary_search_leftmost()` - returns the index of the value if it is found or the index of the next smallest element to the left of where the value would lie if it was in the array.
- `array.binary_search_rightmost()` - returns the index of the value if it is found or the index of the element to the right of where the value would lie if it was in the array.

Added a new optional `nth` parameter for the `array.min()` and `array.max()` functions.

Added `index` in `for..in` operator. It tracks the current iteration's index.

### Table merging and cell tooltips

- It is now possible to merge several cells in a table. A merged cell doesn't have to be a header: you can merge cells in any direction, as long as the resulting cell doesn't affect any already merged cells and doesn't go outside of the table's bounds. Cells can be merged with the new `table.merge_cells()` function.
- Tables now support tooltips, floating labels that appear when you hover over a table's cell. To add a tooltip, pass a string to the `tooltip` argument of the `table.cell()` function or use the new `table.cell_set_tooltip()` function.

### 8.3.11 February 2022

Added templates and the ability to create arrays via templates. Instead of using one of the `array.new_*` () functions, a template function `array.new<type>` can be used. In the example below, we use this functionality to create an array filled with `float` values:

```
1 //@version=5
2 indicator("array.new<float> example")
3 length = 5
4 var a = array.new<float>(length, close)
5 if array.size(a) == length
6     array.remove(a, 0)
7     array.push(a, close)
8 plot(array.sum(a) / length, "SMA")
```

New functions were added:

- `timeframe.in_seconds(timeframe)` - converts the timeframe passed to the `timeframe` argument into seconds.
- `input.text_area()` - adds multiline text input area to the Script settings.
- `strategy.closedtrades.entry_id()` - returns the id of the closed trade's entry.
- `strategy.closedtrades.exit_id()` - returns the id of the closed trade's exit.
- `strategy.opentrades.entry_id()` - returns the id of the open trade's entry.

### 8.3.12 January 2022

Added new functions to clone drawings:

- `line.copy()`
- `label.copy()`
- `box.copy()`

## 8.4 2021

### 8.4.1 December 2021

#### Linefills

The space between lines drawn in Pine Script™ can now be filled! We've added a new `linefill` drawing type, along with a number of functions dedicated to manipulating it. Linefills are created by passing two lines and a color to the `linefill.new()` function, and their behavior is based on the lines they're tied to: they extend in the same direction as the lines, move when their lines move, and are deleted when one of the two lines is deleted.

New linefill-related functions:

- `array.new_linefill()`
- `linefill()`
- `linefill.delete()`
- `linefill.get_line1()`
- `linefill.get_line2()`
- `linefill.new()`
- `linefill.set_color()`
- `linefill.all()`

#### New functions for string manipulation

Added a number of new functions that provide more ways to process strings, and introduce regular expressions to Pine Script™:

- `str.contains(source, str)` - Determines if the `source` string contains the `str` substring.
- `str.pos(source, str)` - Returns the position of the `str` string in the `source` string.
- `str.substring(source, begin_pos, end_pos)` - Extracts a substring from the `source` string.
- `str.replace(source, target, replacement, occurrence)` - Contrary to the existing `str.replace_all()` function, `str.replace()` allows the selective replacement of a matched substring with a replacement string.
- `str.lower(source)` and `str.upper(source)` - Convert all letters of the `source` string to lower or upper case.
- `str.startswith(source, str)` and `str.endswith(source, str)` - Determine if the `source` string starts or ends with the `str` substring.
- `str.match(source, regex)` - Extracts the substring matching the specified regular expression.

## Textboxes

Box drawings now supports text. The `box.new()` function has five new parameters for text manipulation: `text`, `text_size`, `text_color`, `text_valign`, and `text_halign`. Additionally, five new functions to set the text properties of existing boxes were added:

- `box.set_text()`
- `box.set_text_color()`
- `box.set_text_size()`
- `box.set_text_valign()`
- `box.set_text_halign()`

## New built-in variables

Added new built-in variables that return the `bar_index` and `time` values of the last bar in the dataset. Their values are known at the beginning of the script's calculation:

- `last_bar_index` - Bar index of the last chart bar.
- `last_bar_time` - UNIX time of the last chart bar.

New built-in source variable:

- `hlcc4` - A shortcut for `(high + low + close + close) / 4`. It averages the high and low values with the double-weighted close.

## 8.4.2 November 2021

### for...in

Added a new `for...in` operator to iterate over all elements of an array:

```
1 //@version=5
2 indicator("My Script")
3 int[] a1 = array.from(1, 3, 6, 3, 8, 0, -9, 5)
4
5 highest(array) =>
6     var int highestNum = na
7     for item in array
8         if na(highestNum) or item > highestNum
9             highestNum := item
10            highestNum
11
12 plot(highest(a1))
```

## Function overloads

Added function overloads. Several functions in a script can now share the same name, as long one of the following conditions is true:

- Each overload has a different number of parameters:

```

1 // @version=5
2 indicator("Function overload")
3
4 // Two parameters
5 mult(x1, x2) =>
6     x1 * x2
7
8 // Three parameters
9 mult(x1, x2, x3) =>
10    x1 * x2 * x3
11
12 plot(mult(7, 4))
13 plot(mult(7, 4, 2))

```

- When overloads have the same number of parameters, all parameters in each overload must be explicitly typified, and their type combinations must be unique:

```

1 // @version=5
2 indicator("Function overload")
3
4 // Accepts both 'int' and 'float' values - any 'int' can be automatically cast to
5 // 'float'
6 mult(float x1, float x2) =>
7     x1 * x2
8
9 // Returns a 'bool' value instead of a number
10 mult(bool x1, bool x2) =>
11     x1 and x2 ? true : false
12
13 mult(string x1, string x2) =>
14     str.tonumber(x1) * str.tonumber(x2)
15
16 // Has three parameters, so explicit types are not required
17 mult(x1, x2, x3) =>
18     x1 * x2 * x3
19
20 plot(mult(7, 4))
21 plot(mult(7.5, 4.2))
22 plot(mult(true, false) ? 1 : 0)
23 plot(mult("5", "6"))
24 plot(mult(7, 4, 2))

```

### Currency conversion

Added a new `currency` argument to most `request.*()` functions. If specified, price values returned by the function will be converted from the source currency to the target currency. The following functions are affected:

- `request.dividends()`
- `request.earnings()`
- `request.financial()`
- `request.security()`

### 8.4.3 October 2021

Pine Script™ v5 is here! This is a list of the **new** features added to the language, and a few of the **changes** made. See the Pine Script™ v5 Migration guide for a complete list of the **changes** in v5.

#### New features

Libraries are a new type of publication. They allow you to create custom functions for reuse in other scripts. See this manual's page on [Libraries](#).

Pine Script™ now supports `switch` structures! They provide a more convenient and readable alternative to long ternary operators and `if` statements.

`while` loops are here! They allow you to create a loop that will only stop when its controlling condition is false, or a `break` command is used in the loop.

New built-in array variables are maintained by the Pine Script™ runtime to hold the IDs of all the active objects of the same type drawn by your script. They are `label.all`, `line.all`, `box.all` and `table.all`.

The `runtime.error()` function makes it possible to halt the execution of a script and display a runtime error with a custom message. You can use any condition in your script to trigger the call.

Parameter definitions in user-defined functions can now include a default value: a function defined as `f(x = 1) => x` will return 1 when called as `f()`, i.e., without providing an argument for its `x` parameter.

New variables and functions provide better script visibility on strategy information:

- `strategy.closedtrades.entry_price()` and `strategy.opentrades.entry_price()`
- `strategy.closedtrades.entry_bar_index()` and `strategy.opentrades.entry_bar_index()`
- `strategy.closedtrades.entry_time()` and `strategy.opentrades.entry_time()`
- `strategy.closedtrades.size()` and `strategy.opentrades.size()`
- `strategy.closedtrades.profit()` and `strategy.opentrades.profit()`
- `strategy.closedtrades.commission()` and `strategy.opentrades.commission()`
- `strategy.closedtrades.max_runup()` and `strategy.opentrades.max_runup()`
- `strategy.closedtrades.max_drawdown()` and `strategy.opentrades.max_drawdown()`
- `strategy.closedtrades.exit_price()`
- `strategy.closedtrades.exit_bar_index()`
- `strategy.closedtrades.exit_time()`
- `strategy.convert_to_account()`

- `strategy.convert_to_symbol()`
- `strategy.account_currency`

A new `earnings.standardized` constant for the `request.earnings()` function allows requesting standardized earnings data.

A v4 to v5 converter is now included in the Pine Script™ Editor. See the Pine Script™ v5 Migration guide for more information on converting your scripts to v5.

The [Reference Manual](#) now includes the systematic mention of the form and type (e.g., “simple int”) required for each function parameter.

The *User Manual* was reorganized and new content was added.

## Changes

Many built-in variables, functions and function arguments were renamed or moved to new namespaces in v5. The venerable `study()`, for example, is now `indicator()`, and `security()` is now `request.security()`. New namespaces now group related functions and variables together. This consolidation implements a more rational nomenclature and provides an orderly space to accommodate the many additions planned for Pine Script™.

See the Pine Script™ v5 Migration guide for a complete list of the **changes** made in v5.

### 8.4.4 September 2021

New parameter has been added for the `dividends()`, `earnings()`, `financial()`, `quandl()`, `security()`, and `splits()` functions:

- `ignore_invalid_symbol` - determines the behavior of the function if the specified symbol is not found: if `false`, the script will halt and return a runtime error; if `true`, the function will return `na` and execution will continue.

### 8.4.5 July 2021

`tostring` now accepts “bool” and “string” types.

New argument for `time` and `time_close` functions was added:

- `timezone` - timezone of the `session` argument, can only be used when a session is specified. Can be written out in GMT notation (e.g. “GMT-5”) or as an IANA time zone database name (e.g. “America/New\_York”).

It is now possible to place a drawing object in the future with `xloc = xloc.bar_index`.

New argument for `study` and `strategy` functions was added:

- `explicit_plot_zorder` - specifies the order in which the indicator’s plots, fills, and hlines are rendered. If `true`, the plots will be drawn based on the order in which they appear in the indicator’s code, each newer plot being drawn above the previous ones.

## 8.4.6 June 2021

New variable was added:

- `barstate.islastconfirmedhistory` - returns `true` if script is executing on the dataset's last bar when market is closed, or script is executing on the bar immediately preceding the real-time bar, if market is open. Returns `false` otherwise.

New function was added:

- `round_to_mintick(x)` - returns the value rounded to the symbol's mintick, i.e. the nearest value that can be divided by `syminfo.mintick`, without the remainder, with ties rounding up.

Expanded `tostring()` functionality. The function now accepts three new formatting arguments:

- `format.mintick` to format to tick precision.
- `format.volume` to abbreviate large values.
- `format.percent` to format percentages.

## 8.4.7 May 2021

Improved backtesting functionality by adding the Leverage mechanism.

Added support for table drawings and functions for working with them. Tables are unique objects that are not anchored to specific bars; they float in a script's space, independently of the chart bars being viewed or the zoom factor used. For more information, see the [Tables](#) User Manual page.

New functions were added:

- `color.rgb(red, green, blue, transp)` - creates a new color with transparency using the RGB color model.
- `color.from_gradient(value, bottom_value, top_value, bottom_color, top_color)` - returns color calculated from the linear gradient between `bottom_color` to `top_color`.
- `color.r(color), color.g(color), color.b(color), color.t(color)` - retrieves the value of one of the color components.
- `array.from()` - takes a variable number of arguments with one of the types: `int, float, bool, string, label, line, color, box, table` and returns an array of the corresponding type.

A new box drawing has been added to Pine Script™, making it possible to draw rectangles on charts using the Pine Script™ syntax. For more details see the Pine Script™ [reference](#) and the [Lines and boxes](#) User Manual page.

The `color.new` function can now accept series and input arguments, in which case, the colors will be calculated at runtime. For more information about this, see our [Colors](#) User Manual page.

## 8.4.8 April 2021

New math constants were added:

- `math.pi` - is a named constant for Archimedes' constant. It is equal to 3.1415926535897932.
- `math.phi` - is a named constant for the golden ratio. It is equal to 1.6180339887498948.
- `math.rphi` - is a named constant for the golden ratio conjugate. It is equal to 0.6180339887498948.
- `math.e` - is a named constant for Euler's number. It is equal to 2.7182818284590452.

New math functions were added:

- `round(x, precision)` - returns the value of `x` rounded to the nearest integer, with ties rounding up. If the `precision` parameter is used, returns a float value rounded to that number of decimal places.
- `median(source, length)` - returns the median of the series.
- `mode(source, length)` - returns the mode of the series. If there are several values with the same frequency, it returns the smallest value.
- `range(source, length)` - returns the difference between the `min` and `max` values in a series.
- `todegrees(radians)` - returns an approximately equivalent angle in degrees from an angle measured in radians.
- `toradians(degrees)` - returns an approximately equivalent angle in radians from an angle measured in degrees.
- `random(min, max, seed)` - returns a pseudo-random value. The function will generate a different sequence of values for each script execution. Using the same value for the optional seed argument will produce a repeatable sequence.

New functions were added:

- `session.ismarket` - returns `true` if the current bar is a part of the regular trading hours (i.e. market hours), `false` otherwise.
- `session.ispremarket` - returns `true` if the current bar is a part of the pre-market, `false` otherwise.
- `session.ispostmarket` - returns `true` if the current bar is a part of the post-market, `false` otherwise.
- `str.format` - converts the values to strings based on the specified formats. Accepts certain number modifiers: `integer`, `currency`, `percent`.

## 8.4.9 March 2021

New assignment operators were added:

- `+=` - addition assignment
- `-=` - subtraction assignment
- `*=` - multiplication assignment
- `/=` - division assignment
- `%=` - modulus assignment

New parameters for inputs customization were added:

- `inline` - combines all the input calls with the same inline value in one line.
- `group` - creates a header above all inputs that use the same group string value. The string is also used as the header text.
- `tooltip` - adds a tooltip icon to the `Inputs` menu. The tooltip string is shown when hovering over the tooltip icon.

New argument for `fill` function was added:

- `fillgaps` - controls whether fills continue on gaps when one of the `plot` calls returns an `na` value.

A new keyword was added:

- `varip` - is similar to the `var` keyword, but variables declared with `varip` retain their values between the updates of a real-time bar.

New functions were added:

- `tonumber()` - converts a string value into a float.
- `time_close()` - returns the UNIX timestamp of the close of the current bar, based on the resolution and session that is passed to the function.
- `dividends()` - requests dividends data for the specified symbol.
- `earnings()` - requests earnings data for the specified symbol.
- `splits()` - requests splits data for the specified symbol.

New arguments for the `study()` function were added:

- `resolution_gaps` - fills the gaps between values fetched from higher timeframes when using `resolution`.
- `format.percent` - formats the script output values as a percentage.

## 8.4.10 February 2021

New variable was added:

- `time_tradingday` - the beginning time of the trading day the current bar belongs to.

## 8.4.11 January 2021

The following functions now accept a series length parameter:

- `bb()`
- `bbw()`
- `cci()`
- `cmo()`
- `cog()`
- `correlation()`
- `dev()`
- `falling()`
- `mfi()`
- `percentile_linear_interpolation()`
- `percentile_nearest_rank()`
- `percentrank()`
- `rising()`
- `roc()`
- `stdev()`
- `stoch()`
- `variance()`
- `wpr()`

A new type of alerts was added - script alerts. More information can be found in our [Help Center](#).

## 8.5 2020

### 8.5.1 December 2020

New array types were added:

- `array.new_line()`
- `array.new_label()`
- `array.new_string()`

New functions were added:

- `str.length()` - returns number of chars in source string.
- `array.join()` - concatenates all of the elements in the array into a string and separates these elements with the specified separator.
- `str.split()` - splits a string at a given substring separator.

### 8.5.2 November 2020

- New `max_labels_count` and `max_lines_count` parameters were added to the study and strategy functions. Now you can manage the number of lines and labels by setting values for these parameters from 1 to 500.

New function was added:

- `array.range()` - return the difference between the min and max values in the array.

### 8.5.3 October 2020

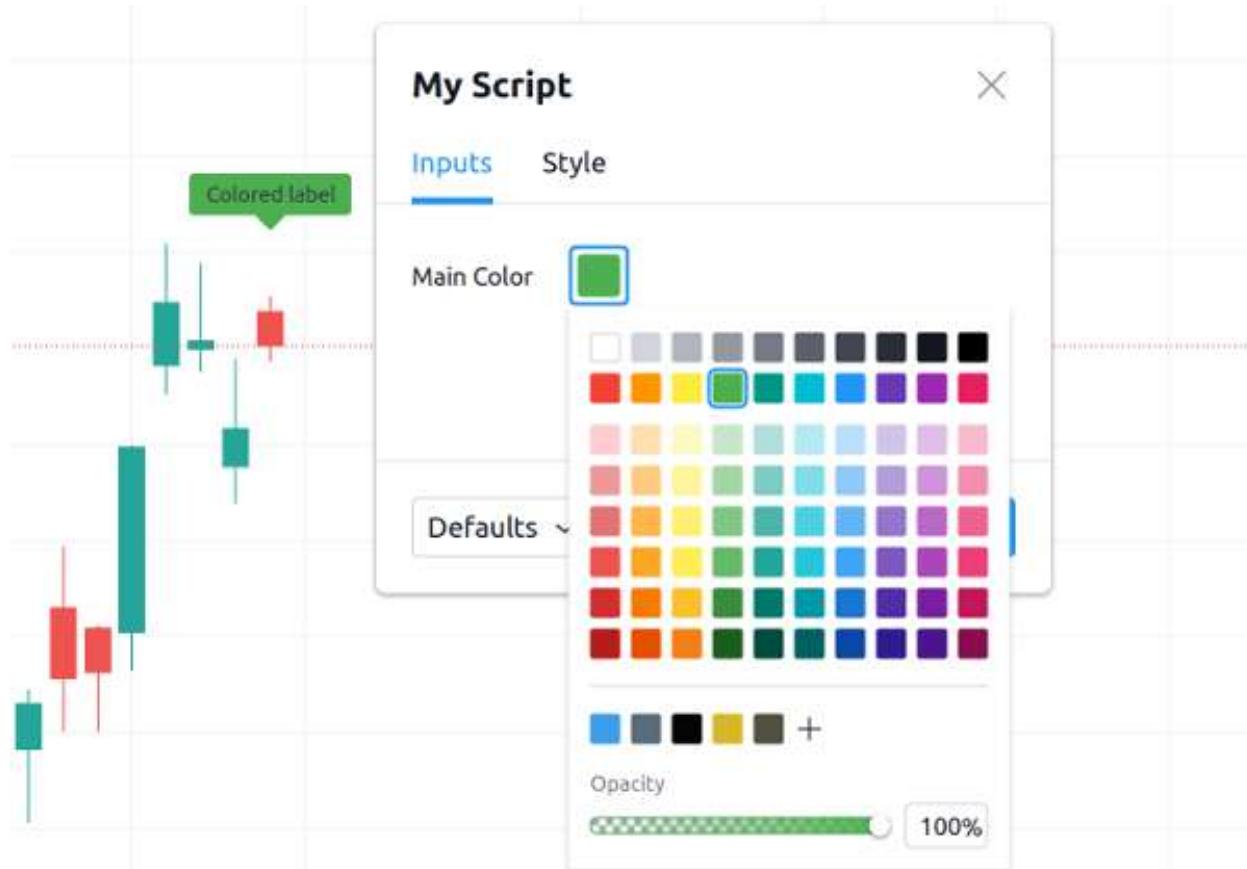
The behavior of `rising()` and `falling()` functions have changed. For example, `rising(close, 3)` is now calculated as following:

```
close[0] > close[1] and close[1] > close[2] and close[2] > close[3]
```

### 8.5.4 September 2020

Added support for `input.color` to the `input()` function. Now you can provide script users with color selection through the script's "Settings/Inputs" tab with the same color widget used throughout the TradingView user interface. Learn more about this feature in our [blog](#)

```
1 // @version=4
2 study("My Script", overlay = true)
3 color c_labelColor = input(color.green, "Main Color", input.color)
4 var l = label.new(bar_index, close, yloc = yloc.abovebar, text = "Colored label")
5 label.set_x(l, bar_index)
6 label.set_color(l, c_labelColor)
```



Added support for arrays and functions for working with them. You can now use the powerful new array feature to build custom datasets. See our User Manual page on [arrays](#) and our [blog](#)

```

1 // @version=4
2 study("My Script")
3 a = array.new_float(0)
4 for i = 0 to 5
5     array.push(a, close[i] - open[i])
6 plot(array.get(a, 4))

```

The following functions now accept a series length parameter. Learn more about this feature in our [blog](#):

- [alma\(\)](#)
- [change\(\)](#)
- [highest\(\)](#)
- [highestbars\(\)](#)
- [linreg\(\)](#)
- [lowest\(\)](#)
- [lowestbars\(\)](#)
- [mom\(\)](#)
- [sma\(\)](#)
- [sum\(\)](#)

- `vwma()`
- `wma()`

```

1 //@version=4
2 study("My Script", overlay = true)
3 length = input(10, "Length", input.integer, minval = 1, maxval = 100)
4 avgBar = avg(highestbars(length), lowestbars(length))
5 float dynLen = nz(abs(avgBar) + 1, length)
6 dynSma = sma(close, int(dynLen))
7 plot(dynSma)

```

## 8.5.5 August 2020

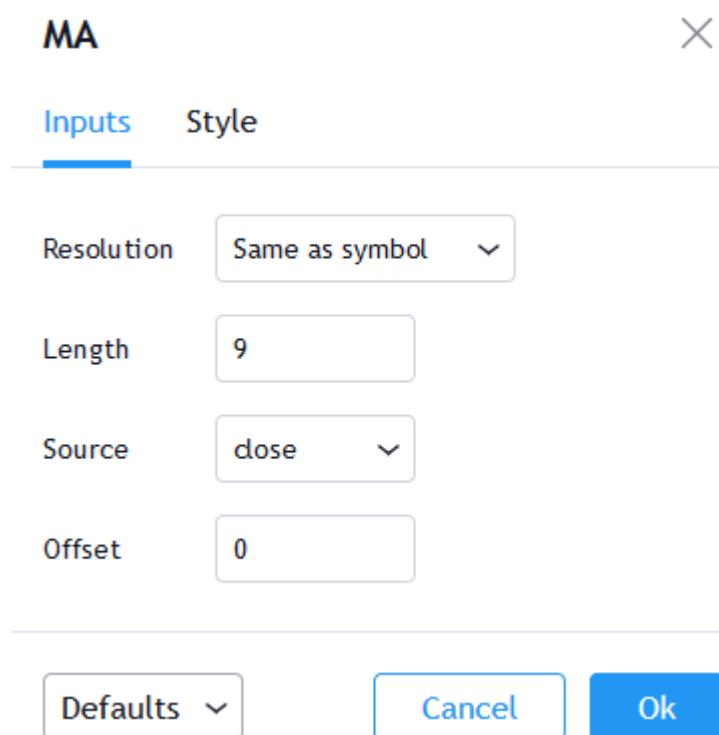
- Optimized script compilation time. Scripts now compile 1.5 to 2 times faster.

## 8.5.6 July 2020

- Minor bug fixes and improvements.

## 8.5.7 June 2020

- New `resolution` parameter was added to the `study` function. Now you can add MTF functionality to scripts and decide the timeframe you want the indicator to run on.



Please note that you need to reapply the indicator in order for the `resolution` parameter to appear.

- The `tooltip` argument was added to the `label.new` function along with the `label.set_tooltip` function:

```
1 // @version=4
2 study("My Script", overlay=true)
3 var l=label.new(bar_index, close, yloc=yloc.abovebar, text="Label")
4 label.set_x(l, bar_index)
5 label.set_tooltip(l, "Label Tooltip")
```



- Added an ability to create alerts on strategies.
- A new function `line.get_price()` can be used to determine the price level at which the line is located on a certain bar.
- New `label` styles allow you to position the label pointer in any direction.



- Find and Replace was added to Pine Editor. To use this, press CTRL+F (find) or CTRL+H (find and replace).



- `timezone` argument was added for time functions. Now you can specify timezone for `second`, `minute`, `hour`, `year`, `month`, `dayofmonth`, `dayofweek` functions:

```

1 //@version=4
2 study("My Script")
3 plot(hour(1591012800000, "GMT+1"))

```

- `syminfo.basecurrency` variable was added. Returns the base currency code of the current symbol. For EURUSD symbol returns EUR.

## 8.5.8 May 2020

- `else if` statement was added
- The behavior of `security()` function has changed: the `expression` parameter can be series or tuple.

## 8.5.9 April 2020

New function was added:

- `quandl()` - request quandl data for a symbol

## 8.5.10 March 2020

New function was added:

- `financial()` - request financial data for a symbol

New functions for common indicators were added:

- `cmo()` - Chande Momentum Oscillator
- `mfi()` - Money Flow Index
- `bb()` - Bollinger Bands
- `bbw()` - Bollinger Bands Width
- `kc()` - Keltner Channels
- `kcw()` - Keltner Channels Width
- `dmi()` - DMI/ADX
- `wpr()` - Williams % R
- `hma()` - Hull Moving Average
- `supertrend()` - SuperTrend

Added a detailed description of all the fields in the [Strategy Tester Report](#).

## 8.5.11 February 2020

- New Pine Script™ indicator VWAP Anchored was added. Now you can specify the time period: Session, Month, Week, Year.
- Fixed a problem with calculating `percentrank` function. Now it can return a zero value, which did not happen before due to an incorrect calculation.
- The default `transparency` parameter for the `plot()`, `plotshape()`, and `plotchar()` functions is now 0%.
- For the functions `plot()`, `plotshape()`, `plotchar()`, `plotbar()`, `plotcandle()`, `plotarrow()`, you can set the `display` parameter, which controls the display of the plot. The following values can be assigned to it:
  - `display.none` - the plot is not displayed
  - `display.all` - the plot is displayed (Default)

- The `textalign` argument was added to the `label.new` function along with the `label.set_textalign` function. Using those, you can control the alignment of the label's text:

## 8.5.12 January 2020

New built-in variables were added:

- `iii` - Intraday Intensity Index
- `wvad` - Williams Variable Accumulation/Distribution
- `wad` - Williams Accumulation/Distribution
- `obv` - On Balance Volume
- `pvt` - Price-Volume Trend
- `nvi` - Negative Volume Index
- `pvi` - Positive Volume Index

New parameters were added for `strategy.close()`:

- `qty` - the number of contracts/shares/lots/units to exit a trade with
- `qty_percent` - defines the percentage of entered contracts/shares/lots/units to exit a trade with
- `comment` - additional notes on the order

New parameter was added for `strategy.close_all`:

- `comment` - additional notes on the order

## 8.6 2019

### 8.6.1 December 2019

- Warning messages were added.

For example, if you don't specify exit parameters for `strategy.exit` - `profit`, `limit`, `loss`, `stop` or one of the following pairs: `trail_offset` and `trail_price` / `trail_points` - you will see a warning message in the console in the Pine Script™ editor.

- Increased the maximum number of arguments in `max`, `min`, `avg` functions. Now you can use up to ten arguments in these functions.

### 8.6.2 October 2019

- `plotchar()` function now supports most of the Unicode symbols:
- New `bordercolor` argument of the `plotcandle()` function allows you to change the color of candles' borders:

```

1 //@version=4
2 study("My Script")
3 plotcandle(open, high, low, close, title='Title', color = open < close ? color.green_
˓→: color.red, wickcolor=color.black, bordercolor=color.orange)

```

- New variables added:
  - `syminfo.description` - returns a description of the current symbol
  - `syminfo.currency` - returns the currency code of the current symbol (EUR, USD, etc.)
  - `syminfo.type` - returns the type of the current symbol (stock, futures, index, etc.)

### 8.6.3 September 2019

New parameters to the `strategy` function were added:

- `process_orders_on_close` allows the broker emulator to try to execute orders after calculating the strategy at the bar's close
- `close_entries_rule` allows to define the sequence used for closing positions

Some fixes were made:

- `fill()` function now works correctly with `na` as the `color` parameter value
- `sign()` function now calculates correctly for literals and constants

`str.replace_all(source, target, replacement)` function was added. It replaces each occurrence of a target string in the source string with a replacement string

### 8.6.4 July-August 2019

New variables added:

- `timeframe.isseconds` returns true when current resolution is in seconds
- `timeframe.isminutes` returns true when current resolution is in minutes
- `time_close` returns the current bar's close time

The behavior of some functions, variables and operators has changed:

- The `time` variable returns the correct open time of the bar for more special cases than before
- An optional `seconds` parameter of the `timestamp()` function allows you to set the time to within seconds
- `security()` function:
  - Added the possibility of requesting resolutions in seconds:
    - 1, 5, 15, 30 seconds (chart resolution should be less than or equal to the requested resolution)
    - Reduced the maximum value that can be requested in some of the other resolutions:
      - from 1 to 1440 minutes
      - from 1 to 365 days
      - from 1 to 52 weeks
      - from 1 to 12 months
  - Changes to the evaluation of ternary operator branches:

In Pine Script™ v3, during the execution of a ternary operator, both its branches are calculated, so when this script is added to the chart, a long position is opened, even if the `long()` function is not called:

## 8.6.5 June 2019

- Support for drawing objects. Added *label* and *line* drawings
- `var` keyword for one time variable initialization
- Type system improvements:
  - *series string* data type
  - functions for explicit type casting
  - syntax for explicit variable type declaration
  - new *input* type forms
- Renaming of built-ins and a version 3 to 4 converter utility
- `max_bars_back` function to control series variables internal history buffer sizes
- Pine Script™ documentation versioning

## 8.7 2018

### 8.7.1 October 2018

- To increase the number of indicators available to the whole community, Invite-Only scripts can now be published by Premium users only.

### 8.7.2 April 2018

- Improved the Strategy Tester by reworking the Maximum Drawdown calculation formula.

## 8.8 2017

### 8.8.1 August 2017

- With the new argument `show_last` in the plot-type functions, you can restrict the number of bars that the plot is displayed on.

### 8.8.2 June 2017

- A major script publishing improvement: it is now possible to update your script without publishing a new one via the Update button in the publishing dialog.

### 8.8.3 May 2017

- Expanded the type system by adding a new type of constants that can be calculated during compilation.

### 8.8.4 April 2017

- Expanded the keyword argument functionality: it is now possible to use keyword arguments in all built-in functions.
- A new `barstate.isconfirmed` variable has been added to the list of variables that return bar status. It lets you create indicators that are calculated based on the closed bars only.
- The `options` argument for the `input()` function creates an input with a set of options defined by the script's author.

### 8.8.5 March 2017

- Pine Script™ v3 is here! Some important changes:
  - Changes to the default behavior of the `security()` function: it can no longer access the future data by default. This can be changes with the `lookahead` parameter.
  - An implicit conversion of boolean values to numeric values was replaced with an implicit conversion of numeric values (integer and float) to boolean values.
  - Self-referenced and forward-referenced variables were removed. Any PineScript code that used those language constructions can be equivalently rewritten using mutable variables.

### 8.8.6 February 2017

- Several improvements to the strategy tester and the strategy report:
  - New Buy & Hold equity graph – a new graph that lets you compare performance of your strategy versus a “buy and hold”, i.e if you just bought a security and held onto it without trading.
  - Added percentage values to the absolute currency values.
  - Added Buy & Hold Return to display the final value of Buy & Hold Equity based on last price.
  - Added Sharpe Ratio – it shows the relative effectiveness of the investment portfolio (security), a measure that indicates the average return minus the risk-free return divided by the standard deviation of return on an investment.
  - Slippage lets you simulate a situation when orders are filled at a worse price than expected. It can be set through the Properties dialog or through the `slippage` argument in the `strategy()` function.
  - Commission allows yet to add commission for placed orders in percent of order value, fixed price or per contract. The amount of commission paid is shown in the Commission Paid field. The commission size and its type can be set through the Properties dialog or through the `commission_type` and `commission_value` arguments in the `strategy()` function.

## 8.9 2016

### 8.9.1 December 2016

- Added invite-only scripts. The invite-only indicators are visible in the Community Scripts, but nobody can use them without explicit permission from the author, and only the author can see the source code.

### 8.9.2 October 2016

- Introduced indicator revisions. Each time an indicator is saved, it gets a new revision, and it is possible to easily switch to any past revision from the Pine Editor.

### 8.9.3 September 2016

- It is now possible to publish indicators with protected source code. These indicators are available in the public Script Library, and any user can use them, but only the author can see the source code.

### 8.9.4 July 2016

- Improved the behavior of the `fill()` function: one call can now support several different colors.

### 8.9.5 March 2016

- Color type variables now have an additional parameter to set default transparency. The transparency can be set with the `color.new()` function, or by adding an alpha-channel value to a hex color code.

### 8.9.6 February 2016

- Added `for` loops and keywords `break` and `continue`.
- Pine Script™ now supports mutable variables! Use the `:=` operator to assign a new value to a variable that has already been defined.
- Multiple improvements and bug fixes for strategies.

### 8.9.7 January 2016

- A new `alertcondition()` function allows for creating custom alert conditions in Pine Script™-based indicators.

## 8.10 2015

### 8.10.1 October 2015

- Pine has graduated to v2! The new version of Pine Script™ added support for `if` statements, making it easier to write more readable and concise code.

### 8.10.2 September 2015

- Added backtesting functionality to Pine Script™. It is now possible to create trading strategies, i.e. scripts that can send, modify and cancel orders to buy or sell. Strategies allow you to perform backtesting (emulation of strategy trading on historical data) and forward testing (emulation of strategy trading on real-time data) according to your algorithms. Detailed information about the strategy's calculations and the order fills can be seen in the newly added Strategy Tester tab.

### 8.10.3 July 2015

- A new `editable` parameter allows hiding the plot from the Style menu in the indicator settings so that it is not possible to edit its style. The parameter has been added to all the following functions: all plot-type functions, `barcolor()`, `bgcolor()`, `hline()`, and `fill()`.

### 8.10.4 June 2015

- Added two new functions to display custom barsets using PineScript: `plotbar()` and `plotcandle()`.

### 8.10.5 April 2015

- Added two new shapes to the `plotshape()` function: `shape.labelup` and `shape.labardown`.
- PineScript Editor has been improved and moved to a new panel at the bottom of the page.
- Added a new `step` argument for the `input()` function, allowing to specify the step size for the indicator's inputs.

### 8.10.6 March 2015

- Added support for inputs with the `source` type to the `input()` function, allowing to select the data source for the indicator's calculations from its settings.

### 8.10.7 February 2015

- Added a new `text` argument to `plotshape()` and `plotchar()` functions.
- Added four new shapes to the `plotshape()` function: `shape.arrowup`, `shape.arrowdown`, `shape.square`, `shape.diamond`.

## 8.11 2014

### 8.11.1 August 2014

- Improved the script sharing capabilities, changed the layout of the Indicators menu and separated published scripts from ideas.

### 8.11.2 July 2014

- Added three new plotting functions, `plotshape()`, `plotchar()`, and `plotarrow()` for situations when you need to highlight specific bars on a chart without drawing a line.
- Integrated QUANDL data into Pine Script™. The data can be accessed by passing the QUANDL ticker to the `security` function.

### 8.11.3 June 2014

- Added Pine Script™ sharing, enabling programmers and traders to share their scripts with the rest of the TradingView community.

### 8.11.4 April 2014

- Added line wrapping.

### 8.11.5 February 2014

- Added support for inputs, allowing users to edit the indicator inputs through the properties window, without needing to edit the Pine script.
- Added self-referencing variables.
- Added support for multiline functions.
- Implemented the type-casting mechanism, automatically casting constant and simple float and int values to series when it is required.
- Added several new functions and improved the existing ones:
  - `barssince()` and `valuewhen()` allow you to check conditions on historical data easier.
  - The new `barcolor()` function lets you specify a color for a bar based on filling of a certain condition.
  - Similar to the `barcolor()` function, the `bgcolor()` function changes the color of the background.
  - Reworked the `security()` function, further expanding its functionality.
  - Improved the `fill()` function, enabling it to be used more than once in one script.
  - Added the `round()` function to round and convert float values to integers.

## 8.12 2013

- The first version of Pine Script™ is introduced to all TradingView users, initially as an open beta, on December 13th.



## MIGRATION GUIDES



### 9.1 To Pine Script™ version 5

- *Introduction*
- *v4 to v5 converter*
- *Renamed functions and variables*
- *Renamed function parameters*
- *Removed an `rsi()` overload*
- *Reserved keywords*
- *Removed `iff()` and `offset()`*
- *Split of `input()` into several functions*
- *Some function parameters now require built-in arguments*
- *Deprecated the `transp` parameter*
- *Changed the default session days for `time()` and `time\_close()`*
- *`strategy.exit()` now must do something*
- *Common script conversion errors*
- *All variable, function, and parameter name changes*

## 9.1.1 Introduction

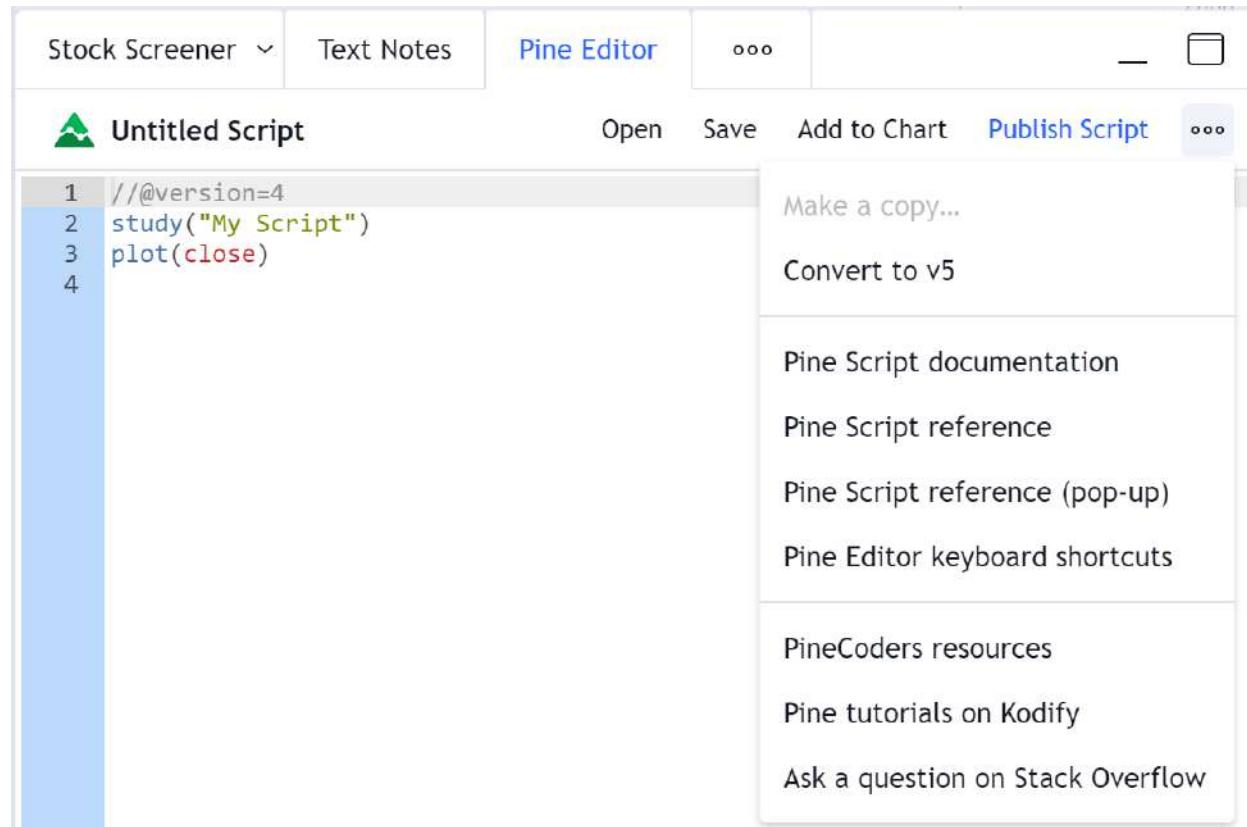
This guide documents the **changes** made to Pine Script™ from v4 to v5. It will guide you in the adaptation of existing Pine scripts to Pine Script™ v5. See our [Release notes](#) for a list of the **new** features in Pine Script™ v5.

The most frequent adaptations required to convert older scripts to v5 are:

- Changing `study()` for `indicator()` (the function's signature has not changed).
- Renaming built-in function calls to include their new namespace (e.g., `highest()` in v4 becomes `ta.highest()` in v5).
- Restructuring inputs to use the more specialized `input.*()` functions.
- Eliminating uses of the deprecated `transp` parameter by using `color.new()` to simultaneously define color and transparency for use with the `color` parameter.
- If you used the `resolution` and `resolution_gaps` parameters in v4's `study()`, they will require changing to `timeframe` and `timeframe_gaps` in v5's `indicator()`.

## 9.1.2 v4 to v5 converter

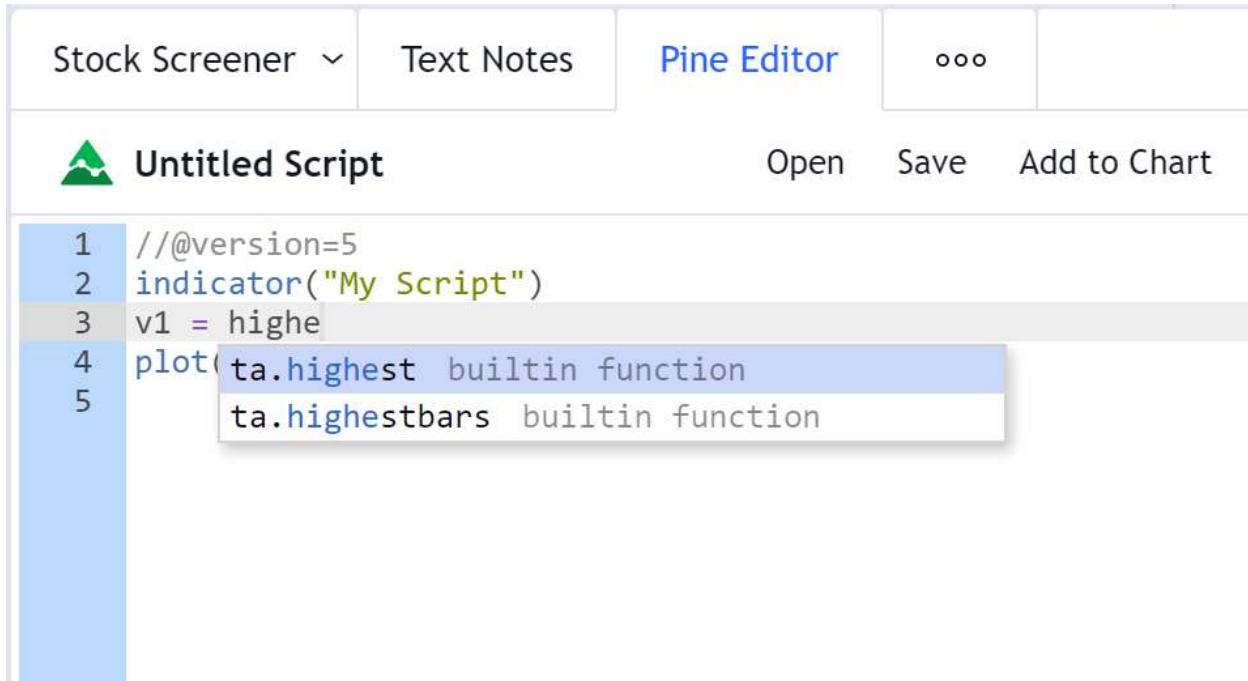
The Pine Editor includes a utility to automatically convert v4 scripts to v5. To access it, open a script with `// @version=4` in it and select the “Convert to v5” option in the “More” menu identified by three dots at the top-right of the Editor's pane:



Not all scripts can be automatically converted from v4 to v5. If you want to convert the script manually or if your indicator returns a compilation error after conversion, use the following sections to determine how to complete the conversion. A list of some errors you can encounter during the automatic conversion and how to fix them can be found in the Common script conversion errors section of this guide.

### 9.1.3 Renamed functions and variables

For clarity and consistency, many built-in functions and variables were renamed in v5. The inclusion of v4 function names in a new namespace is the cause of most changes. For example, the `sma()` function in v4 is moved to the `ta.` namespace in v5: `ta.sma()`. Remembering the new namespaces is not necessary; if you type the older name of a function without its namespace in the Editor and press the ‘Auto-complete’ hotkey (Ctrl + Space, or Cmd + Space on MacOS), a popup showing matching suggestions appears:



Not counting functions moved to new namespaces, only two functions have been renamed:

- `study()` is now `indicator()`.
- `tickerid()` is now `ticker.new()`.

The full list of renamed functions and variables can be found in the All variable, function, and parameter name changes section of this guide.

### 9.1.4 Renamed function parameters

The parameter names of some built-in functions were changed to improve the nomenclature. This has no bearing on most scripts, but if you used these parameter names when calling functions, they will require adaptation. For example, we have standardized all mentions:

```

// Valid in v4. Not valid in v5.
timev4 = time(resolution = "1D")
// Valid in v5.
timev5 = time(timeframe = "1D")
// Valid in v4 and v5.
timeBoth = time("1D")

```

The full list of renamed function parameters can be found in the All variable, function, and parameter name changes section of this guide.

## 9.1.5 Removed an `rsi()` overload

In v4, the `rsi()` function had two different overloads:

- `rsi(series float, simple int)` for the normal RSI calculation, and
- `rsi(series float, series float)` for an overload used in the MFI indicator, which did a calculation equivalent to  $100.0 - (100.0 / (1.0 + \text{arg1} / \text{arg2}))$ .

This caused a single built-in function to behave in two very different ways, and it was difficult to distinguish which one applied because it depended on the type of the second argument. As a result, a number of indicators misused the function and were displaying incorrect results. To avoid this, the second overload was removed in v5.

The `ta.rsi()` function in v5 only accepts a “simple int” argument for its `length` parameter. If your v4 code used the now deprecated overload of the function with a `float` second argument, you can replace the whole `rsi()` call with the following formula, which is equivalent:

```
100.0 - (100.0 / (1.0 + arg1 / arg2))
```

Note that when your v4 code used a “series int” value as the second argument to `rsi()`, it was automatically cast to “series float” and the second overload of the function was used. While this was syntactically correct, it most probably did **not** yield the result you expected. In v5, `ta.rsi()` requires a “simple int” for the argument to `length`, which precludes dynamic (or “series”) lengths. The reason for this is that RSI calculations use the `ta.rma()` moving average, which is similar to `ta.ema()` in that it relies on a length-dependent recursive process using the values of previous bars. This makes it impossible to achieve correct results with a “series” length that could vary bar to bar.

If your v4 code used a length that was “const int”, “input int” or “simple int”, no changes are required.

## 9.1.6 Reserved keywords

A number of words are reserved and cannot be used for variable or function names. They are: `catch`, `class`, `do`, `ellipse`, `in`, `is`, `polygon`, `range`, `return`, `struct`, `text`, `throw`, `try`. If your v4 indicator uses any of these, rename your variable or function for the script to work in v5.

## 9.1.7 Removed `iff()` and `offset()`

The `iff()` and `offset()` functions have been removed. Code using the `iff()` function can be rewritten using the ternary operator:

```
// iff(<condition>, <return_when_true>, <return_when_false>)
// Valid in v4, not valid in v5
barColorIff = iff(close >= open, color.green, color.red)
// <condition> ? <return_when_true> : <return_when_false>
// Valid in v4 and v5
barColorTernary = close >= open ? color.green : color.red
```

Note that the ternary operator is evaluated “lazily”; only the required value is calculated (depending on the condition’s evaluation to `true` or `false`). This is different from `iff()`, which always evaluated both values but returned only the relevant one.

Some functions require evaluation on every bar to correctly calculate, so you will need to make special provisions for these by pre-evaluating them before the ternary:

```
// `iff()` in v4: `highest()` and `lowest()` are calculated on every bar
v1 = iff(close > open, highest(10), lowest(10))
plot(v1)
```

(continues on next page)

(continued from previous page)

```
// In v5: forced evaluation on every bar prior to the ternary statement.
h1 = ta.highest(10)
l1 = ta.lowest(10)
v1 = close > open ? h1 : l1
plot(v1)
```

The `offset()` function was deprecated because the more readable `[]` operator is equivalent:

```
// Valid in v4. Not valid in v5.
prevClosev4 = offset(close, 1)
// Valid in v4 and v5.
prevClosev5 = close[1]
```

### 9.1.8 Split of `input()` into several functions

The v4 `input()` function was becoming crowded with a plethora of overloads and parameters. We split its functionality into different functions to clear that space and provide a more robust structure to accommodate the additions planned for inputs. Each new function uses the name of the `input.*` type of the v4 `input()` call it replaces. E.g., there is now a specialized `input.float()` function replacing the v4 `input(1.0, type = input.float)` call. Note that you can still use `input(1.0)` in v5, but because only `input.float()` allows for parameters such as `minval`, `maxval`, etc., it is more powerful. Also note that `input.int()` is the only specialized input function that does not use its equivalent v4 `input.integer` name. The `input.*` constants have been removed because they were used as arguments for the `type` parameter, which was deprecated.

To convert, for example, a v4 script using an input of type `input.symbol`, the `input.symbol()` function must be used in v5:

```
// Valid in v4. Not valid in v5.
aaplTicker = input("AAPL", type = input.symbol)
// Valid in v5
aaplTicker = input.symbol("AAPL")
```

The `input()` function persists in v5, but in a simpler form, with less parameters. It has the advantage of automatically detecting input types “bool/color/int/float/string/source” from the argument used for `defval`:

```
// Valid in v4 and v5.
// While "AAPL" is a valid symbol, it is only a string here because `input.symbol()` ↴
// is not used.
tickerString = input("AAPL", title = "Ticker string")
```

### 9.1.9 Some function parameters now require built-in arguments

In v4, built-in constants such as `plot.style_area` used as arguments when calling Pine Script™ functions corresponded to pre-defined values of a specific type. For example, the value of `barmerge.lookahead_on` was `true`, so you could use `true` instead of the named constant when supplying an argument to the `lookahead` parameter in a `security()` function call. We found this to be a common source of confusion, which caused unsuspecting programmers to produce code yielding unintended results.

In v5, the use of correct built-in named constants as arguments to function parameters requiring them is mandatory:

```
// Not valid in v5: `true` is used as an argument for `lookahead`.
request.security(syminfo.tickerid, "1D", close, lookahead = true)
```

(continues on next page)

(continued from previous page)

```
// Valid in v5: uses a named constant instead of `true`.
request.security(syminfo.tickerid, "1D", close, lookahead = barmerge.lookahead_on)

// Would compile in v4 because `plot.style_columns` was equal to 5.
// Won't compile in v5.
a = 2 * plot.style_columns
plot(a)
```

To convert your script from v4 to v5, make sure you use the correct named built-in constants as function arguments.

### 9.1.10 Deprecated the `transp` parameter

The `transp=` parameter used in the signature of many v4 plotting functions was deprecated because it interfered with RGB functionality. Transparency must now be specified along with the color as an argument to parameters such as `color`, `textcolor`, etc. The `color.new()` or `color.rgb()` functions will be needed in those cases to join a color and its transparency.

Note that in v4, the `bcolor()` and `fill()` functions had an optional `transp` parameter that used a default value of 90. This meant that the code below could display Bollinger Bands with a semi-transparent fill between two bands and a semi-transparent background color where bands cross price, even though no argument is used for the `transp` parameter in its `bcolor()` and `fill()` calls:

```
1 // @version=4
2 study("Bollinger Bands", overlay = true)
3 [middle, upper, lower] = bb(close, 5, 4)
4 plot(middle, color=color.blue)
5 p1PlotID = plot(upper, color=color.green)
6 p2PlotID = plot(lower, color=color.green)
7 crossUp = crossover(high, upper)
8 crossDn = crossunder(low, lower)
9 // Both `fill()` and `bcolor()` have a default `transp` of 90
10 fill(p1PlotID, p2PlotID, color = color.green)
11 bcolor(crossUp ? color.green : crossDn ? color.red : na)
```

In v5 we need to explicitly mention the 90 transparency with the `color`, yielding:

```
1 // @version=5
2 indicator("Bollinger Bands", overlay = true)
3 [middle, upper, lower] = ta.bb(close, 5, 4)
4 plot(middle, color=color.blue)
5 p1PlotID = plot(upper, color=color.green)
6 p2PlotID = plot(lower, color=color.green)
7 crossUp = ta.crossover(high, upper)
8 crossDn = ta.crossunder(low, lower)
9 var TRANSP = 90
10 // We use `color.new()` to explicitly pass transparency to both functions
11 fill(p1PlotID, p2PlotID, color = color.new(color.green, TRANSP))
12 bcolor(crossUp ? color.new(color.green, TRANSP) : crossDn ? color.new(color.red, ↵TRANSP) : na)
```

### 9.1.11 Changed the default session days for `time()` and `time\_close()`

The default set of days for session strings used in the `time()` and `time_close()` functions, and returned by `input.session()`, has changed from "23456" (Monday to Friday) to "1234567" (Sunday to Saturday):

```
// On symbols that are traded during weekends, this will behave differently in v4 and
// v5.
t0 = time("1D", "1000-1200")
// v5 equivalent of the behavior of `t0` in v4.
t1 = time("1D", "1000-1200:23456")
// v5 equivalent of the behavior of `t0` in v5.
t2 = time("1D", "1000-1200:1234567")
```

This change in behavior should not have much impact on scripts running on conventional markets that are closed during weekends. If it is important for you to ensure your session definitions preserve their v4 behavior in v5 code, add ":23456" to your session strings. See this manual's page on [Sessions](#) for more information.

### 9.1.12 `strategy.exit()` now must do something

Gone are the days when the `strategy.exit()` function was allowed to loiter. Now it must actually have an effect on the strategy by using at least one of the following parameters: `profit`, `limit`, `loss`, `stop`, or one of the following pairs: `trail_offset` combined with either `trail_price` or `trail_points`. When uses of `strategy.exit()` not meeting these criteria trigger an error while converting a strategy to v5, you can safely eliminate these lines, as they didn't do anything in your code anyway.

### 9.1.13 Common script conversion errors

#### Invalid argument 'style'/'linestyle' in 'plot()'/'hline' call

To make this work, you need to change the "int" arguments used for the `style` and `linestyle` arguments in `plot()` and `hline()` for built-in constants:

```
// Will cause an error during conversion
plotStyle = input(1)
hlineStyle = input(1)
plot(close, style = plotStyle)
hline(100, linestyle = hlineStyle)

// Will work in v5
//@version=5
indicator("")
plotStyleInput = input.string("Line", options = ["Line", "Stepline", "Histogram",
// "Cross", "Area", "Columns", "Circles"])
hlineStyleInput = input.string("Solid", options = ["Solid", "Dashed", "Dotted"])

plotStyle = plotStyleInput == "Line" ? plot.style_line :
plotStyleInput == "Stepline" ? plot.style_stepline :
plotStyleInput == "Histogram" ? plot.style_histogram :
plotStyleInput == "Cross" ? plot.style_cross :
plotStyleInput == "Area" ? plot.style_area :
plotStyleInput == "Columns" ? plot.style_columns :
plot.style_circles

hlineStyle = hlineStyleInput == "Solid" ? hline.style_solid :
```

(continues on next page)

(continued from previous page)

```

hlineStyleInput == "Dashed" ? hline.style_dashed :
hline.style_dotted

plot(close, style = plotStyle)
hline(100, linestyle = hlineStyle)

```

See the Some function parameters now require built-in arguments section of this guide for more information.

### Undeclared identifier 'input.%input\_name%'

To fix this issue, remove the `input.*` constants from your code:

```

// Will cause an error during conversion
_integer = input.integer
_bool = input.bool
i1 = input(1, "Integer", _integer)
i2 = input(true, "Boolean", _bool)

// Will work in v5
i1 = input.int(1, "Integer")
i2 = input.bool(true, "Boolean")

```

See the User Manual's page on [Inputs](#), and the Some function parameters now require built-in arguments section of this guide for more information.

### Invalid argument 'when' in 'strategy.close' call

This is caused by a confusion between `strategy.entry()` and `strategy.close()`.

The second parameter of `strategy.close()` is `when`, which expects a “bool” argument. In v4, it was allowed to use `strategy.long` an argument because it was a “bool”. With v5, however, named built-in constants must be used as arguments, so `strategy.long` is no longer allowed as an argument to the `when` parameter.

The `strategy.close("Short", strategy.long)` call in this code is equivalent to `strategy.close("Short")`, which is what must be used in v5:

```

// Will cause an error during conversion
if (longCondition)
    strategy.close("Short", strategy.long)
    strategy.entry("Long", strategy.long)

// Will work in v5:
if (longCondition)
    strategy.close("Short")
    strategy.entry("Long", strategy.long)

```

See the Some function parameters now require built-in arguments section of this guide for more information.

## Cannot call ‘input.int’ with argument ‘minval’=%value%. An argument of ‘literal float’ type was used but a ‘const int’ is expected

In v4, it was possible to pass a “float” argument to `minval` when an “int” value was being input. This is no longer possible in v5; “int” values are required for “int” inputs:

```
// Works in v4, will break on conversion because minval is a 'float' value
int_input = input(1, "Integer", input.integer, minval = 1.0)

// Works in v5
int_input = input.int(1, "Integer", minval = 1)
```

See the User Manual’s page on [Inputs](#), and the Some function parameters now require built-in arguments section of this guide for more information.

### 9.1.14 All variable, function, and parameter name changes

#### Removed functions and variables

v4	v5
<code>input.bool</code> input	Replaced by <code>input.bool()</code>
<code>input.color</code> input	Replaced by <code>input.color()</code>
<code>input.float</code> input	Replaced by <code>input.float()</code>
<code>input.integer</code> input	Replaced by <code>input.int()</code>
<code>input.resolution</code> input	Replaced by <code>input.timeframe()</code>
<code>input.session</code> input	Replaced by <code>input.session()</code>
<code>input.source</code> input	Replaced by <code>input.source()</code>
<code>input.string</code> input	Replaced by <code>input.string()</code>
<code>input.symbol</code> input	Replaced by <code>input.symbol()</code>
<code>input.time</code> input	Replaced by <code>input.time()</code>
<code>iff()</code>	Use the <code>? :</code> operator instead
<code>offset()</code>	Use the <code>[ ]</code> operator instead

#### Renamed functions and parameters

##### No namespace change

v4	v5
<code>study(&lt;...&gt;, resolution, resolution_gaps, &lt;...&gt;)</code>	<code>indicator(&lt;...&gt;, timeframe, timeframe_gaps, &lt;...&gt;)</code>
<code>strategy.entry(long)</code>	<code>strategy.entry(direction)</code>
<code>strategy.order(long)</code>	<code>strategy.order(direction)</code>
<code>time(resolution)</code>	<code>time(timeframe)</code>
<code>time_close(resolution)</code>	<code>time_close(timeframe)</code>
<code>nz(x, y)</code>	<code>nz(source, replacement)</code>

**“ta” namespace for technical analysis functions and variables**

v4	v5
<b>Indicator functions and variables</b>	
accdist	ta.accdist
alma()	ta.alma()
atr()	ta.atr()
bb()	ta.bb()
bbw()	ta.bbw()
cci()	ta.cci()
cmo()	ta.cmo()
cog()	ta.cog()
dmi()	ta.dmi()
ema()	ta.ema()
hma()	ta.hma()
iii	ta.iii
kc()	ta.kc()
kcw()	ta.kcw()
linreg()	ta.linreg()
macd()	ta.macd()
mfi()	ta.mfi()
mom()	ta.mom()
nvi	ta.nvi
obv	ta.obv
pvi	ta.pvi
pvt	ta.pvt
rma()	ta.rma()
roc()	ta.roc()
rsi(x, y)	ta.rsi(source, length)
sar()	ta.sar()
sma()	ta.sma()
stoch()	ta.stoch()
supertrend()	ta.supertrend()
swma(x)	ta.swma(source)
tr	ta.tr
tr()	ta.tr()
tsi()	ta.tsi()
vwap	ta.vwap
vwap(x)	ta.vwap(source)
vwma()	ta.vwma()
wad	ta.wad
wma()	ta.wma()
wpr()	ta.wpr()
wvad	ta.wvad
<b>Supporting functions</b>	
barsince()	ta.barsince()
change()	ta.change()
correlation(source_a, source_b, length)	ta.correlation(source1, source2, length)
cross(x, y)	ta.cross(source1, source2)
crossover(x, y)	ta.crossover(source1, source2)
crossunder(x, y)	ta.crossunder(source1, source2)
cum(x)	ta.cum(source)

continues on next page

Table 1 – continued from previous page

dev()	ta.dev()
falling()	ta.falling()
highest()	ta.highest()
highestbars()	ta.highestbars()
lowest()	ta.lowest()
lowestbars()	ta.lowestbars()
median()	ta.median()
mode()	ta.mode()
percentile_linear_interpolation()	ta.percentile_linear_interpolation()
percentile_nearest_rank()	ta.percentile_nearest_rank()
percentrank()	ta.percentrank()
pivothigh()	ta.pivothigh()
pivotlow()	ta.pivotlow()
range()	ta.range()
rising()	ta.rising()
stdev()	ta.stdev()
valuewhen()	ta.valuewhen()
variance()	ta.variance()

**“math” namespace for math-related functions and variables**

v4	v5
abs(x)	math.abs(number)
acos(x)	math.acos(number)
asin(x)	math.asin(number)
atan(x)	math.atan(number)
avg()	math.avg()
ceil(x)	math.ceil(number)
cos(x)	math.cos(angle)
exp(x)	math.exp(number)
floor(x)	math.floor(number)
log(x)	math.log(number)
log10(x)	math.log10(number)
max()	math.max()
min()	math.min()
pow()	math.pow()
random()	math.random()
round(x, precision)	math.round(number, precision)
round_to_mintick(x)	math.round_to_mintick(number)
sign(x)	math.sign(number)
sin(x)	math.sin(angle)
sqrt(x)	math.sqrt(number)
sum()	math.sum()
tan(x)	math.tan(angle)
todegrees()	math.todegrees()
toradians()	math.toradians()

“request” namespace for functions that request external data

v4	v5
financial()	request.financial()
quandl()	request.quandl()
security(<...>, resolution, <...>)	request.security(<...>, timeframe, <...>)
splits()	request.splits()
dividends()	request.dividends()
earnings()	request.earnings()

“ticker” namespace for functions that help create tickers

v4	v5
heikinashi()	ticker.heikinashi()
kagi()	ticker.kagi()
linebreak()	ticker.linebreak()
pointfigure()	ticker.pointfigure()
renko()	ticker.renko()
tickerid()	ticker.new()

“str” namespace for functions that manipulate strings

v4	v5
tostring(x, y)	str.tostring(value, format)
tonumber(x)	str tonumber(string)

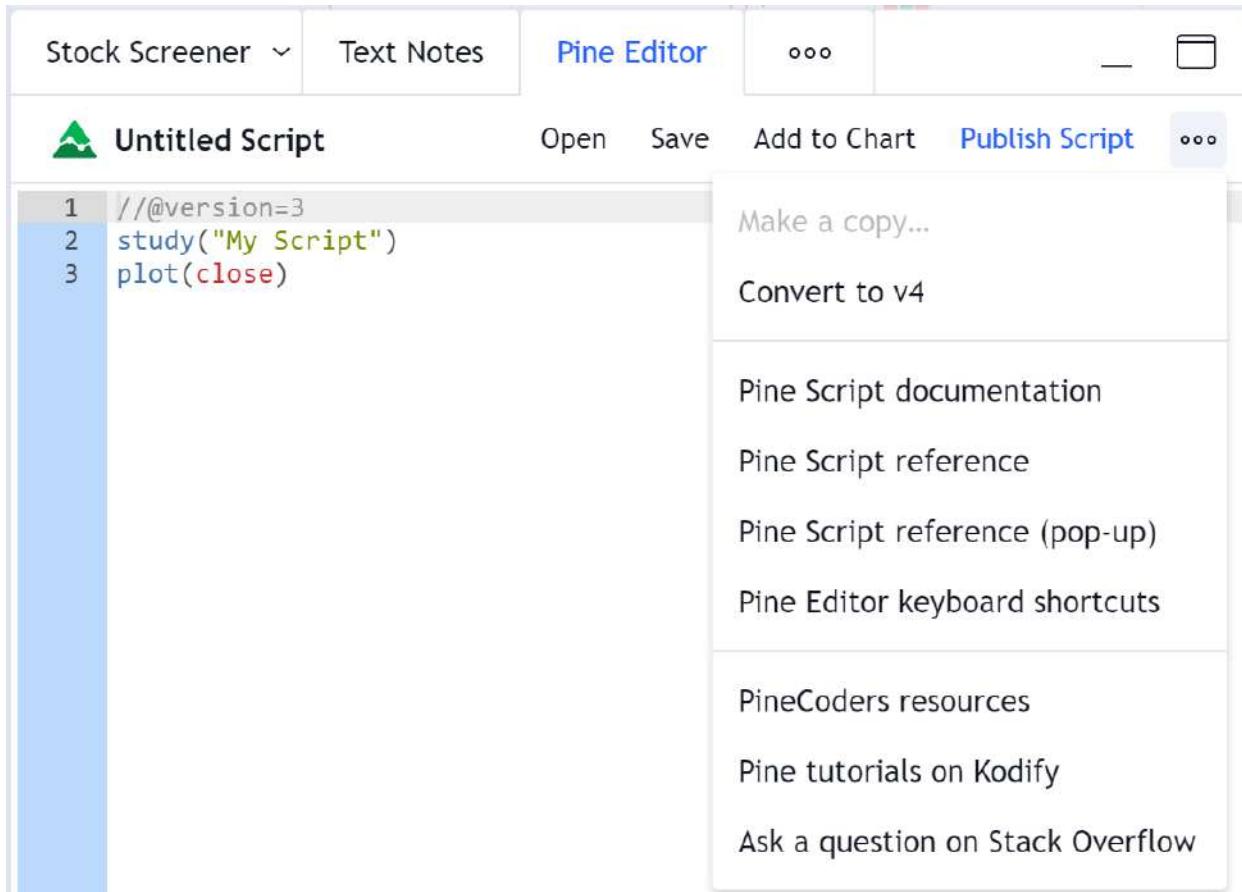


## 9.2 To Pine Script™ version 4

This is a guide to converting Pine Script™ code from @version=3 to @version=4.

## 9.2.1 Converter

The Pine Editor comes with a utility to automatically convert v3 indicators and strategies to v4. To access it, open a script with `//@version=3` in it and select the `Convert to v4` option in the More dropdown menu:



Not all scripts can be automatically converted from v3 to v4. If you want to convert the script manually or if your indicator returns a compilation error after conversion, consult the guide below for more information.

## 9.2.2 Renaming of built-in constants, variables, and functions

In Pine Script™ v4 the following built-in constants, variables, and functions were renamed:

- Color constants (e.g. `red`) are moved to the `color.*` namespace (e.g. `color.red`).
- The `color` function has been renamed to `color.new`.
- Constants for `input()` types (e.g. `integer`) are moved to the `input.*` namespace (e.g. `input.integer`).
- The plot style constants (e.g. `histogram` style) are moved to the `plot.style_*` namespace (e.g. `plot.style_histogram`).
- Style constants for the `hline` function (e.g. the `dotted` style) are moved to the `hline.style_*` namespace (e.g. `hline.style_dotted`).
- Constants of days of the week (e.g. `sunday`) are moved to the `dayofweek.*` namespace (e.g. `dayofweek.sunday`).

- The variables of the current chart timeframe (e.g. period, isintraday) are moved to the timeframe.\* namespace (e.g. timeframe.period, timeframe.isintraday).
- The interval variable was renamed to timeframe.multiplier.
- The ticker and tickerid variables are renamed to syminfo.ticker and syminfo.tickerid respectively.
- The n variable that contains the bar index value has been renamed to bar\_index.

The reason behind renaming all of the above was to structure the standard language tools and make working with code easier. New names are grouped according to assignments under common prefixes. For example, you will see a list with all available color constants if you type ‘color’ in the editor and press Ctrl + Space.

### 9.2.3 Explicit variable type declaration

In Pine Script™ v4 it's no longer possible to create variables with an unknown data type at the time of their declaration. This was done to avoid a number of issues that arise when the variable type changes after its initialization with the na value. From now on, you need to explicitly specify their type using keywords or type functions (for example, float) when declaring variables with the na value:



## 9.3 To Pine Script™ version 3

This document helps to migrate Pine Script™ code from @version=2 to @version=3.

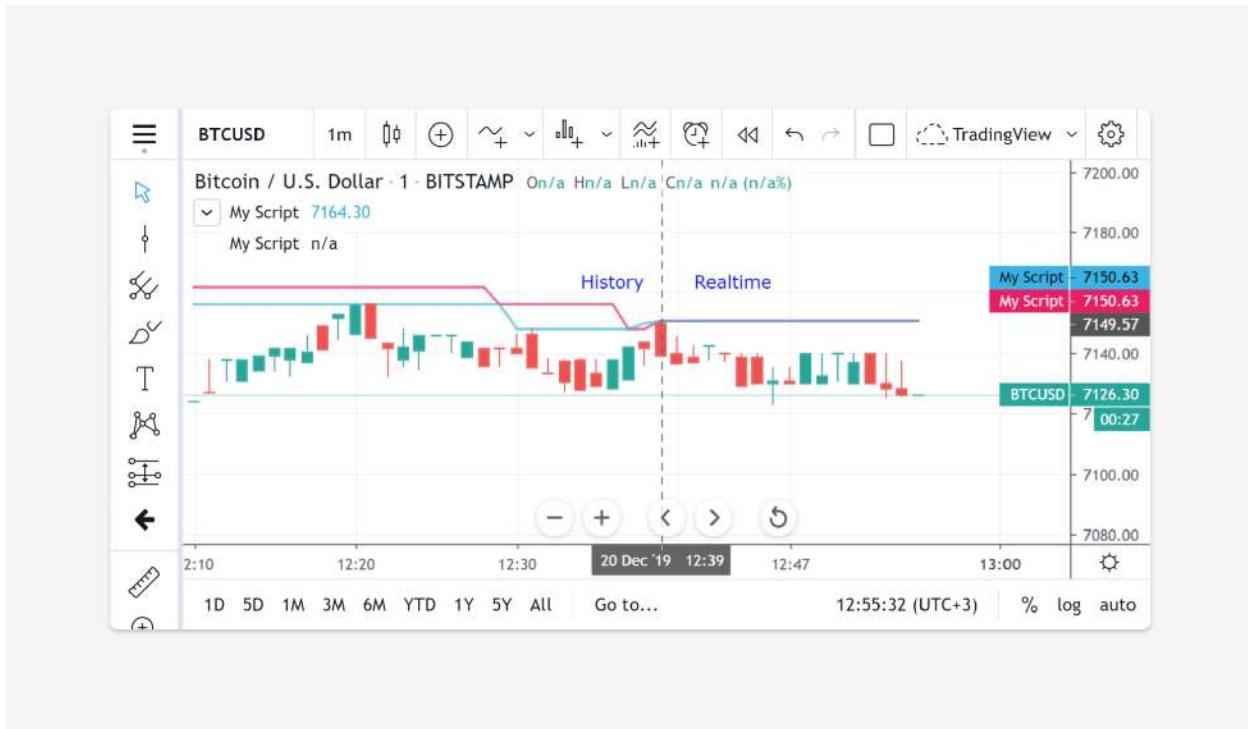
### 9.3.1 Default behaviour of security function has changed

Let's look at the simple security function use case. Add this indicator on an intraday chart:

```
1 // Add this indicator on an intraday (e.g., 30 minutes) chart
2 //@version=2
3 study("My Script", overlay=true)
4 s = security(tickerid, 'D', high, false)
5 plot(s)
```

This indicator is calculated based on historical data and looks somewhat *into the future*. At the first bar of every session an indicator plots the high price of the entire day. This could be useful in some cases for analysis, but doesn't work for backtesting strategies.

We worked on this and made changes in Pine Script™ version 3. If this indicator is compiled with //@version=3 directive, we get a completely different picture:



The old behaviour is still available though. We added a parameter to the `security` function (the fifth one) called `lookahead`.

It can take on the form of two different values: `barmerge.lookahead_off` (and this is the default for Pine Script™ version 3) or `barmerge.lookahead_on` (which is the default for Pine Script™ version 2).

### 9.3.2 Self-referenced variables are removed

Pine Script™ version 2 pieces of code, containing a self-referencing variable:

```
1 // @version=2
2 // ...
3 s = nz(s[1]) + close
```

Compiling this piece of code with Pine Script™ version 3 will give you an `Undeclared identifier 's'` error. It should be rewritten as:

```
1 // @version=3
2 // ...
3 s = 0.0
4 s := nz(s[1]) + close
```

`s` is now a *mutable variable* that is initialized at line 3. At line 3 the initial value gives the Pine Script™ compiler the information about the variable type. It's a float in this example.

In some cases you may initialize that mutable variable (like `s`) with a `na` value. But in complex cases that won't work.

### 9.3.3 Forward-referenced variables are removed

```

1 // @version=2
2 //...
3 d = nz(f[1])
4 e = d + 1
5 f = e + close

```

In this example `f` is a forward-referencing variable, because it's referenced at line 3 before it was declared and initialized. In Pine Script™ version 3 this will give you an error `Undeclared identifier 'f'`. This example should be rewritten in Pine Script™ version 3 as follows:

```

1 // @version=3
2 //...
3 f = 0.0
4 d = nz(f[1])
5 e = d + 1
6 f := e + close

```

### 9.3.4 Resolving a problem with a mutable variable in a security expression

When you migrate script to version 3 it's possible that after removing self-referencing and forward-referencing variables the Pine Script™ compiler will give you an error:

```

1 // @version=3
2 //...
3 s = 0.0
4 s := nz(s[1]) + close
5 t = security(tickerid, period, s)

```

Cannot use mutable variable as an argument for security function!

This limitation exists since mutable variables were introduced in Pine Script™, i.e., in version 2. It can be resolved as before: wrap the code with a mutable variable in a function:

```

1 // @version=3
2 //...
3 calcS() =>
4     s = 0.0
5     s := nz(s[1]) + close
6 t = security(tickerid, period, calcS())

```

### 9.3.5 Math operations with booleans are forbidden

In Pine Script™ v2 there were rules of implicit conversion of booleans into numeric types. In v3 this is forbidden. There is a conversion of numeric types into booleans instead (0 and `na` values are `false`, all the other numbers are `true`). Example (In v2 this code compiles fine):

```

1 // @version=2
2 study("My Script")
3 s = close >= open
4 s1 = close[1] >= open[1]
5 s2 = close[2] >= open[2]
6 sum = s + s1 + s2

```

(continues on next page)

(continued from previous page)

```

7 col = sum == 1 ? white : sum == 2 ? blue : sum == 3 ? red : na
8 bgcolor(col)

```

Variables `s`, `s1` and `s2` are of `bool` type. But at line 6 we add three of them and store the result in a variable `sum`. `sum` is a number, since we cannot add booleans. Booleans were implicitly converted to numbers (`true` values to `1.0` and `false` to `0.0`) and then they were added.

This approach leads to unintentional errors in more complicated scripts. That's why we no longer allow implicit conversion of booleans to numbers.

If you try to compile this example as a Pine Script™ v3 code, you'll get an error: `Cannot call `operator +` with arguments (series_bool, series_bool); <...>` It means that you cannot use the addition operator with boolean values. To make this example work in Pine Script™ v3 you can do the following:

```

1 //@version=3
2 study("My Script")
3 bton(b) =>
4     b ? 1 : 0
5 s = close >= open
6 s1 = close[1] >= open[1]
7 s2 = close[2] >= open[2]
8 sum = bton(s) + bton(s1) + bton(s2)
9 col = sum == 1 ? white : sum == 2 ? blue : sum == 3 ? red : na
10 bgcolor(col)

```

Function `bton` (abbreviation of boolean-to-number) explicitly converts any boolean value to a number if you really need this.





## WHERE CAN I GET MORE INFORMATION?

- *External resources*
- *Download this manual*

- A description of all the Pine Script™ operators, variables and functions can be found in the [Reference Manual](#).
- Use the code from one of TradingView's built-in scripts to start from. Open a new chart and click the "Pine Editor" button on the toolbar. Once in the editor window, click the "Open" button, then select "Built-in script..." from the dropdown list to open a dialog box containing a list of TradingView's built-in scripts.
- There is a TradingView public chat dedicated to Pine Script™ [Q&A](#) where active developers of our community help each other out.
- Information about major releases and modifications to Pine Script™ (as well as other features) is regularly published on [TradingView's blog](#).
- TradingView's [Community Scripts](#) contain all user-published scripts. They can also be accessed from charts using the "Indicators & Strategies" button and the "Community Scripts" tab of the script searching dialog box.

### 10.1 External resources

- The [PineCoders](#) account on TradingView publishes useful information for Pine Script™ programmers. They also have content on their [website](#).
- [Kodify](#) has TradingView tutorials on various topics for beginners and more experienced programmers alike. Topics include plotting, alerts, strategy orders, and complete example indicators and strategies.
- [Backtest Rookies](#) publishes good quality blog articles focusing on realizing specific tasks in Pine Script™.
- You can ask questions about programming in Pine Script™ in the `[pine-script]` tag on [StackOverflow](#).

### 10.2 Download this manual

Available versions:

- PDF

