ProRealTime

PROGRAMMING GUIDE

Indicators & Basic Functions (ProBuilder)

V 6.1.0 – 20251201

Table of contents

Introduction to ProBuilder

ProBuilder is ProRealTime's programming language. It allows you to create personalised technical indicators, trading strategies (ProBacktest) or screening programs (ProScreener). Two specific manuals exist for ProBacktest and ProScreener due to some specific characteristics of each of these modules.

ProBuilder is a BASIC-type programming language, very easy to handle and exhaustive in terms of available possibilities.

You will be able to create your own programs using the quotes from any instrument provided by ProRealTime. Some basic available elements include:

- Opening of each bar: `Open`
- Closing of each bar: `Close`
- Highest price of each bar: `High`
- Lowest price of each bar: `Low`
- Volume of each bar: `Volume`

Bars or candlesticks are the common charting representations of real-time quotes. Of course, ProRealTime offers you the possibility of personalizing the style of the chart. You can use Renko, Kagi, Heikin-Ashi and many other styles.

ProBuilder evaluates the data of each price bar starting from the oldest bar to the most recent one, and then executes the formula developed in the language in order to determine the value of the indicators on the current bar.

The indicators coded in ProBuilder can be displayed either in the price chart or in their own charts.

In this document, you will learn, step by step, how to use the available commands necessary to program in this language thanks to a clear theoretical overview and concrete examples.

At the end of the manual, you will find a Glossary which will give you an overall view of all the ProBuilder commands, pre-existing indicators and other functions completing what you would have learned after reading the previous parts.

Users more confident in their programming skills can skip directly to chapter II or just refer to the Glossary to quickly find the information they want.

For those who are less confident, we recommend watching our video tutorial entitled "**Programming simple and dynamic indicators**" and reading the whole manual.

If you have any questions about ProBuilder, you can ask our ProRealTime community on the **ProRealCode forum**, where you will also find **online documentation** with many examples.

Your dedicated ProRealTime account manager can also help you answer such questions. Feel free to ask.

We wish you success and hope you will enjoy the manual!

The ProRealTime team

Chapter I: Fundamentals

Using ProBuilder

Indicator creation quick tutorial

The indicator programming area is available from the "Indicators" button on each chart in your ProRealTime platform or from the menu Display > Indicators/Backtest.

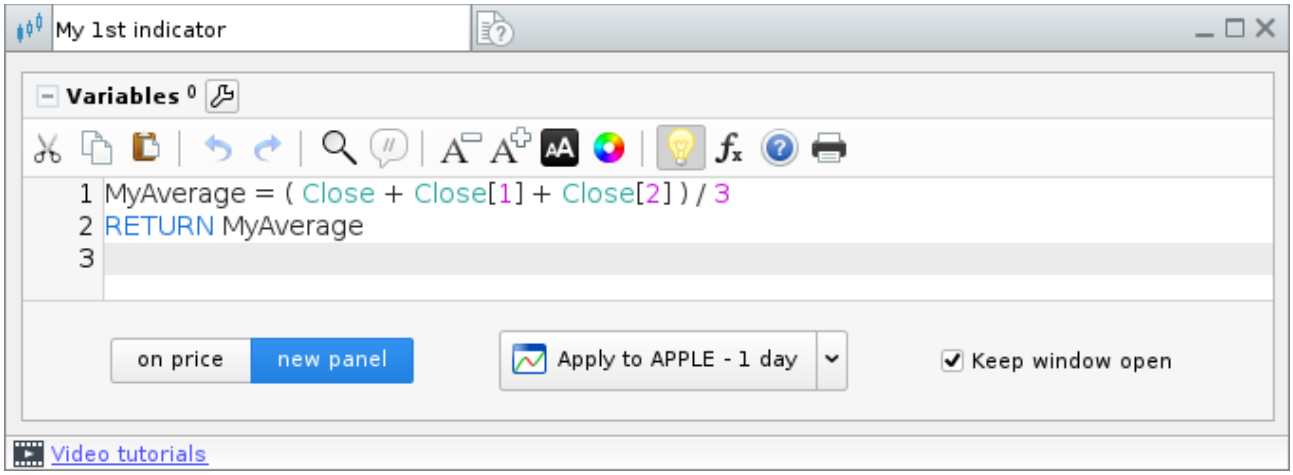
The indicators management window will be displayed. You will then be able to:

- Display a pre-existing indicator
- Create a personalised indicator, which can be used afterwards on any security

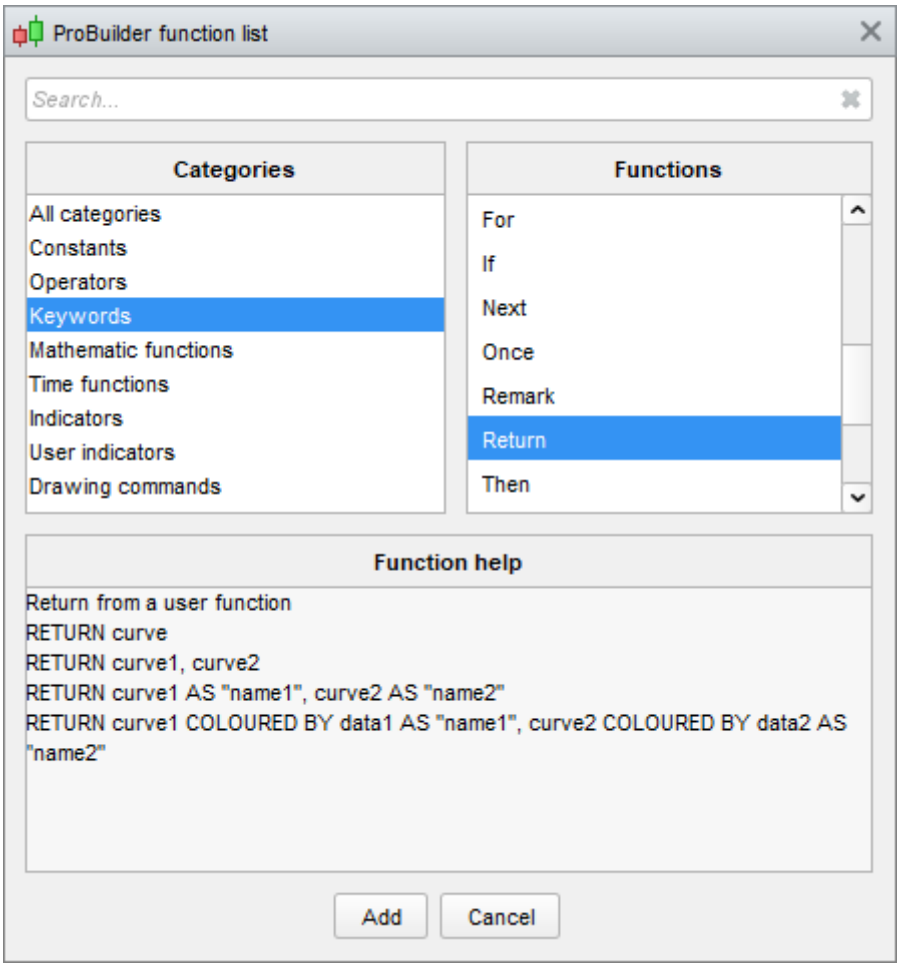
If you choose the second possibility, click on "Create" to access the programming window.

At that time, you will be able to choose between:

- Programming the indicator directly in the text zone designed for writing code or
- Using the help function by clicking on "Insert Function" (f_x icon). This will open a new window in which you can find all the functions available. This library is divided in 8 categories, to give you constant assistance while programming.



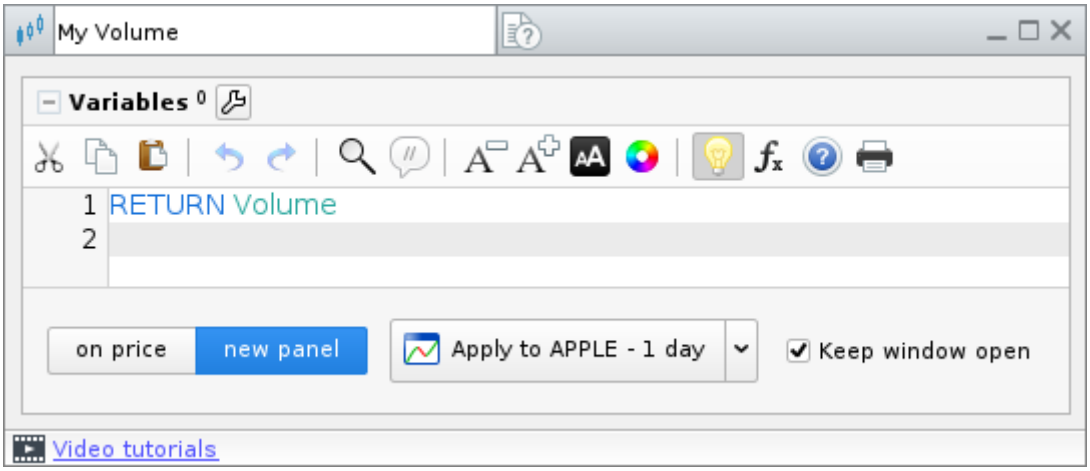
Let's take for example the first ProBuilder key element: the "**RETURN**" function, available in the "ProBuilder function list" f_x (see the image below).



Select the word "**RETURN**" and click on "Add". The command will be added to the programming zone.

RETURN allows you to display the result

Suppose we want to create an indicator displaying the Volume. If you have already inserted the function "**RETURN**", then you just need to click one more time on "Insert Function". Next, click on "Constants" in the "Categories" section, then in the right side of the window, in the section named "Functions", click on "**Volume**". Finally, click on "Add". Don't forget to add a space in between each instruction as shown below.



Before clicking on the "Apply" button, specify at the top of the window the name of your indicator. Finally, click on "Apply" and you will see the chart with your indicator.

Programming window keyboard shortcuts

The programming window has a number of useful features that can be accessed by keyboard shortcuts:

- Select all (Ctrl + A): Select all text in the programming window
- Copy (Ctrl + C): Copy the selected text
- Cut (Ctrl + X): Cut the selected text
- Paste (Ctrl + V): Paste copied text
- Undo (Ctrl + Z): Undo the last action in the programming window
- Redo (Ctrl + Y): Redo the last action in the programming window
- Find / Replace (Ctrl + F): Find a text in the programming window / replace a text in the programming window
- Comment / Uncomment (Ctrl + R): Comment the selected code / Uncomment the selected code (commented code will be preceded by "//" and colored grey. It will not be taken into account when the code is executed).
- Auto-complete (Ctrl + Space): Allows you to display suggested instructions or keywords

For Mac users, the same keyboard shortcuts can be accessed with the "Command" key in place of the "Ctrl" key.

Most of these features can also be accessed by right-clicking in the programming window.

Specificities of ProBuilder programming language

Specificities

The ProBuilder language allows you to use many classic commands as well as sophisticated tools which are specific to technical analysis. These commands will be used to program from simple to very complex indicators.

The main ideas to know in the ProBuilder language are:

- It is **not necessary to declare variables**
- It is **not necessary to type variables**
- There is **no difference between capital letters and small letters**
- **We use the same symbol "=" for mathematical equality and to attribute a value to a variable**

What does this mean?

- Declaring a variable X means indicating its existence. In ProBuilder, you can directly use X without having to declare it. Let's take an example:

  With declaration: let X be a variable and assign the value 5 to X.

  Without declaration: We attribute to X the value 5 (therefore, implicitly, X exists and the value 5 is attributed to it)

  In ProBuilder, you just need to write: X=5

- Typing a variable means defining its nature. For example: is the variable an integer (e.g.: 3; 8; 21; 643; …), a decimal number (e.g.: 1.76453534535…), a boolean (RIGHT=1, WRONG=0),…

- In ProBuilder, you can write your command with capital letters or small letters. For example, the group of commands `IF` / `THEN` / `ELSE` / `ENDIF` can be written `iF` / `tHeN` / `ELse` / `endIf`

- Assigning a value to a variable means giving the variable a value. In order to understand this principle, you must think of a variable as an empty box which you can fill with an expression (ex: a number). The following diagram illustrates the Assignment Rule with the Volume value assigned to the variable X:

X ⟵ Volume

As you can see, we must read from right to left: Volume is assigned to X.

If you want to write it under ProBuilder, you just need to replace the arrow with an equal sign:

X = Volume

The same = symbol is used:

- For the assignment of a variable (like the previous example)
- As the mathematical comparison operator (1+ 1= 2 is equivalent to 2 = 1 + 1).

The Execution Model

Unlike traditional programming languages that run once and then stop, ProBuilder executes once per candlestick, starting with the oldest candlestick.

When it reaches the candlestick that is currently being formed, the behaviour changes depending on the ProBuilder engine in use (Indicator, ProBacktest, ProOrder AutoTrading, ProScreener):

- **Indicator**: the code is re-evaluated on every tick.
- **ProBacktest** and **ProOrder**: the code is evaluated at candlestick close.
- **ProScreener**: the code is re-run from the very first candlestick as soon as the market has been fully scanned. This keeps the ProScreener window permanently up-to-date.

Variables

In classic programming languages, a variable holds a single value. In ProBuilder, variables work differently: **they store a history of their values for every candlestick encountered**. The history therefore grows as the code moves forward through the historical candlesticks.

Every value in the history is instantly accessible using the **`[n]`** operator, which returns the value n candlesticks ago relative to the current candlestick. You can also fetch the n-th value in the history with **`[BarIndex+1-n]`**:

- `MyVariable[3]` → value of MyVariable 3 candlesticks before the current one.
- `MyVariable[BarIndex-2]` → value of MyVariable at the 3rd candlestick in the history.

Although ProBuilder variables look like lists from other languages, their behaviour is quite different:

- You cannot modify the past: `MyVariable[3] = 42` is invalid. The history can be read but never altered.
- Only the value for the current candlestick can be changed: MyVariable = `42` is valid.
- If a variable is not assigned on the current candlestick, it automatically keeps the value it had on the previous candlestick – as if the following line were silently inserted at the top of the code for every variable: `MyVariable = MyVariable[1]`.

There are also array-type variables (`$MyArray[MyIndex]`) which will be covered in detail later in this manual. Unlike classic ProBuilder variables, **arrays are not historised**. The operation `$MyArray[MyIndex][3]` is **invalid**; it does **not** retrieve the value of `$MyArray[MyIndex]` from 3 candlesticks ago.

 It is essential to have a solid grasp of the execution model and the concept of historised variables in order to unlock the full power of ProBuilder!

Financial constants

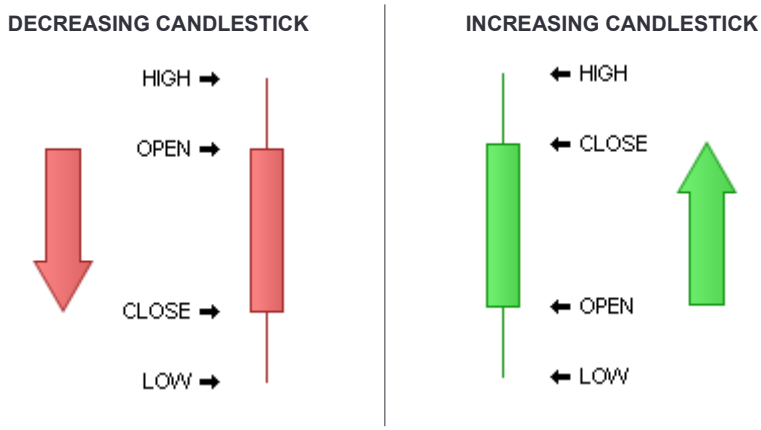
Before coding your personal indicators, you must examine the elements you need to write your code such as the opening price, the closing price, etc.
These are the "fundamentals" of technical analysis and the main things to know for coding indicators.

You will then be able to combine them in order to draw out some information provided by financial markets. We can group them together in 5 categories:

Price and volume constants adapted to the timeframe of the chart

These are the "classical" constants and also the ones used the most. They report by default the value of the current bar (whatever the timeframe used).

- **Open**: Opening price of the current bar.
- **High**: Highest price of the current bar.
- **Low**: Lowest price of the current bar.
- **Close**: Closing price of the current bar.
- **Volume**: The number of securities or contracts exchanged during the current bar.



Example: Range of the current bar

```
a = High
b = Low
MyRange = a - b
RETURN MyRange // The "Range" constant also exists to simplify access to this value.
```

If you want to use the information of previous bars rather than the current bar, you just need to add between square brackets the number of bars that you want to go back into the past.

Let's take for example the closing price constant. Calling the price is done in the following way:
Value of the closing price of the current bar: `Close`
Value of the closing price of the bar preceding the current bar: `Close[1]`
Value of the closing price of the nth bar preceding the current one: `Close[n]`

This rule is valid for any constant or variable. For example, the opening price of the 2nd bar preceding the current can be expressed as: `Open[2]`.
The reported value will depend on the displayed timeframe of the chart.

Daily price constants

Contrary to the constants adapted to the timeframe of the chart, the daily price constants refer to the value of the day, regardless of the timeframe of the chart.

Another difference between Daily price constants and constants adapted to the timeframe of the chart is that the daily price constants use parentheses and not square brackets to call the values of previous days.

- **DOpen(n):** Opening price of the n-th day before the one of the current bar
- **DHigh(n):** Highest price of the n-th day before the one of the current bar
- **DLow(n):** Lowest price of the n-th day before the one of the current bar
- **DClose(n):** Closing price of the n-th day before the one of the current bar

Note: if "n" is equal to 0, "n" refers to the current day. The maximum and minimum values are not yet definitive for n=0, we will obtain results which can change during the day depending on the minimum and maximum reached by the value.

The constants adapted to the timeframe of the chart use square brackets while the daily price constants use parentheses.

`Close[3]` ➡ The closing price 3 periods ago

`DClose(3)` ➡ The closing price 3 days ago

Temporal constants

Time is often a neglected component of technical analysis. However, traders know very well the importance of some time periods in the day or dates in the year. It is possible in your programs to take into account time and date and improve the efficiency of your indicators. The Temporal constants are described hereafter:

- **Date:** indicates the date of the close of each bar in the format YearMonthDay (YYYYMMDD)

Temporal constants are considered by ProBuilder as whole numbers. The `Date` constant, for example, must be used as one number made up of 8 figures.

Let's write down the program:

```
RETURN Date
```

Suppose today is July 4th, 2020. The program above will return the result 20200704.

The date can be read in the following way:

20200704 = 2020 years 07 months and 04 days.

Note that when writing a date in the format YYYYMMDD, MM must be between 01 and 12 and DD must be between 01 and 31.

- **Time:** indicates the time of closing of each bar in the format HHMMSS (HourMinuteSecond)

Example:

```
RETURN Time
```

This indicator shows us the closing time of each bar in the format HHMMSS:

Time can be read as follows:

160000 = 16 hours, 00 minutes, and 00 seconds.

Note that when writing a time in the format HHMMSS, HH must be between 00 and 23, MM must be between 00 and 59 and SS must be also between 00 and 59.

It is also possible to use **Time** and **Date** in the same indicator to do analysis or display results at a precise moment. In the following example, we want to limit our indicator to the date of October 1st 2025 at precisely 9am and 1 second:

```
a = (Date = 20251001)
b = (Time = 090001)
RETURN (a AND b)
```

The following constants work the same way:

- **Timestamp**: UNIX date and time (number of seconds since January 1st, 1970) of the close of each bar.
- **Second**: Second of the close of each bar (between 0 and 59).
- **Minute**: Minute of the close of each bar (from 0 to 59): Only for intraday charts.
- **Hour**: Hour of the close of each bar (from 0 to 23): Only for intraday charts.
- **Day**: Day of the month of the closing price of each bar (from 1 to 28 or 29 or 30 or 31)
- **Month**: Month of the closing price of each bar (from 1 to 12)
- **Year**: Year of the closing price of each bar
- **DayOfWeek**: Day of the Week of the close of each bar (0=Sunday, 1=Monday, 2=Tuesday, 3=Wednesday, 4=Thursday, 5=Friday, 6=Saturday)

Derivative constants also exist for **Open:**

- **OpenTimestamp**: UNIX date and time of the open of each bar.
- **OpenSecond**: Second of the open of each bar (between 0 and 59).
- **OpenMinute**: Minute of the open of each bar (between 0 and 59).
- **OpenHour**: Hour of the open of each bar (between 0 and 23).
- **OpenDay**: Day of the month of the open of each bar (between 1 and 28 or 29 or 30 or 31).
- **OpenMonth**: Month of the open of each bar (between 1 and 12).
- **OpenYear**: Year of the open of each bar.
- **OpenDayOfWeek**: Day of the week at the open of each bar (0=Sunday, 1=Monday, 2=Tuesday, 3=Wednesday, 4=Thursday, 5=Friday, 6=Saturday).
- **OpenTime**: HourMinuteSecond encoded as HHMMSS indicating the opening time of each bar.
- **OpenDate**: Date (YYYYMMDD) of the open of each bar.

Example of the use of these constants:

```
RETURN (Hour > 17) AND (Day = 30)
```

- **CurrentHour**: Current Hour (of the local market).
- **CurrentMinute**: Current Minute (of the local market).
- **CurrentMonth**: Current Month (of the local market).
- **CurrentSecond**: Current Second (of the local market).
- **CurrentTime**: Current HourMinuteSecond (of the local market).
- **CurrentYear**: Current Year (of the local market).
- **CurrentDayOfWeek**: Current Day of the week with the market time zone as a reference.

The following picture brings to light that difference (applied on the **CurrentTime** and **Time** constants). We can highlight the fact that for "Current" constants, we must set aside the time axis and only take in consideration the displayed value (the value of the current time is displayed over the whole history of the chart).

Time indicates the closing time of each bar.

CurrentTime indicates the current market time.

If you want to set up your indicators with counters (number of days passed, number of bars passed etc…), you can use the **Days**, **BarIndex** and **IntradayBarIndex** constants.

- **Days**: Counter of days since 1900

This constant is quite useful when you want to know the number of days that have passed. It is particularly relevant when you work with an (x) tick or (x) volume view.

The following example shows the number of days passed since 1900.

```
RETURN Days
```

(Be careful not to confuse the constants "**Day**" and "**Days**").

- **BarIndex**: Counter of bars since the beginning of the displayed historical data

The counter starts from left to right and counts each bar, including the current bar. The first bar loaded is considered bar number 0. Most of the time, **BarIndex** is used with the **IF** instruction presented later in the manual.

- **IntradayBarIndex**: Counter of intraday bars

The counter displays the number of bars since the beginning of the day and then resets to zero at the beginning of every new day. The first bar of the counter is considered bar number 0.

Let's compare the two counter constants with two separate indicators:

```
RETURN BarIndex
```

and

```
RETURN IntradayBarIndex
```

We can clearly see the difference between them: **IntradayBarIndex** resets itself to zero at the beginning of every new day.

Constants derived from price

These constants allow you to get more complete information compared to **Open**, **High**, **Low** and **Close**, since they combine those prices so as to emphasize some aspects of the financial market psychology shown on the current bar.

- **Range**: difference between **High** and **Low**.
  - `Range = High - Low`
- **TypicalPrice**: average between **High**, **Low** and **Close**
  - `TypicalPrice = (High + Low + Close) / 3`
- **WeightedClose**: weighted average of **High**, **Low** and **Close**
  - `WeightedClose = (High + Low + 2 * Close) / 4`
- **MedianPrice**: average between **High** and **Low**
  - `MedianPrice = (High + Low) / 2`
- **TotalPrice**: average between **Open**, **High**, **Low** and **Close**
  - `TotalPrice = (Open + High + Low + Close) / 4`

Range shows the volatility of the current bar, which is an estimation of how nervous investors are.

WeightedClose focuses on the importance of the closing price.

TypicalPrice and **TotalPrice** emphasize intraday financial market psychology since they take 3 or 4 predominant prices of the current bar into account.

MedianPrice is the median price of the candlestick, calculated by computing the average of the High and Low.

Range in %:

```
pctRange = (Range / Range[1] - 1) * 100
RETURN pctRange
```

The Undefined constant

The keyword **Undefined** allows you to indicate to the software not to display the value of the indicator.

- **Undefined**: undefined data (equivalent to an empty box)

You can find an example later in the manual.

How to use pre-existing indicators?

Up until now, we have described the possibilities offered by ProBuilder concerning constants and how to call values of bars of the past using these constants. Pre-existing indicators (the ones already programmed in ProRealTime) function the same way and so do the indicators you will code.

ProBuilder indicators are made up of three elements whose syntax is:

```
NameOfFunction [calculated over n periods] (applied to which price or indicator)
```

When using the "Insert Function" button to look for a ProBuilder function and then enter it into your program, default values are given for both the period and the price or indicator argument. Example for a moving average of 20 periods:

```
Average[20](Close)
```

The values can be modified. For example, we can replace the 20 bars defined by default with any number of bars (ex: `Average[10]`, `Average[15]`, `Average[30]`, …, `Average[n]`). In the same way, we can replace "**`Close`**" with "**`Open`**" or **`RSI`** **(Relative strength index)**. This would give us for example:

```
Average[20](RSI[5](Close))
```

In this case the average is calculated on the last 20 candles of the RSI applied to the last 5 closing prices.

Here are some sample programs:

Program calculating the exponential moving average over 20 periods applied to the closing price:

```
RETURN ExponentialAverage[20](Close)
```

Program calculating the weighted moving average over 20 bars applied to the typical price

```
RETURN WeightedAverage[20](TypicalPrice)
```

Program calculating the Wilder average over 100 candlesticks applied to the Volume

```
RETURN WilderAverage[100](Volume)
```

Program calculating the MACD (histogram) applied to the closing price.

The MACD is built with the difference between the 12-period exponential moving average (EMA) minus the 26-period EMA. Then, we smooth it with an exponential moving average over 9 periods and applied to the MACD line to get the Signal line. Finally, the MACD is the difference between the MACD line and the Signal line.

```
// Calculation of the MACD line
LineMACD = ExponentialAverage[12](Close) - ExponentialAverage[26](Close)
// Calculation of the MACD Signal line
SignalMACD = ExponentialAverage[9](LineMACD)
// Calculation of the difference between the MACD line and its Signal
MACDHistogram = LineMACD - SignalMACD
RETURN MACDHistogram STYLE (HISTOGRAM)
```

Calculation of an average with two parameters

You also have the possibility to use a second parameter with the **Average** function (which indicates the type of average to use). We obtain the following formula :

```
Average [Nbr. of periods, Type of average]
```

The parameter Type of average designates, as its name indicates, the type of average that will be used. There are 9 of them and they are indexed from 0 to 8:

0=Single	4=Triangular	8=Zero delay
1=Exponential	5=Least Squares	
2=Weighted	6=Time series	
3=Wilder	7=Hull	

Calculation of Ichimoku lines

Since the Ichimoku indicator includes many lines, some of these lines have been introduced separately in the ProBuilder language to allow you to get the most out of this indicator.

The lines as follows:

- `TenkanSen[TenkanPeriod,KijunPeriod,Senkou-SpanBPeriod]`
- `KijunSen[TenkanPeriod,KijunPeriod,Senkou-SpanBPeriod]`
- `SenkouSpanA[TenkanPeriod,KijunPeriod,Senkou-SpanBPeriod]`
- `SenkouSpanB[TenkanPeriod,KijunPeriod,Senkou-SpanBPeriod]`

With for each line the usual Ichimoku parameters:

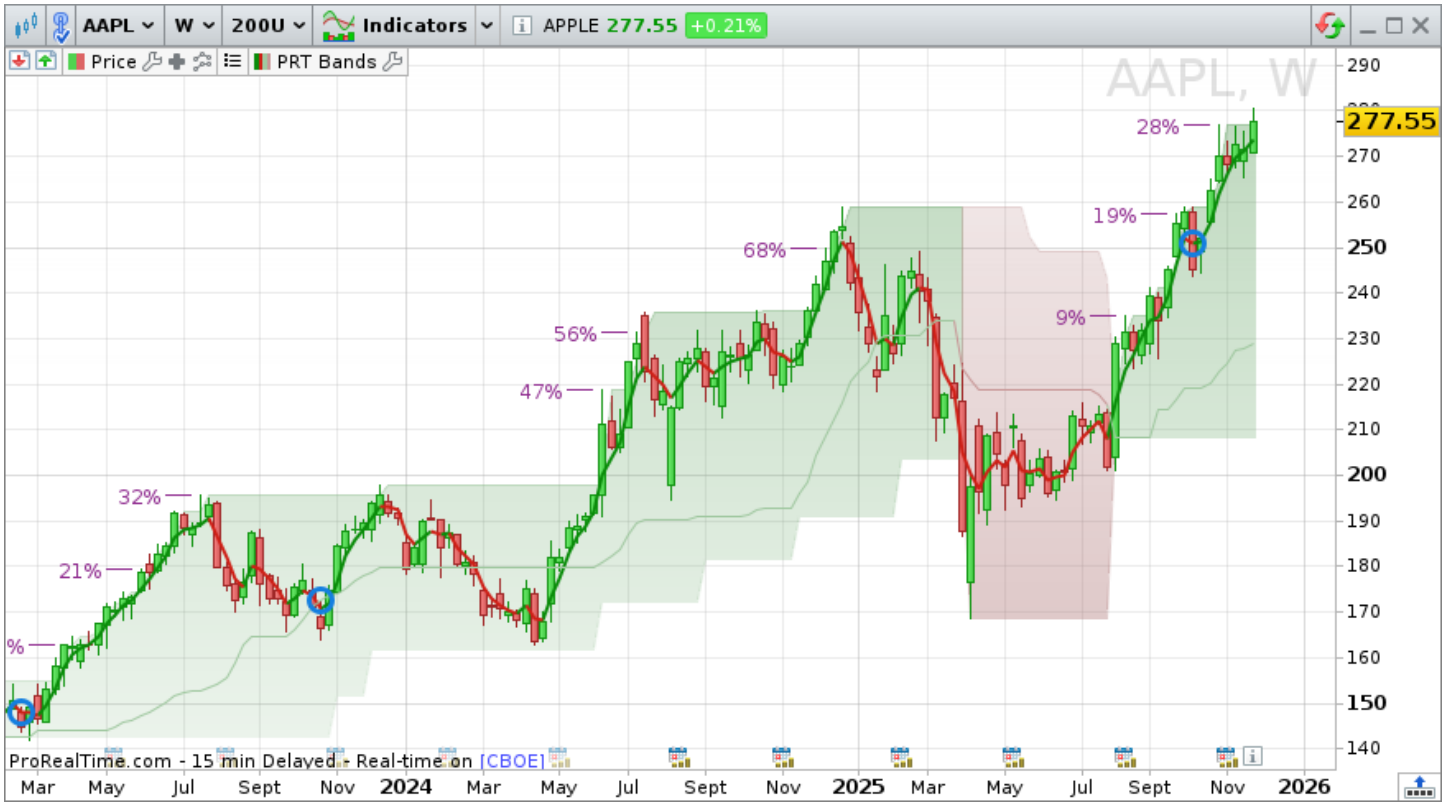
- `TenkanPeriod`: alert line, (high point + low point)/2 over the last ***n*** periods
- `KijunPeriod`: signal line, (high point + low point)/2 over the last ***n*** periods
- `Senkou-SpanBPeriod`: long-term average point projection, (high point + low point)/2 over the last ***n*** periods

Calculation of PRT Bands

PRT Bands is a visual indicator that simplifies the detection and monitoring of trends. It is exclusive to the ProRealTime platform.

It can help you to:

- detect a reversal of trends
- identify and follow an upward trend
- measure the intensity of the trend
- find potential entry and exit points



Here are the different PRT Bands data available in the ProBuilder language:

- **PRTBANDSUP**: returns the value of the top line of the indicator
- **PRTBANDSDOWN**: returns the value of the bottom line of the indicator
- **PRTBANDSSHORTTERM**: returns the value of the short-term (thick) line of the indicator
- **PRTBANDSMEDIUMTERM**: returns the value of the medium-term (thin) line of the indicator

Learn more about the PRT Bands indicator

Adding customizable variables

When you code an indicator, you may want to use customizable variables. The variables option in the upper-left corner of the window allows you to assign a default value to an undefined variable in your program and change its value in the settings window of the indicator without modifying the code of your program.

Let's calculate a simple moving average on 20 periods:

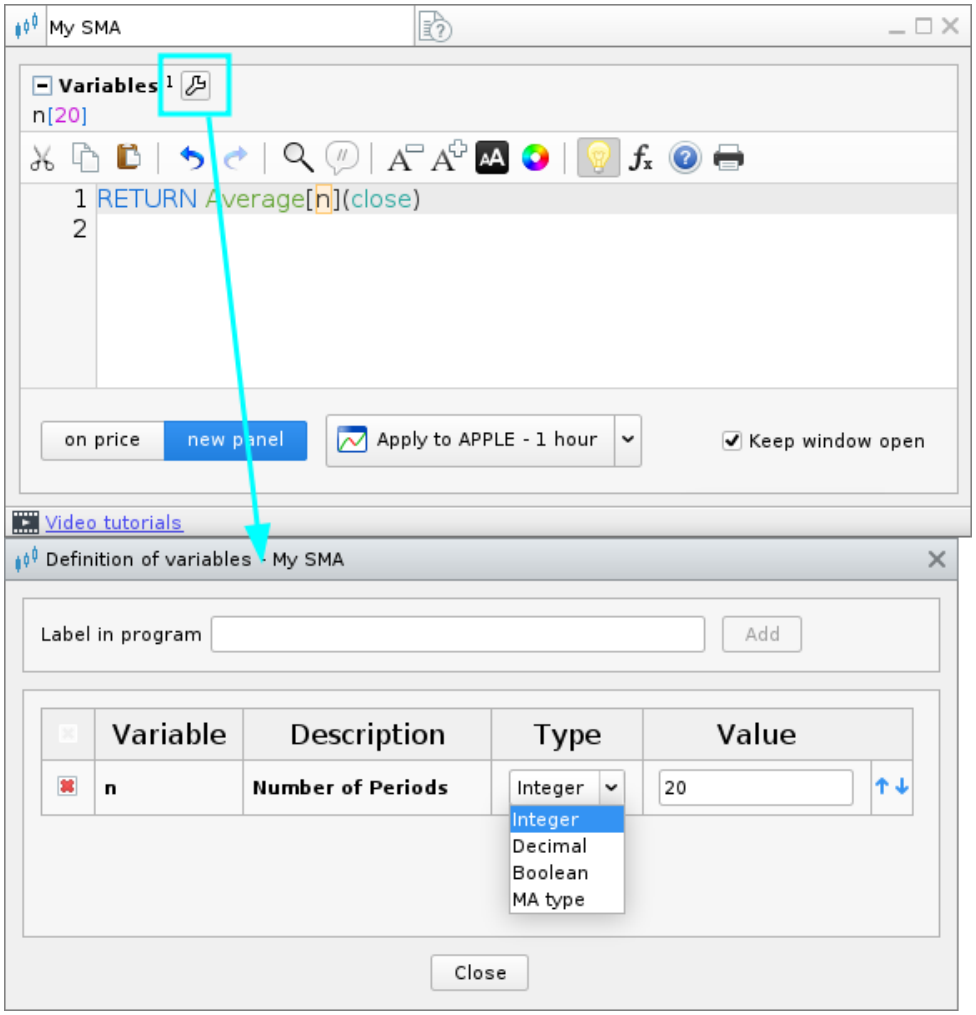
```
RETURN Average[20](Close)
```

In order to modify the number of periods for the calculation directly from the indicator "Settings" interface, replace 20 with the variable "n":

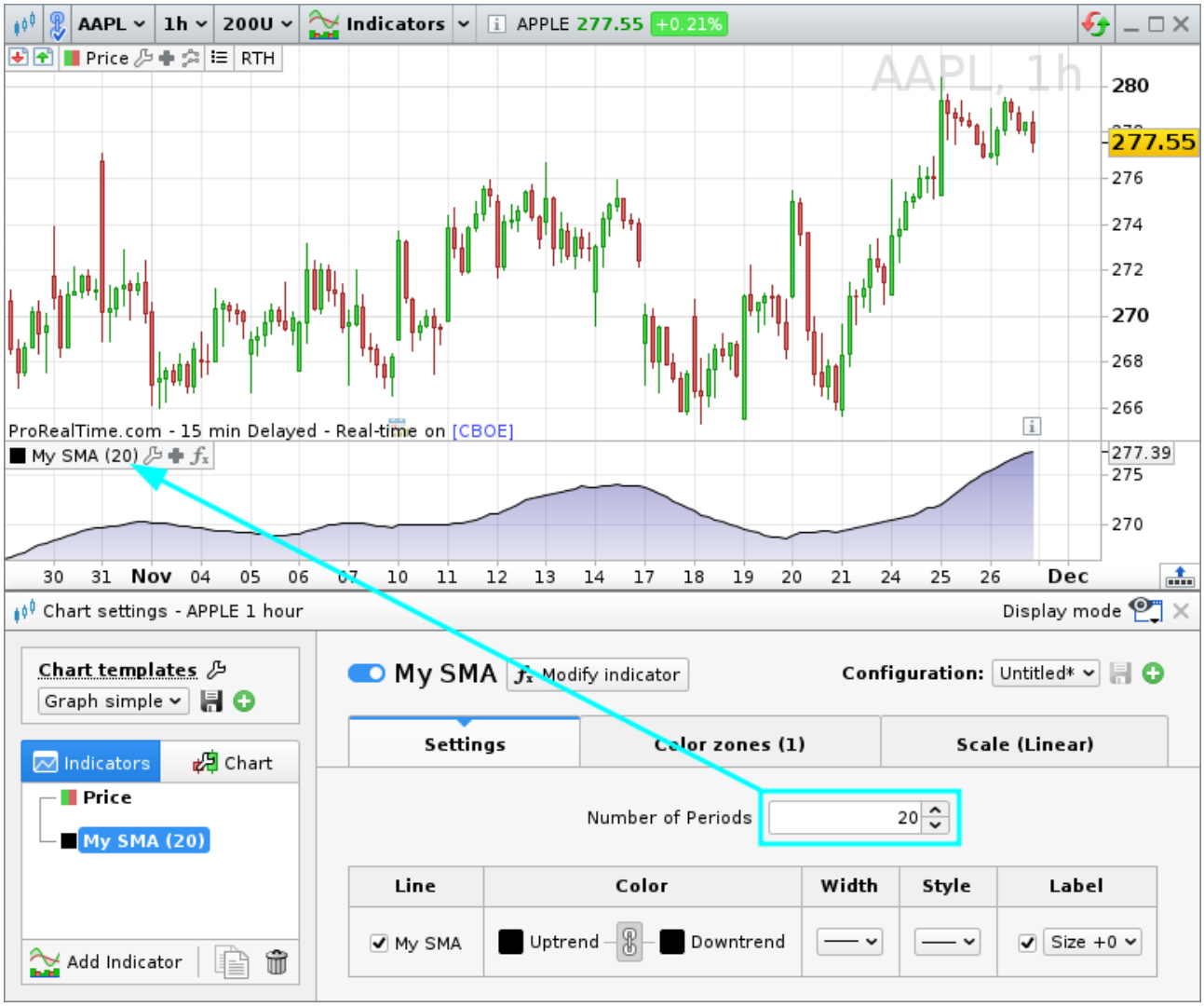
```
RETURN Average[n](Close)
```

Then, click on the wrench button next to "Variables" and another window named "Variable definition" will be displayed.

Enter the name of your variable, here "n", and press "Enter" (or click "Add"). You can then fill in a description to be displayed in the indicator's property window, a Type and a Default Value as shown in the example below:



In the "Settings" tab you will see a new parameter which will allow you to modify the number of periods used in the calculation of the moving average:



List of available types for variables:

- **Integer:** integer between -2,000,000,000 and 2,000,000,000 (ex: 450)
- **Decimal:** decimal number with a precision of 5 significant digits (ex: 1.03247)
- **Boolean:** True (**1**) or False (**0**)
- **Moving Average Type:** Allows you to set the value of the second parameter which defines the type of moving average used in the calculation of the `Average` indicator (see above).

Of course, it is possible to create many variables giving you the possibility to manipulate multiple parameters at the same time.

Chapter II: Math Functions and ProBuilder instructions

Control Structures

Conditional IF instruction

The **IF** instruction is used to make a choice of conditional actions, i.e. to make a result dependent on the verification of one or more defined conditions.

The structure is made up of the instructions **IF**, **THEN**, **ELSE**, **ELSIF**, **ENDIF**, which are used depending on the complexity of the conditions you defined.

One condition, one result (IF THEN ENDIF)

We can look for a condition and define an action if that condition is true. On the other hand, if the condition is not valid, then nothing will happen.

In this example, if the current price is greater than the 20-period moving average, then we display “Result = 1” on the chart.

```
Result = 0
IF Close > Average[20](Close) THEN
   Result = 1
ENDIF
RETURN Result
```

Result is equal to 0.
IF closing price > 20-period moving average
THEN Result = 1
OTHERWISE Result is unchanged
END OF CONDITION

One condition, two results (IF THEN ELSE ENDIF)

We can also define a different result if the condition is not true. Let us go back to the previous example: if the price is greater than the moving average on 20 periods, then display 1, else, displays -1.

```
IF Close > Average[20](Close) THEN
   Result = 1
ELSE
   Result = -1
ENDIF
RETURN Result
```

NB: We have created a binary indicator. For more information, see the section on binary and ternary indicators later in this manual.

Sequential IF conditions

You can create sub-conditions after the validation of the main condition, meaning conditions which must be validated one after another. For that, you need to build a sequence of **IF** structures, one included in the other. You should be careful to insert in the code as many **ENDIF** as **IF**. Example:

Double conditions on moving averages:

```
IF Average[12](Close) > Average[20](Close) THEN
   IF ExponentialAverage[12](Close) > ExponentialAverage[20](Close) THEN
      Result = 1
   ELSE
      Result = -1
   ENDIF
ENDIF
RETURN Result
```

Multiple conditions (IF THEN ELSE ELSIF ENDIF)

You can define a specific result for a specific condition. The indicator reports many states: if Condition 1 is valid then do Action1; else, if Condition 2 is valid, then do Action 2 …if none of the previously mentioned conditions are valid then do Action n.

This structure uses the following instructions: **IF**, **THEN**, **ELSIF**, **THEN**.... **ELSE**, **ENDIF**.

The syntax is:

```
IF (Condition1) THEN
   (Action1)
ELSIF (Condition2) THEN
   (Action2)
ELSIF (Condition3) THEN
   (Action3)
...
ELSE
   (Action n)
ENDIF
```

You can also replace **ELSIF** with **ELSE IF** but your program will take longer to write. Of course, you will have to end the loop with as many instance of **ENDIF** as **IF**. If you want to make multiple conditions in your program, we advise you to use **ELSIF** rather than **ELSE IF** for this reason.

Example: detection of bearish and bullish engulfing lines using the **Elsif** instruction

This indicator displays 1 if a bullish engulfing line is detected, -1 if a bearish engulfing line is detected, and 0 if neither of them is detected.

```
// Detection of a bullish engulfing line
Condition1 = Close[1] < Open[1]
Condition2 = Open < Close[1]
Condition3 = Close > Open[1]
Condition4 = Open < Close

// Detection of a bearish engulfing line
Condition5 = Close[1] > Open[1]
Condition6 = Close < Open
Condition7 = Open > Close[1]
Condition8 = Close < Open[1]

IF Condition1 AND Condition2 AND Condition3 AND Condition4 THEN
   a = 1
ELSIF Condition5 AND Condition6 AND Condition7 AND Condition8 THEN
   a = -1
ELSE
   a = 0
ENDIF
RETURN a
```

Example: Resistance Demark pivot

```
IF DClose(1) > DOpen(1) THEN
   Phigh = DHigh(1) + (DClose(1) - DLow(1)) / 2
   Plow = (DClose(1) + DLow(1)) / 2
ELSIF DClose(1) < DOpen(1) THEN
   Phigh = (DHigh(1) + DClose(1)) / 2
   Plow = DLow(1) - (DHigh(1) - DClose(1)) / 2
ELSE
   Phigh = DClose(1) + (DHigh(1) - DLow(1)) / 2
   Plow = DClose(1) - (DHigh(1) - DLow(1)) / 2
ENDIF
RETURN Phigh , Plow
```

Example: BarIndex

In chapter I of our manual, we presented **BarIndex** as a counter of bars loaded. **BarIndex** is often used with **IF**. For example, if we want to know if the number of bars in your chart exceeds 23 bars, then we will write:

```
IF BarIndex <= 23 THEN
   a = 0
ELSIF BarIndex > 23 THEN
   a = 1
ENDIF
RETURN a
```

Note: This code is shown as an example to describe how the **IF** instruction works. But here is a cleaner and more readable way to reach the same result.

```
RETURN BarIndex > 23
```

Iterative FOR Loop

FOR is used when we want to exploit a finite series of elements. This series must be made up of whole numbers (ex: 1, 2, 3, ..., 6, 7 or 7, 6, ..., 3, 2, 1) and ordered.

Its structure is formed of **FOR**, **TO**, **DOWNTO**, **DO**, **NEXT**. **TO** and **DOWNTO** are used depending on the order of appearance in the series of the elements (ascending order or descending order). We also highlight the fact that what is between **FOR** and **DO** are the extremities of the interval to scan.

Ascending loop (FOR, TO, DO, NEXT)

```
FOR Variable = BeginningValueOfTheSeries TO EndingValueOfTheSeries DO
   (Action)
NEXT
```

Example: Smoothing of a 12-period moving average

Let's create a storage variable (Result) which will sum the 11, 12 and 13-period moving averages.

```
Result = 0
FOR Variable = 11 TO 13 DO
  Result = Result + Average[Variable](Close)
NEXT
// Let's create a storage variable (AverageResult) which will divide Result by 3 and display average result. Average result is a smoothing of the 12-period moving average.
AverageResult = Result / 3
RETURN AverageResult
```

Let's see what is happening step by step:

Mathematically, we want to calculate the average of the arithmetic moving averages of periods 11, 12 and 13.

Variable will thus take successively the values 11, 12 then 13

```
Result = 0
Variable = 11
```

Result receives the value of the previous **Result** + MA11 i.e.: (0) + MA11 = (0 + MA11)

The **NEXT** instruction takes us to the next value of the counter

```
Variable = 12
```

Result receives the value of the previous **Result** + MA12 or: (0 + MA11) + MA12 = (0 + MA11 + MA12)

The **NEXT** instruction takes us to the next value of the counter

```
Variable = 13
```

Result receives the value of the previous **Result** + MA13 or: (0 + MA11 + MA12) + MA13 = (0 + MA11 + MA12 + MA13)

The value 13 is the last value of the counter.

NEXT closes the **FOR** loop because there is no more next value.

Result / 3 is displayed

This code simply means that **Variable** will initially take the value of the beginning of the series, then **Variable** will take the next value (the previous one + 1) and so on until **Variable** exceeds or is equal to the value of the end of the series. Then the loop ends.

Example: Average of the highest value over the last 5 bars

```
SUMhigh = 0
IF BarIndex < 4 THEN
   MAhigh = Undefined
ELSE
   FOR i = 0 TO 4 DO
      SUMhigh = High[i]+SUMhigh
   NEXT
ENDIF
MAhigh = SUMhigh / 5
RETURN MAhigh
```

If there are not yet 5 periods displayed
Then we attribute to MAhigh value "Undefined" (not displayed)
ELSE
FOR values of i between 0 to 4
We sum the 5 last "High" values

We calculate the average for the last 5 periods and store the result in MAhigh
We display MAhigh

Note: This code is shown as an example to describe how the **FOR** loop works. But it is important to remember that you should avoid using **FOR** loops whenever other alternatives exist. For example, this code is much more readable, gives the same result, and is more optimized, it will therefore run significantly quicker:

```
Return Average[5](High)
```

Descending loop (FOR, DOWNTO, DO, NEXT)

The descending loop uses the following instructions: **FOR**, **DOWNTO**, **DO**, **NEXT**.
Its syntax is:

```
FOR Variable = EndingValueOfTheSeries DOWNTO BeginningValueOfTheSeries DO
   (Action)
NEXT
```

Let us go back to the previous example (the 5-period moving average of "High"):
Note that we have just reversed the limits of the scanned interval.

```
SUMhigh = 0
IF BarIndex < 4 THEN
   MAhigh = Undefined
ELSE
   FOR i = 4 DOWNTO 0 DO
      SUMhigh = High[i] + SUMhigh
   NEXT
ENDIF
MAhigh = SUMhigh / 5
RETURN Mahigh
```

Conditional WHILE Loop

WHILE is used to keep doing actions while a condition remains true. You will see that this instruction is very similar to the simple conditional instruction **IF**/**THEN**/**ENDIF**.

This structure uses the following instructions: **WHILE**, (**DO** optional), **WEND** (end **WHILE**). Its syntax is:

```
WHILE (Condition) DO
   (Action 1)
   …
   (Action n)
WEND
```

This code lets you show the number of bars separating the current candlestick from a previous higher candlestick within the limit of 30 periods.

```
i = 1
WHILE high > high[i] and i < 30 DO
   i = i + 1
WEND
RETURN i
```

Example: indicator calculating the number of consecutive increases

```
Increase = Close > Close[1]
Count = 0
WHILE Increase[Count] DO
   Count = Count + 1
WEND
RETURN Count
```

Note: This code is shown as an example to describe how the **WHILE** loop works. But it is important to remember that you should avoid using **WHILE** loops whenever other alternatives exist. For example, the following code is much more optimized, runs faster and gives the same result.It takes advantage of the fact that ProBuilder code is run on each historical candle: incrementing a counter when the condition is verified works as well as looking in the past for each candle but is more efficient.

```
Increase = Close > Close[1]
IF Increase THEN
    Count = Count + 1
ELSE
    Count = 0
ENDIF
RETURN Count
```

BREAK

The **BREAK** instruction allows you to make a forced exit out of a **WHILE** loop or a **FOR** loop. Combinations are possible with the **IF** command, inside a **WHILE** loop or a **FOR** loop.

BREAK with WHILE

When we want to exit a conditional **WHILE** loop, we use **BREAK** in the following way:

```
WHILE (Condition) DO
   (Action)
   IF (ConditionBreak) THEN
       BREAK
   ENDIF
WEND
```

The use of **BREAK** in a **WHILE** loop is only interesting if we want to test an additional condition for which the value can only be known inside the **WHILE** loop. For example, let's look at a stochastic which is only calculated in a bullish trend:

```
ret = 0
Increase = Close > Close[1]
i = 0
WHILE Increase[i] or BarIndex > 0 DO
       i = i + 1
       // If high = low, we exit the loop to avoid a division by zero.
       IF high[i] = low[i] then
          BREAK
       ENDIF
       osc = (close - low) / (high - low)
       ret = AVERAGE[i](osc)
WEND
RETURN ret
```

BREAK with FOR

When we try to get out of an iterative **FOR** loop, without reaching the last value of the series, we use **BREAK** this way.

```
FOR Variable = SeriesStartValue TO SeriesEndValue DO
   (Action)
   BREAK
NEXT
```

Let's take for example an indicator cumulating increases of the volume of the last 19 periods. This indicator will be equal to 0 if the volume decreases.

```
Count = 0
FOR i = 0 TO 19 DO
   IF Volume[i] > Volume[i + 1] THEN
      Count = Count + 1
   ELSE
      BREAK
   ENDIF
NEXT
RETURN Count
```

In this code, if **BREAK** weren't used, the loop would have continued until 19 (last element of the series) even if the condition count is not valid.

With **BREAK**, on the other hand, as soon as the condition is no longer validated, it returns the result.

CONTINUE

The **CONTINUE** instruction is used to finish the current iteration of a **WHILE** or **FOR** loop. This command is often used with **BREAK**, either to leave the loop (**BREAK**) or to stay in the loop (**CONTINUE**).

CONTINUE with WHILE

Let's create a program counting the number of candlesticks whose close and open are greater than those of the candlestick preceding them. If the condition is not valid, then the counter will be reset to 0.

```
Increase = Close > Close[1]
condition = Open > Open[1]
Count = 0
WHILE condition[Count] DO
   IF Increase[Count] THEN
      Count = Count + 1
      CONTINUE
   ENDIF
   BREAK
WEND
RETURN Count
```

When using **CONTINUE**, if the **IF** condition is valid, then the **WHILE** loop is not ended. This allows us to count the number of candlesticks detected with this condition verified. Without the **CONTINUE** instruction, the program would leave the loop, whether the **IF** condition is verified or not. Then, we would not be able to continue counting the number of candlesticks detected and the result would be binary (1, 0).

CONTINUE with FOR

Let's create a program counting the number of candlesticks whose close and open are greater than those of the candlestick preceding them. If the condition is not valid, then the counter will be reset to 0.

```
Increase = Close > Close[1]
Count = 0
FOR i = 1 TO BarIndex DO
   IF Increase[Count] THEN
      Count = Count + 1
      CONTINUE
   ENDIF
BREAK
NEXT
RETURN Count
```

FOR gives you the possibility to test the condition over all the data loaded. When used with **CONTINUE**, if the **IF** condition is validated, then we do not leave the **FOR** loop and resume it with the next value of i. This is how we count the number of patterns detected by this condition.

Without **CONTINUE**, the program would leave the loop, even if the **IF** condition is validated. Then, we would not be able to count the number of patterns detected and the result would be binary (1, 0).

> It is important to make sure to always have a valid exit condition for **FOR** and **WHILE** loops to ensure that your code works properly and avoid infinite loops.

ONCE

The **ONCE** instruction is used to initialize a variable at a certain value "**only once**".

Knowing that for the whole program, the language will read the code for each bar displayed on the chart before returning the result, you must then keep in mind that:

- **ONCE** is processed only on the first encounter of the instruction.
- Subsequent encounters will be ignored.
- **ONCE** can also be used with an **IF**, a **FOR** and a **WHILE** loop.

To fully understand how this command works, you need to perceive how the language processes the code, hence the usefulness of the next example.

These are two programs returnin 0 and 15 respectively and whose only difference is the **ONCE** command added:

Program 1

```
1 Count = 0
2 i = 0
3 IF i <= 5 THEN
4    Count = Count + i
5    i = i + 1
6 ENDIF
7 RETURN Count
```

Program 2

```
1 ONCE Count = 0
2 ONCE i = 0
3 IF i <= 5 THEN
4    Count = Count + i
5    i = i + 1
6 ENDIF
7 RETURN Count
```

Let's see how the language reads the code.

Program 1:

The language will read L1 (Count = 0; i = 0), then L2, L3, L4, L5 and L6 (Count = 0; i = 1), then return to L1 and reread everything exactly the same way. The result displayed is 0 (zero), as after the first reading.

Program 2:

For the first bar, the language will read L1 (Count = 0; i = 0), then L2, L3, L4, L5, L6 (Count = 0; i = 1). When it arrives at the line "**RETURN**", it restarts the loop to calculate the value of the next bar starting from L3 (**the lines with ONCE are processed only one time**), L4, L5, L6 (Count = 1; i = 2), then go back again (Count = 3; i = 3) and so forth to (Count = 15; i = 6). Arrived at this result, the **IF** loop is not processed anymore because the condition is not valid anymore; the only line left to read is L7, hence the result is 15 for the remaining bars loaded.

Mathematical Functions

Common unary and binary Functions

Let's focus now on the Mathematical Functions. You will find in ProBuilder the main functions known in mathematics. Please note that a and b are examples and can be numbers or any other variable in your program.

- **MIN**(a,b): calculates the minimum of *a* and *b*
- **MAX**(a,b): calculates the maximum of *a* and *b*
- **ROUND**(a,n): rounds *a* to the nearest whole number, with a precision of *n* digits after the decimal point
- **ABS**(a): calculates the absolute value of *a*
- **SGN**(a): shows the sign of *a* (1 if positive, -1 if negative)
- **SQUARE**(a): calculates *a* squared
- **SQRT**(a): calculates the square root of *a*
- **LOG**(a): calculates the Neperian logarithm of *a*
- **POW**(a,b): calculates *a* raised to the power of *b*
- **EXP**(a): calculates the exponent of *a*
- **COS**(a) / **SIN**(a) / **TAN**(a): calculates the cosine/sine/tangent of *a* (in degrees)
- **ACOS**(a) / **ASIN**(a) / **ATAN**(a): calculates the arc-cosine/arc-sine/arc-tangent (in degrees) of *a.*
- **FLOOR**(a,n): returns the largest round number less than *a* with a precision of *n*
- **CEIL**(a,n): returns the smallest round number greater than *a* with a precision of *n*
- **RANDOM**(a,b): generates a random integer between *a* and *b* (included)

Let's code the example of the normal distribution in mathematics. It's interesting because it uses the square function, the square root function and the exponential function:

```
// Normal Law applied to x = 10, StandardDeviation = 6 and MathExpectation = 8
// Let's define the following variables in the variable option:
StandardDeviation = 6
MathExpectation = 8
x = 10
Indicator = EXP( - (1 / 2) * (SQUARE(x - MathExpectation) / StandardDeviation)) / (StandardDeviation * SQRT(2 / 3.1415 ))
RETURN Indicator
```

Common mathematical operators

- **a < b:** a is strictly less than b
- **a <= b or a =< b:** a is less than or equal to b
- **a > b:** a is strictly greater than b
- **a >= b or a => b:** a is greater than or equal to b
- **a = b:** a is equal to b (or b is attributed to a)
- **a <> b:** a is different from b

Charting comparison functions

- **a CROSSES OVER b:** curve a crosses over curve b
- **a CROSSES UNDER b:** curve a crosses under curve b

Summation functions

- **CUMSUM:** Calculates the sum of a price or indicator over all bars loaded on the chart

The syntax of cumsum is:

```
CUMSUM(price or indicator)
```

Ex: `CUMSUM(Close)` calculates the sum of the close of all the bars loaded on the chart.

- **SUMMATION:** Calculates the sum of a price or indicator over the last n bars

The sum is calculated starting from the most recent value (from right to left)

The syntax of **SUMMATION** is:

```
SUMMATION[number of bars](price or indicator)
```

Ex: `SUMMATION[20](Open)` calculates the sum of the open of the last 20 bars.

Statistical functions

The syntax of all these functions is the same as the syntax for the Summation function, that is:

```
LOWEST[number of bars](price or indicator)
```

- **LOWEST:** displays the lowest value of the price or indicator written between brackets, over the number of periods defined
- **HIGHEST:** displays the highest value of the price or indicator written between brackets, over the number of periods defined
- **STD:** displays the standard deviation of a price or indicator, over the number of periods defined
- **STE:** displays the standard error of a price or indicator, over the number of periods defined

Logical operators

As in any programming language, it is necessary to have at our disposal some Logical Operators to create relevant indicators. These are the 4 Logical Operators of ProBuilder:

- **NOT**(a): logical NO
- a **OR** b: logical OR
- a **AND** b: logical AND
- a **XOR** b: exclusive OR (a OR b but not a AND b)

ProBuilder instructions

- **RETURN**: displays the result of your indicator
- **CALL**: calls another ProBuilder indicator to use in your current program
- **AS**: names the result displayed
- **COLOURED**: colors the displayed curve with the color of your choice

RETURN

We have already seen in chapter I how important the **RETURN** instruction was. It has some specific properties we need to know to avoid programming errors.

The main points to keep in mind when using **RETURN** in order to write a program correctly are that **RETURN** is used:

- Once and only once in each ProBuilder program
- Always at the last line of code
- Optionally with other functions such as **AS**, **COLOURED** and **STYLE**
- To display many results; we write **RETURN** followed with what we want to display and separated with a comma (example: `RETURN a,b`)

Comments

// or //** allow you to write comments inside the code. They are mainly useful to remember how a function you coded works. These remarks will be read but of course not processed by the program. Let's illustrate the concept with the following example:

```
// This program returns the moving average over 20 periods applied to the closing price
RETURN Average[20](Close)
```


Don't use special characters (examples: é,ù,ç,ê,-,_,&…) in ProBuilder code. Special characters may be used only within comments.

CustomClose

CustomClose is a variable allowing you to display the **Close**, **Open**, **High**, **Low** constants and many others, which can be customized in the Settings window of the indicator.

Its syntax is the same as the one of the constants adapted to the timeframe of the chart:

```
CustomClose[n]
```

Example:

```
RETURN CustomClose[2]
```

By clicking on the wrench in the upper left corner of the chart, you will see that it is possible to customize the prices used in the calculation.

CALCULATEONLASTBARS

This parameter allows you to increase the speed at which an indicator will be calculated by defining the number of bars that can be used to calculate the indicator (less bars used in the calculation = faster calculation speed).

Example: `DEFPARAM CALCULATEONLASTBARS = 200`

Warning: the use of the **DEFPARAM** instruction must be done at the beginning of the code.

CALL

CALL allows you to use a personal indicator you have previously coded on the platform.

The quickest method is to click “Insert Function” then select the "User Indicators" category and then select the name of the indicator you want to use and click "Add".

For example, imagine you have coded the Histogram MACD and named it HistoMACD.

Select your indicator and click on "Add". You will see in the programming zone:

```
myHistoMACD = CALL "HistoMACD"
```

ProBuilder gave the name "myHistoMACD" to the variable representing your indicator "HistoMACD".

Here is an example when several variables are returned by your **CALL**:

```
myExponentialMovingAverage, mySimpleMovingAverage = CALL "Averages"
```

AS

The keyword **AS** allows you to name the different results displayed. This instruction is used with **RETURN** and its syntax is:

```
RETURN Result1 AS "Curve Name1", Result2 AS "Curve Name2", …
```

This keyword makes it easier to identify the different curves on your chart.

Example:

```
a = ExponentialAverage[200](Close)
b = WeightedAverage[200](Close)
c = Average[200](Close)
RETURN a AS "Exponential Average", b AS "Weighted Average", c AS "Arithmetic Average"
```

COLOURED

COLOURED is used after the **RETURN** command to color the value displayed with the color of your choice, defined with the RGB norm (Red, Green, Blue) or by using predefined colors.

The 140 predefined colors can be found in the following documentation:

W3School: HTML Color Names

Here are the main colors of the RGB standard as well as their predefined HTML name:

COLOR	RGB VALUE (between 0 and 255) (RED, GREEN, BLUE)	HTML Color name
	(0, 0, 0)	Black
	(255, 255, 255)	White
	(255, 0, 0)	Red
	(0, 255, 0)	Green
	(0, 0, 255)	Blue
	(255, 255, 0)	Yellow
	(0, 255, 255)	Cyan
	(255, 0, 255)	Magenta

The syntax for using the **COLOURED** command is as follows:

```
RETURN Indicator COLOURED(RedValue, GreenValue, BlueValue)
```

Or alternatively:

```
RETURN Indicator COLOURED("cyan")
```

Optionally, you can control the opacity of your curve with the alpha parameter (between 0 and 255):

```
RETURN Indicator COLOURED(RedValue, GreenValue, BlueValue, AlphaValue)
```

The **AS** command can be associated with the **COLOURED**(. , . , .) command:

```
RETURN Indicator COLOURED(RedValue, GreenValue, BlueValue) AS "Name of the curve"
```

Let's go back to the previous example and insert **COLOURED** in the "**RETURN**" line.

```
a = ExponentialAverage[200](Close)
b = WeightedAverage[200](Close)
c = Average[200](Close)
RETURN a COLOURED("red") AS "Exponential Moving Average", b COLOURED("green") AS "Weighted Moving Average", c COLOURED("blue") AS "Simple Moving Average"
```

This picture shows you the color customization of the result.


AAPL
1h
1M
Indicators
APPLE INC.
-2.54%
Price
RTH
Preopen: 141.61(p)
(-0.91% -1.30)
APPLE INC.
Open 147.77
High 150.92
Low 147.50
Close 149.96
High-Low (2.32%) 3.42
Var. / prev. day (1.45%) +2.15
Var. / prev. unit (1.45%) +2.15
150
148
146
143.92
142.91
142
ProRealTime.com - Real Time
Colored
Simple Moving Average 146.68375
Exponential Moving Average 146.66063
WeightedMoving Average 146.52284
147
146.47
146.33
Un-colored
Simple Moving Average 146.68375
Exponential Moving Average 146.66063
WeightedMoving Average 146.52284
147
146.47
146.33
29
30
Dec
02
Mon 5 Dec 2022, 3:00 pm
06

Drawing instructions

These commands allow you to draw objects on the charts but also to customize your candles, the bars of your charts as well as the colors of all these elements.

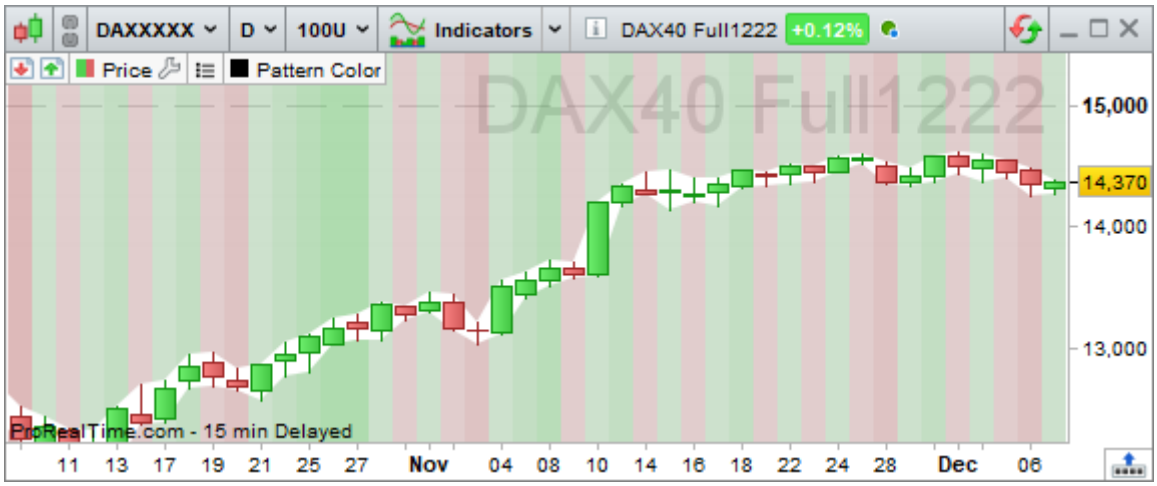
For each instruction below, the color can be defined in a similar way to the color of your curve (**COLOURED** instruction above) with either a predefined color (HTML Color Name) in quotes, or an RGB value (R,G,B) on which you can apply an alpha opacity parameter: (HTML Color Name,alpha) or (R,G,B,alpha)

- **BACKGROUNDCOLOR**(R, G, B, a): Lets you color the background of the chart or specific bars (such as odd/even days). The colored zone starts halfway between the previous bar and the next bar

Example: BACKGROUNDCOLOR (0, 127, 255, 25)

It is possible to use a variable for the colors if you want the background color to change based on your conditions.

Example: BACKGROUNDCOLOR (0, color, 255, 25)



- **COLORBETWEEN**: Allows you to fill the space between two values with a certain color.

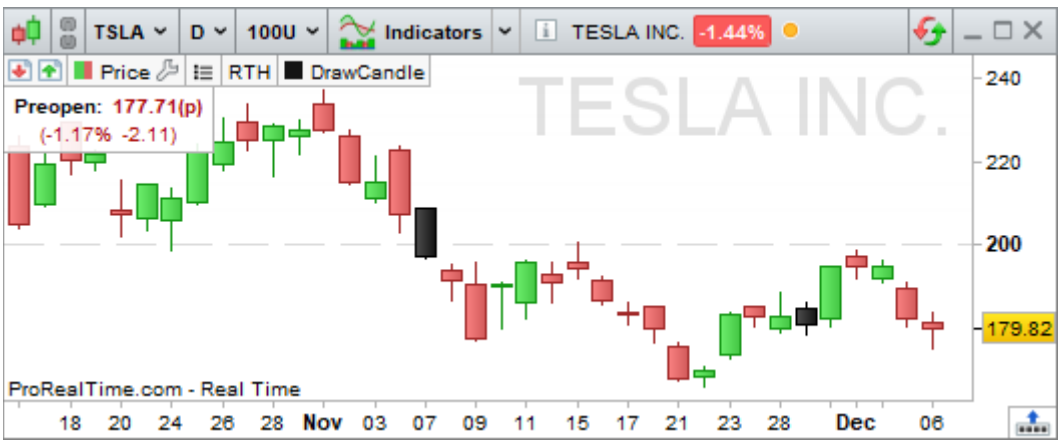
Example: COLORBETWEEN (open, close, "white")

- **DRAWBARCHART**: Draws a custom bar on the chart. Open, high, low and close can be constants or variables.

Example: DRAWBARCHART (open, high, low, close) COLOURED (0, 255, 0)

- **DRAWCANDLE**: Draws a custom candlestick on the chart. Open, high, low and close can be constants or variables.

Example: DRAWCANDLE (open, high, low, close) COLOURED ("black")



For all the drawing instructions below, the x-axis is expressed by default as a bar number (**BARINDEX**) and the y-axis corresponds to the vertical scale of the values in your graph. However, you can change this behavior with the **ANCHOR** command described later.

- **DRAWARROW**: Draws an arrow pointing right. You need to define a point for the arrow (x and y axis). You can also choose a color.

Example: `DRAWARROW (x1, y1) COLOURED (R, G, B, a)`

- **DRAWARROWUP**: Draws an arrow pointing up. You need to define a point for the arrow (x and y axis). You can also choose a color.

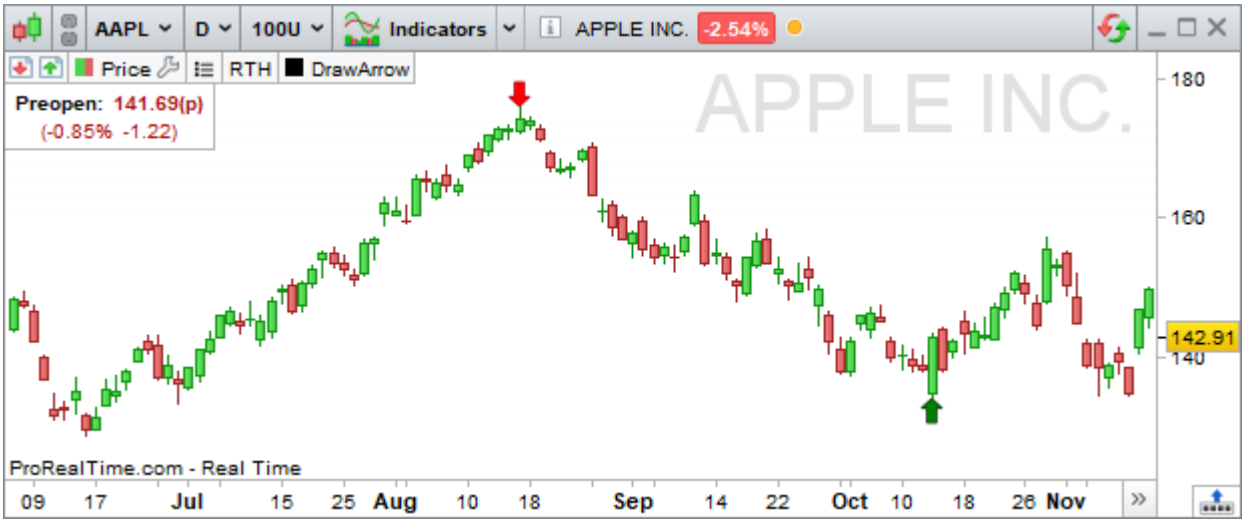
Example: `DRAWARROWUP (x1, y1) COLOURED (R, G, B, a)`

This is useful to add visual buy signals.

- **DRAWARROWDOWN**: Draws an arrow pointing down. You need to define a point for the arrow (x and y axis). You can also choose a color.

Example: `DRAWARROWDOWN (x1, y1) COLOURED (R, G, B, a)`

- This is useful to add visual sell or buy signals.



- **DRAWRECTANGLE**: Draws a rectangle on the chart.

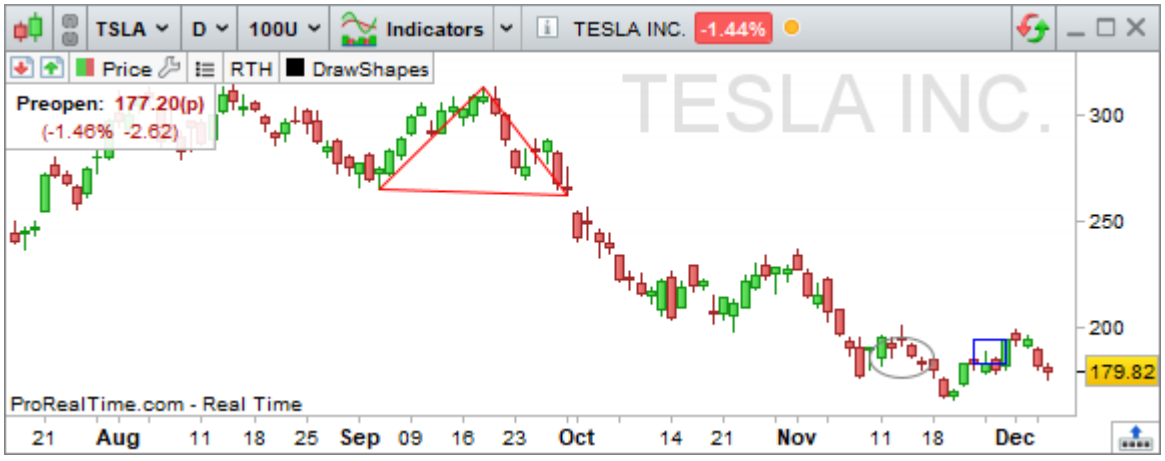
Example: `DRAWRECTANGLE (x1, y1, x2, y2) COLOURED (R, G, B, a)`

- **DRAWTRIANGLE**: Draws a triangle on the chart.

Example: `DRAWTRIANGLE (x1, y1, x2, y2, x3, y3) COLOURED (R, G, B, a)`

- **DRAWELLIPSE**: Draws an ellipse on the chart.

Example: `DRAWELLIPSE (x1, y1, x2, y2) COLOURED (R, G, B, a)`



- **DRAWPOINT**: Draws a point on the chart.

Example: `DRAWPOINT (x1, y1, pointSize) COLOURED (R, G, B, a)`

- **DRAWLINE**: Draws a line on the chart.

Example: `DRAWLINE (x1, y1, x2, y2) COLOURED (R, G, B, a)`

- **DRAWHLINE**: Draws a horizontal line on the chart.

Example: `DRAWHLINE (y1) COLOURED (R, G, B, a)`

- **DRAWVLINE**: Draws a vertical line on the chart.

Example: `DRAWVLINE (x1) COLOURED (R, G, B, a)`

- **DRAWSEGMENT**: Draws a segment on the chart.

Example: `DRAWSEGMENT (x1, y1, x2, y2) COLOURED (R, G, B, a)`

Example: `DRAWSEGMENT (barindex, close, barindex[5], close[5])`

- **DRAWRAY**: Draws a ray on the graph

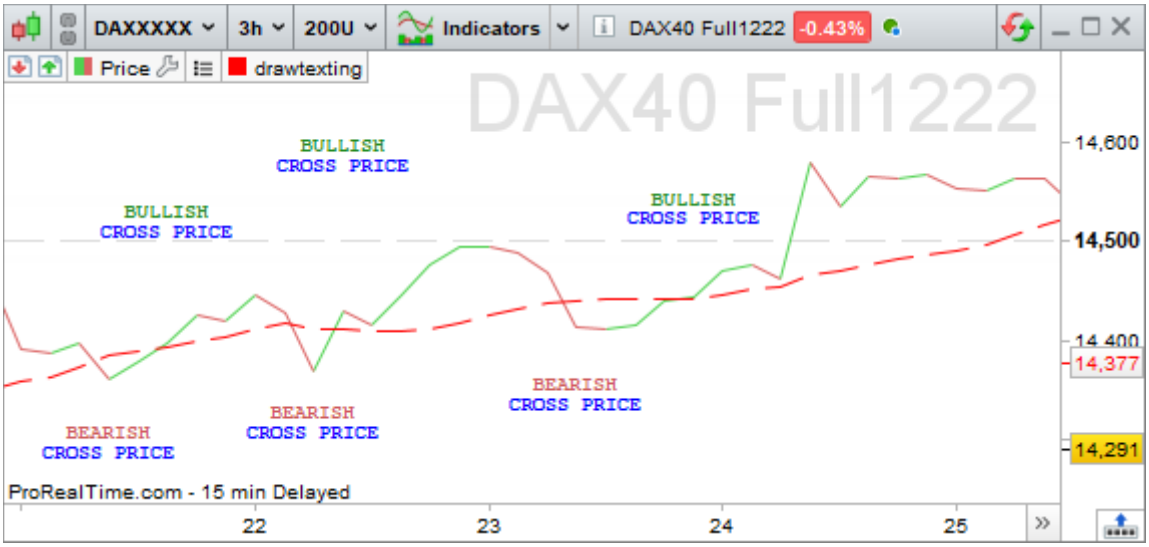
Example: `DRAWRAY ( x1, y1, x2, y2)`

- **DRAWTEXT**: Adds a text field to the chart with text of your choice at a specified location.The syntax #Variable# allows the display of the value of Variable inside a text.

Example: `DRAWTEXT(value, x1, y1, font, fontStyle, fontSize) COLOURED (R,G,B,a)`

Example: `DRAWTEXT("your text", x1, y1, SERIF, BOLD, 10) COLOURED (R, G, B, a)`

Example: `DRAWTEXT("My Variable is #Variable#", x1, y1) COLOURED ("green")`



- Here are the different possible values for the **font** and **font style** parameters, the **font size** is between `1` and `30`:

Font	Font style
DIALOG	STANDARD
MONOSPACED	BOLD
SANSERIF	BOLDITALIC
SERIF	ITALIC

- **DRAWONLASTBARONLY**: Parameter that lets you draw objects on the last bar only. This parameter should always be used with "**CALCULATEONLASTBARS**" to optimize calculations.

Example: `DEFPARAM DRAWONLASTBARONLY = true`

Additional parameters

For some of these design commands, various additional instructions can be applied in no particular order:

BORDERCOLOR

This instruction allows you to define the color of the border of a drawn object (excluding lines and arrows).

Example 1: `DRAWRECTANGLE(barindex, close, barindex[5], close[5]) BORDERCOLOR(r,g,b,a)`

Example 2: `DRAWRECTANGLE(barindex, close, barindex[5], close[5]) BORDERCOLOR("red")`

ANCHOR

This instruction allows you to define the anchor point of the object when you want to draw it from a starting point other than the candlesticks.

```
DRAWTEXT("text", n, p) ANCHOR(referencePoint, horizontalShift, verticalShift)
```

It can take several values as parameters:

- **Parameter 1**: the position of the anchor

Value	Description
TOPLEFT	Fixed at the top left of the chart
TOP	Fixed at the top of the chart (middle)
TOPRIGHT	Fixed at the top right of the chart
RIGHT	Fixed to the right of the graph (middle)
BOTTOMRIGHT	Fixed at the bottom right of the chart
BOTTOM	Fixed at the bottom of the graph (middle)
BOTTOMLEFT	Fixed at the bottom left of the chart
LEFT	Fixed to the left of the graph (middle)
MIDDLE	Fixed in the center of the graph

- **Parameter 2**: the type of value to set the positioning on the horizontal axis
  - `INDEX:` The values entered in the drawing of the object for the horizontal axis will refer to the barindex of the candlesticks
  - `XSHIFT:` The values entered in the drawing of the object for the horizontal axis will refer to an offset value in pixels (positive or negative with respect to an orthonormal reference frame)
- **Parameter 3**: the type of value to set the positioning on the vertical axis
  - `VALUE:` The values entered in the drawing of the object for the vertical axis will refer to a price
  - `YSHIFT:` The values entered in the drawing of the object for the vertical axis will refer to an offset value in pixels (positive or negative with respect to an orthonormal reference frame)

Examples:

```
DRAWTEXT("My Variable value: #Var#", -20, -50) ANCHOR(TOPRIGHT, XSHIFT, YSHIFT)
```

Displays the `Var` variable value at the top right of the graph with an offset of `-20` on the horizontal axis and `-50` on the vertical axis.

```
DRAWTEXT("Top", Barindex-10, -20) ANCHOR(TOP, INDEX, YSHIFT)
```

Draws the text `"Top"` at the top of the chart with an offset of `-20` on the vertical axis and positioned in the continuity of the 10th **`BarIndex`** before the last bar.

STYLE

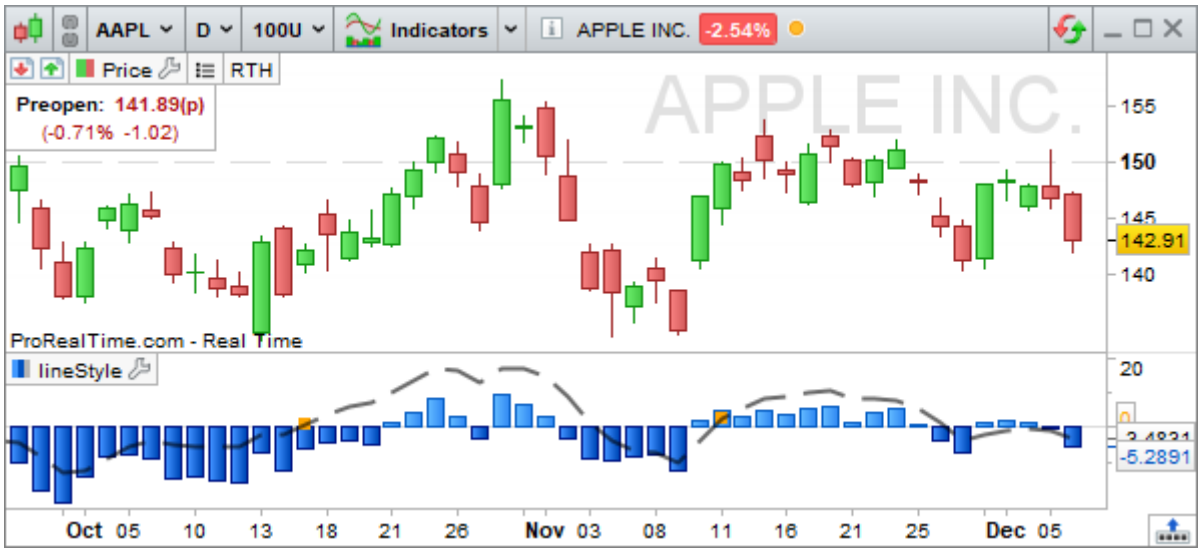
This instruction allows you to define a style for objects (except arrows) or for returned values.

```
DRAWRECTANGLE(x1, y1, x2, y2) STYLE(style, lineWidth)
```

There are different styles:

- `DOTTEDLINE`: this style transforms the line into a dotted line. There are 5 different configurations that represent 5 different dotted line lengths: `DOTTEDLINE`, `DOTTEDLINE1`, `DOTTEDLINE2`, `DOTTEDLINE3`, `DOTTEDLINE4`
- `LINE`: this style restores the default line style (full line)
- `HISTOGRAM:` this style, only applicable in the `RETURN` instruction of an indicator, displays the returned values as a histogram.
- `POINT:` this style, only applicable in the `RETURN` instruction of an indicator, displays the returned values as a point.

`LineWidth`, which defines the thickness of the line, will take a value between `1` (the thinnest) and `5` (the thickest).



Note: for the drawing functions it is possible to specify a date rather than a candlestick index thanks to the **DateToBarIndex** function which allows you to transform a date to the nearest associated bar index.
The instruction is written in the following form:

```
DateToBarIndex(date)
```

Expected date formats:

- *YYYY* / Example: 2022
- *YYYYMM* / Example: 202208
- *YYYYMMDD* / Example: 20220815
- *YYYYMMDDHH* / Example: 2022081517
- *YYYYMMDDHHMM* / Example: 202208151730
- *YYYYMMDDHHMMSS* / Example: 20220815173020

Multi-period instructions

ProBuilder allows you to work on different time periods in your Backtests, Indicators and Screeners, giving you access to more complete data when designing your codes. The instruction is structured as follows:

```
TIMEFRAME(X TimeUnit , Mode)
```

With the following parameters:

- `TimeUnit`: The type of period chosen (see **List of available time frames**)
- `X`: The value associated with the selected period
- `Mode`: The selected calculation mode (optional)

Example: `TIMEFRAME(1 Hour)`

You can use multi-timeframe instructions only to call time units greater than your base time unit (time unit of the chart).

The secondary time units called must also be a **multiple** of the base time unit .

Thus, on a 10-minutes chart:

We can call the following time frames: 20 minutes, 1 hour, 1 day.

We can't call the 5 minutes or 17 minutes time frames.

To enter a higher time frame, you need to use the instruction:

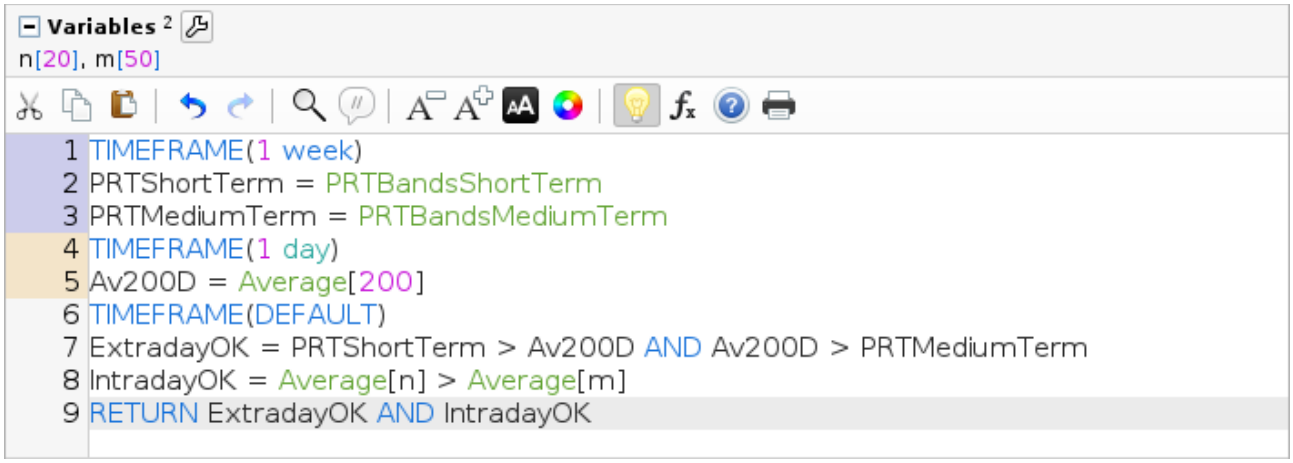
`TIMEFRAME(X TimeUnit)`

To return to the base time frame of the chart, use the following command:

`TIMEFRAME(DEFAULT)`

You can also indicate the time frame of the base chart.

The platform editor colors the background of the code blocks in higher `TimeFrame` to help you visualize the pieces of code calculated in each different time frame.



It is also possible to use two calculation modes in a larger time unit in order to have more flexibility in your calculations:

```
TIMEFRAME(X TimeUnit , DEFAULT)
TIMEFRAME(X TimeUnit , UPDATEONCLOSE)
```

DEFAULT: this is the default mode of the **TimeFrame** (mode used when the second parameter is not specified), the calculations in the higher time frames are performed at each new price received in the base time unit of the chart.

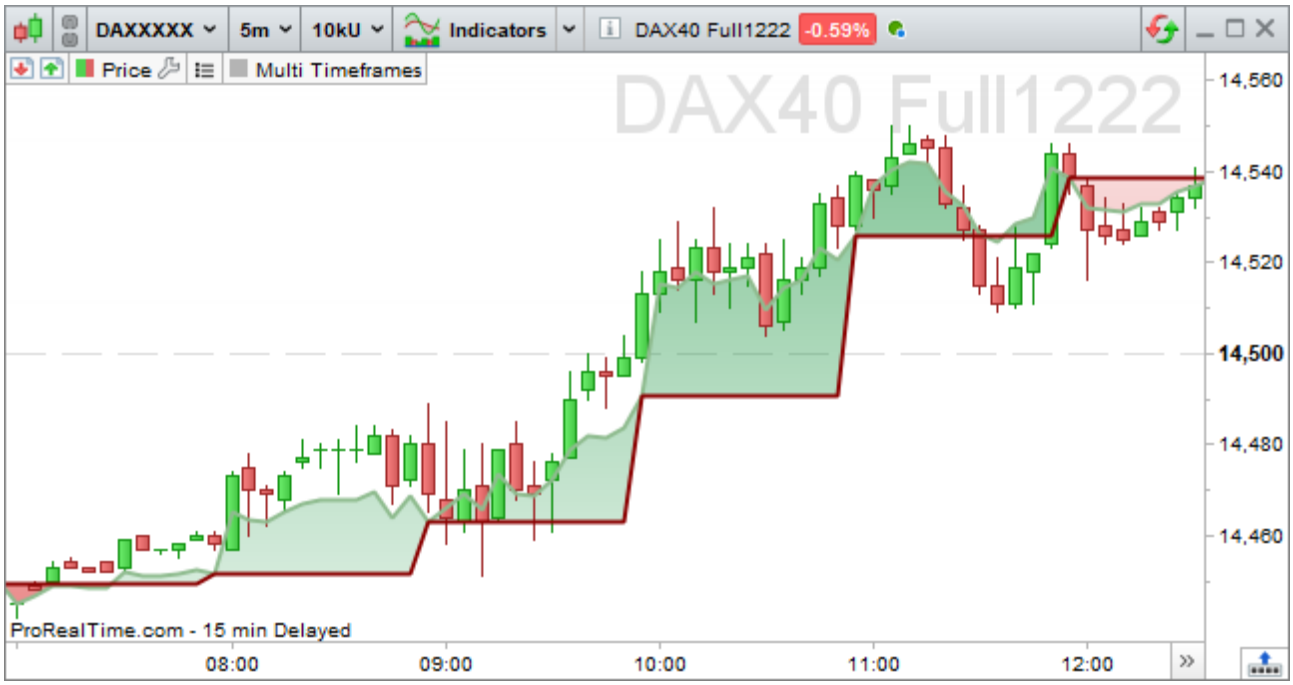
UPDATEONCLOSE: the calculations contained in a time frame in this mode are performed at the closing of the candlestick of the higher time frame.

Here is an example of code showing the difference between the two calculation modes:

```
// Average price computation between opening and closing in the two available modes.
TIMEFRAME(1 Hour)
MidPriceDefault=(Open+Close)/2

TIMEFRAME(1 Hour, UPDATEONCLOSE)
MidPriceUpdateOnClose=(Open+Close)/2

Return MidPriceDefault as "Average Price Default mode" COLOURED ("DarkSeaGreen"),
MidPriceUpdateOnClose as "Average Price UpdateOnClose mode" COLOURED ("DarkRed")
```



Here we notice that my `MidPriceDefault` (in green) is updated after every 5 minute candlestick, while `MidPriceUpdateOnClose` (in red) is updated after every 1 hour candlestick.

Note on the use of the `TIMEFRAME` instruction:

- A variable calculated in one time frame cannot be overwritten by a calculation in another time frame, on the other hand the variables can be used in all time frames contained in the same code.
- There is a limit of 10 `TIMEFRAME` intraday instructions (smaller than daily) for automatic trading and backtesting.
- For the ProScreener module, only the `DEFAULT` mode is available, so it is not necessary to specify the mode. Moreover, in order to guarantee the performance of calculations on many real time values, only a predefined list of available time frames is authorized for this module.
  For more information, please read the **Programming Guide - Market Scans (ProScreener)**.

List of available time frames

Periods	Examples
`Tick / Ticks`	`TIMEFRAME(1 Tick)`
`sec / Second / Seconds`	`TIMEFRAME(10 Seconds)`
`mn / Minute / Minutes`	`TIMEFRAME(5 Minutes)`
`Hour / Hours`	`TIMEFRAME(1 Hour)`
`Day / Days`	`TIMEFRAME(5 Days)`
`Week / Weeks`	`TIMEFRAME(1 Weeks)`
`Month / Months`	`TIMEFRAME(2 Month)`
`Year / Years`	`TIMEFRAME(1 Year)`

Arrays (Data tables)

In order to be able to store several values on the same candlestick or to store values only when necessary, we suggest you to use Arrays (data tables) instead of variables.

A code can contain as many arrays as necessary, which can contain up to one million values each.

An array is always prefixed with the $ symbol.

Syntax of a variable	Syntax of an array
A	$A

An array starts from index 0 to index 999 999

Index	0	1	2	3	4	5	6	...	999 999
Value									

To insert a value in an array, simply use

```
$Array[Index] = value
```

For example if we want to insert the value of the calculation of the moving average of period 20 at index 0 of array A, we will write:

```
$A[0] = Average[20](Close)
```

To read the value of an index of the array, we will use, on the same principle:

```
$Array[Index]
```

For example if we want to create a condition that checks that the close is greater than the value of the first index of the array A:

```
Condition = Close > $A[0]
```

When inserting a value at an index **n** of an array, ProBuilder will initialize the values to zero for all the undefined indices from 0 to **n-1 in** order to facilitate the use of the data contained in this array.

Specific functions

Several functions specific to arrays are available to facilitate their manipulation and use:

- **ArrayMax**($Array): returns the highest value of the array that has been defined. The zeros filled automatically by ProBuilder are not taken into account.
- **ArrayMin**($Array): returns the smallest value of the array that has been defined. The zeros filled automatically by ProBuilder are not taken into account.
- **ArraySort**($Array, MODE): Sorts the array in ascending order (mode=**ASCEND**) or in descending order (mode=**DESCEND**). The zeros filled automatically by ProBuilder will then be removed.
- **IsSet**($Array[index]): returns 1 if the index of the array has been defined, 0 if it has not been defined. The zeros filled automatically by ProBuilder are not considered as having been defined so the function will return 0 on these indices.
- **LastSet**($Array): returns the highest defined index of the array, if no index has been defined in the array, the function will return -1.
- **UnSet**($Array): Resets the array to 0 by completely deleting its content.

Note: unlike variables and other calculations performed in our language, arrays are not historized. It is therefore not possible to retrieve the value of a cell from an array calculated on a previous candlestick.

If you want to see examples of how to use these functions, we recommend **this link** from our partner ProRealCode which details the use of arrays through different examples.

PRINT

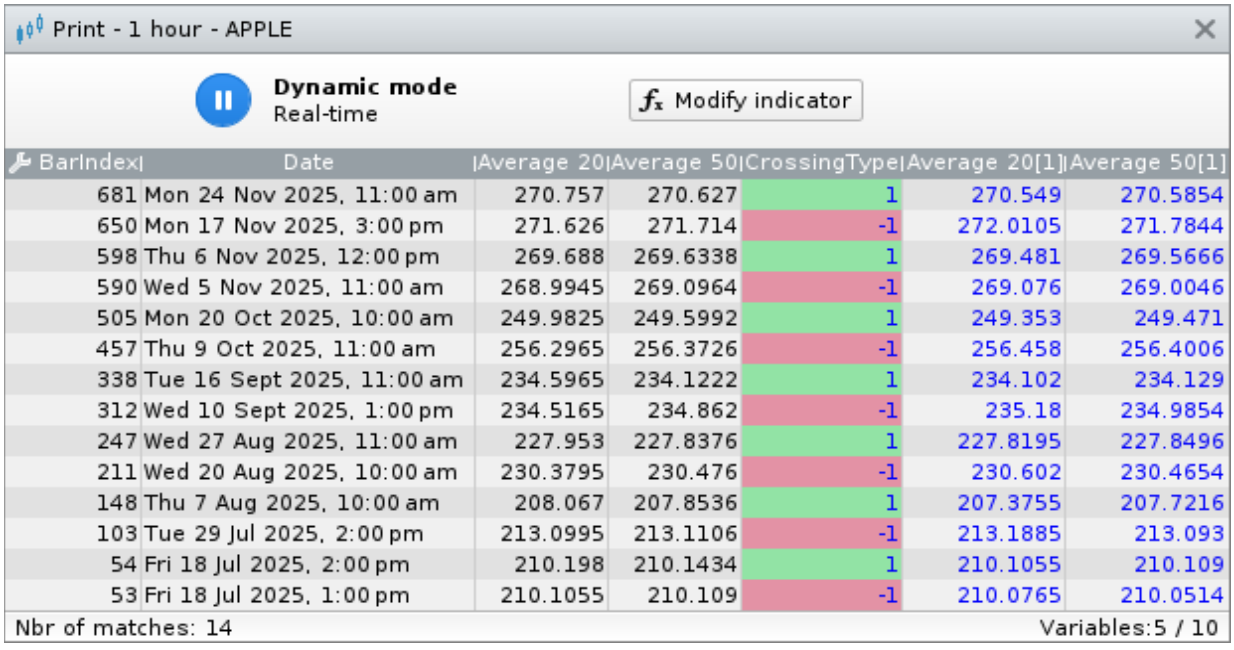
PRINT is used to display variable values from the ProBuilder program in a table format along with the **Date** and the **BarIndex**.

It is very useful when coding and debugging. The syntax is as follows:

```
PRINT Var AS "Column header" FILLCOLOR (r,g,b,a) COLOURED (r,g,b,a)
```

Here is an example code and a possible output:

```
Av20 = Average[20]
Av50 = Average[50]
CrossingOver = Av20 CROSSES OVER Av50
CrossingUnder = Av20 CROSSES UNDER Av50
IF CrossingOver OR CrossingUnder THEN
    IF CrossingOver THEN
        r = 35
        g = 200
    ELSIF CrossingUnder THEN
        r = 200
        g = 35
    ENDIF
    PRINT Av20 AS "Average 20"
    PRINT Av50 AS "Average 50"
    PRINT Av20[1] AS "Average 20[1]" COLOURED(0,0,250)
    PRINT Av50[1] AS "Average 50[1]" COLOURED(0,0,250)
    PRINT CrossingOver - CrossingUnder AS "CrossingType" FILLCOLOR(r,g,75,126)
COLOURED(0,0,250)
ENDIF
RETURN CrossingOver - CrossingUnder
```



BarIndex	Date	Average 20	Average 50	CrossingType	Average 20[1]	Average 50[1]
681	Mon 24 Nov 2025, 11:00 am	270.757	270.627	1	270.549	270.5854
650	Mon 17 Nov 2025, 3:00 pm	271.626	271.714	-1	272.0105	271.7844
598	Thu 6 Nov 2025, 12:00 pm	269.688	269.6338	1	269.481	269.5666
590	Wed 5 Nov 2025, 11:00 am	268.9945	269.0964	-1	269.076	269.0046
505	Mon 20 Oct 2025, 10:00 am	249.9825	249.5992	1	249.353	249.471
457	Thu 9 Oct 2025, 11:00 am	256.2965	256.3726	-1	256.458	256.4006
338	Tue 16 Sept 2025, 11:00 am	234.5965	234.1222	1	234.102	234.129
312	Wed 10 Sept 2025, 1:00 pm	234.5165	234.862	-1	235.18	234.9854
247	Wed 27 Aug 2025, 11:00 am	227.953	227.8376	1	227.8195	227.8496
211	Wed 20 Aug 2025, 10:00 am	230.3795	230.476	-1	230.602	230.4654
148	Thu 7 Aug 2025, 10:00 am	208.067	207.8536	1	207.3755	207.7216
103	Tue 29 Jul 2025, 2:00 pm	213.0995	213.1106	-1	213.1885	213.093
54	Fri 18 Jul 2025, 2:00 pm	210.198	210.1434	1	210.1055	210.109
53	Fri 18 Jul 2025, 1:00 pm	210.1055	210.109	-1	210.0765	210.0514

Nbr of matches: 14 Variables:5 / 10

Option:

- **FILLCOLOR**: Allows you to set the background color of the cell displayed by **Print**.
- **COLOURED**: Allows you to set the text color of the cell displayed by **Print**.
- **AS**: Allows you to choose the column header dedicated to the variable (it is impossible to have two columns with the same name).

Colors can be defined using RGBA parameters or W3C-standard color names. Please refer to the coloring section (**COLOURED**) of the ProBuilder keywords, which follow the same rules.

Note: As shown in the example, a row is added to the table only when the instruction is encountered. **PRINT** is therefore very handy to use inside **IF** loops. When ProBuilder codes become complex, certain variables sometimes need to be non-zero or positive. To ensure this, it is convenient to use the **PRINT** instruction:

```
// ...
IF MyPositiveVariable < 0 THEN
    PRINT MyPositiveVariable AS "Error: SQRT of a negative variable"
ENDIF
OtherVariable = SQRT(MyPositiveVariable)
// ...
```

In this case, if everything goes well, no table will be displayed. However, you can be certain that MyPositiveVariable is indeed positive throughout the entire history and will not cause issues downstream in your code.

It is possible to display up to 10 different calculated values during the execution of the same code (any values beyond the first ten will be ignored, **PRINT** statements that are never encountered throughout the execution are not counted).

The displayed values window can hold a history of the 200 most recent calculations, which are updated in real time as your code performs computations.

Chapter III: Practical aspects

Create a binary or ternary indicator: why and how ?

A binary or ternary indicator is by definition an indicator that can only return two or three possible results (usually 0, 1 or -1). Its main use in a stock market context is to make the verification of the conditions that constitute the indicator immediately identifiable.

Uses of a binary or ternary indicator:

- Enable the detection of the main Japanese candlestick patterns
- Facilitate the reading of a chart graph when trying to verify several conditions at once
- So that you can place standard one-condition alerts on an indicator that contains multiple conditions.
- Detecting complex conditions also on historical data
- Facilitate the creation or execution of a backtest

Binary or ternary indicators are constructed using the **IF** function. We advise you to reread the relative section before continuing reading.

Let's picture the creation of these indicators to detect price patterns:

Binary Indicator: hammer detection

```
Hammer = Close > Open AND High = Close AND (Open-Low) >= 3*(Close-Open)
IF Hammer THEN
   Result = 1
ELSE
   Result = 0
ENDIF
RETURN Result AS "Hammer"
```

This simplified code will also give the same results:

```
Hammer = Close > Open AND High = Close AND ( Open - Low ) >= 3 * ( Close - Open )
RETURN Hammer AS "Hammer"
```

Ternary Indicator: Golden Cross and Death Cross detection

```
EA10 = ExponentialAverage[10](Close)
EA20 = ExponentialAverage[20](Close)

GoldenCross = EA10 CROSSES OVER EA20
DeathCross = EA10 CROSSES UNDER EA20

// Boolean variables are either 0 or 1 so, when mutually exclusive, substracting two Boolean variables gives a 3-state variable (0-1=-1,0-0=0,or 1-0=1)
Cross = goldenCross - deathCross

RETURN Cross STYLE (HISTOGRAM) COLOURED(DeathCross*210,GoldenCross*210,100)
```

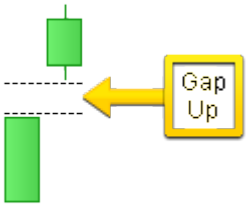
Note: we have displayed the exponential moving average over 10 and 20 periods both applied to the close in order to highlight the results of the indicator.

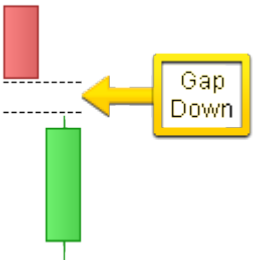
You can find other candlestick pattern indicators in the "Exercises" chapter later in this manual.

Chapter IV: Exercises

Candlestick patterns

- **GAP UP or DOWN**





The color of the candlesticks is not important.

We define a customizable variable, Amplitude = 0.001

A gap is defined by these two conditions:

- (the current low is strictly greater than the high of the previous bar) or (the current high is strictly lesser than the low of the previous bar)
- the absolute value of ((the current low – the high of the previous bar)/the high of the previous bar) is strictly greater than amplitude or ((the current high – the low of the previous bar)/the low of the previous bar) is strictly greater than amplitude

```
ONCE Amplitude = 0.001
// useDailyCandle allows you to use DHigh and DLow instead of High and Low
// Gaps are usually computed on Daily candles but it can be interesting to compute them on other timeframes
ONCE useDailyCandle = 1

IF useDailyCandle THEN
    GapUpc1 = DLow(0) > DHigh(1)
    GapUpc2 = ABS((DLow(0) - DHigh(1)) / DHigh(1)) > Amplitude
    GapDownc1 = DHigh(0) < DLow(1)
    GapDownc2 = ABS((DHigh(0) - DLow(1)) / DLow(1)) > Amplitude
ELSE
    GapUpc1 = Low > High[1]
    GapUpc2 = ABS((Low - High[1]) / High[1]) > Amplitude
    GapDownc1 = High < Low[1]
    GapDownc2 = ABS((High - Low[1]) / Low[1]) > Amplitude
ENDIF
GapUp = GapUpc1 AND GapUpc2
GapDown = GapDownc1 AND GapDownc2
// Drawing a horizontal line to highlight the level 0 (no gap)
DRAWHLINE(0) STYLE (DOTTEDLINE2) COLOURED("blue")
// Boolean variables are either 0 or 1 so, when mutually exclusive, substracting 2 boolean variables gives a 3-state variable (0-1=-1,0-0=0,or 1-0=1)
RETURN GapUp - GapDown AS "Gap" STYLE (HISTOGRAM) COLOURED(GapDown*210,GapUp*210,100)
```

- **Doji (flexible version)**

In this code, we define a doji to be a candlestick with a range (High – Close) greater than 5 times the absolute value of (Open – Close).

```
Doji = Range > ABS(Open - Close) * 5
RETURN Doji AS "Doji"
```

- **Doji (strict version)**

We define the doji with a Close equal to its Open.

```
Doji = (Open = Close)
RETURN Doji AS "Doji"
```

Indicators

BODY MOMENTUM

Body Momentum is mathematically defined by: BodyMomentum = 100 * BodyUp / (BodyUp + BodyDown)

BodyUp is a counter of bars for which close is greater than open during a certain number of periods (in this example : 14).

BodyDown is a counter of bars for which open is greater than close during a certain number of periods (in this example : 14).

```
ONCE Periods = 14

// Since ( Close > Open ) and ( Close < Open ) are boolean variables their value is either 0 (false) or 1 (true).
// Thus applying summation on these variables allows us to count the number of cases where the boolean variable is true within the time window (here 14)
BodyUp = summation[Periods]( Close > Open )
BodyDown = summation[Periods]( Close < Open )

BodyM = (BodyUp / (BodyUp + BodyDown)) * 100

RETURN BodyM AS "Body Momentum"
```

ELLIOTT WAVE OSCILLATOR

The Elliott wave oscillator shows the difference between two moving averages.

This oscillator permits to distinguish between wave 3 and wave 5 using Elliott wave theory.

The short MA shows short-term price action whereas the long MA shows the longer term trend.

When the prices form wave 3, the prices climb strongly which shows a high value of the Elliott Wave Oscillator.

In wave 5, the prices climb more slowly, and the oscillator will show a lower value.

```
RETURN Average[5](MedianPrice) - Average[35](MedianPrice) AS "Elliott Wave Oscillator"
```

- **Williams %R**

This is an indicator very similar to the Stochastic oscillator. To draw it, we define 2 curves:
1) The curve of the highest high over 14 periods
2) The curve of the lowest low over 14 periods
The %R curve is defined by this formula: (Close – Lowest Low) / (Highest High – Lowest Low) * 100

```
HighestH = highest[14](High)
LowestL = lowest[14](Low)
MyWilliams = (Close - LowestL) / (HighestH - LowestL) * 100
RETURN MyWilliams AS "Williams %R"
```

- **Bollinger Bands**

The middle band is a simple 20-period moving average applied to close.
The upper band is the middle band plus 2 times the standard deviation over 20 periods applied to close.
The lower band is the middle band minus 2 times the standard deviation over 20 periods applied to close.

```
ONCE Per = 20
ONCE nbSTD = 2

Bmid = Average[Per](Close)
StdDeviation = STD[Per](Close)

Bsup = Bmid + nbSTD * StdDeviation
Binf = Bmid - nbSTD * StdDeviation

RETURN Bmid AS "Average", Bsup AS "Bollinger Up", Binf AS "Bollinger Down"
```

You can visit our ProRealTime community on the **ProRealCode forum** to find **online documentation** and many more examples.

Glossary

A

CODE	SYNTAX	FUNCTION
ABS	ABS(a)	Mathematical function "Absolute Value" of a.
AccumDistr	AccumDistr(price)	Classical Accumulation/Distribution indicator.
ACOS	ACOS(a)	Mathematical function "Arc cosine" (returns an angle in degrees).
AdaptiveAverage	AdaptiveAverage[x,y,z](price)	Adaptive Average Indicator.
ADX	ADX[N]	Indicator Average Directional Index or "ADX" of n periods.
ADXR	ADXR[N]	Indicator Average Directional Index Rate or "ADXR" of n periods.
AND	a AND b	Logical AND Operator.
ArrayMax	ArrayMax($MyArray)	Returns the max value of the array.
ArrayMin	ArrayMin($MyArray)	Returns the min value of the array.
ArraySort	ArraySort($MyArray, ASCEND)	Sort the table in ascending (ASCEND) or descending (DESCEND) order.
AroonDown	AroonDown[P]	Aroon Down indicator.
AroonUp	AroonUp[P]	Aroon Up indicator.
ATAN	ATAN(a)	Mathematical function "Arc tangent" (returns an angle in degrees).
ANCHOR	ANCHOR(direction, index, yshift)	Anchor function for drawings.
AS	RETURN x AS "ResultName"	Names a line or indicator displayed on chart. Used with "RETURN".
ASIN	ASIN(a)	Mathematical function "Arc sine" (returns an angle in degrees).
Average	Average[N](price)	Simple Moving Average of n periods.
AverageTrueRange	AverageTrueRange[N](price)	"Average True Range" - True Range smoothed with the Wilder method.

B

CODE	SYNTAX	FUNCTION
BACKGROUNDCOLOR	BACKGROUNDCOLOR(R,G,B,a)	Sets the background color of the chart or a specific bar.
BarIndex	BarIndex	Number of bars since the beginning of data loaded (in a chart in the case of a ProBuilder indicator or for a trading system in the case of ProBacktest or ProOrder).
BarsSince	BarsSince(condition,occurence)	Returns the number of candles since the nth occurrence of the specified condition (n=0 means last occurrence and is the default, n=1 means second last occurrence).
Bold	DRAWTEXT(« text »,barindex,close,Serif,Bold, 10)	Bold style to be applied to the text.
BoldItalic	DRAWTEXT(« text »,barindex,close,Serif,BoldItalic, 10)	Bold italic style to be applied to the text.
BollingerBandWidth	BollingerBandWidth[N](price)	Bollinger Bandwidth indicator.
BollingerDown	BollingerDown[N](price)	Lower Bollinger band.
BollingerUp	BollingerUp[N](price)	Upper Bollinger band.
BOTTOM	ANCHOR(BOTTOM,INDEX,YSHIFT)	Anchor at the bottom of the chart.
BOTTOMLEFT	ANCHOR(BOTTOMLEFT,INDEX,YSHIFT)	Anchor at the bottom left of the chart.
BOTTOMRIGHT	ANCHOR(BOTTOMRIGHT,INDEX,YSHIFT)	Anchor at the bottom right of the chart.
BORDERCOLOR	BORDERCOLOR("red")	Adds a colored border to the associated object.
BREAK	(FOR/DO/BREAK/NEXT) or (WHILE/DO/BREAK/WEND)	Instruction to exit a FOR or a WHILE loop.

C

CODE	SYNTAX	FUNCTION
CALCULATEONLASTBARS	DEFPARAM CalculateOnLastBars = 200	Lets you decrease the calculation time by defining the number of bars to display the results on, starting with the most recent bar.
CALL	myResult = CALL myFunction	Calls a user indicator to be used in the program you are coding.
CCI	CCI[N](price)	Commodity Channel Index indicator.
CEIL	CEIL(N, M)	Returns the smallest number greater than N applied to the decimal m.
ChaikinOsc	ChaikinOsc[Ch1, Ch2](price)	Chaikin oscillator.
Chandle	Chandle[N](price)	Chande Momentum Oscillator.
ChandeKrollStopUp	ChandeKrollStopUp[Pp, Qq, X]	Chande and Kroll Protection Stop on long positions.
ChandeKrollStopDown	ChandeKrollStopDown[Pp, Qq, X]	Chande and Kroll Protection Stop on short positions.
Close	Close[N]	Closing price of the current bar or of the nth last bar.
COLOURED	RETURN x COLOURED(R,G,B)	Colors a curve with the color you defined using the RGB convention.
COLORBETWEEN	COLORBETWEEN(a, b, color)	Color the space between two values.
COS	COS(a)	Cosine function ('a' argument in degrees).
CROSSES OVER	a CROSSES OVER b	Boolean Operator checking whether a curve has crossed over another one.
CROSSES UNDER	a CROSSES UNDER b	Boolean Operator checking whether a curve has crossed under another one.
Cumsum	Cumsum(price)	Sums a certain price on the whole data loaded.
CurrentDayOfWeek	CurrentDayOfWeek	Represents the current day of the week.
CurrentHour	CurrentHour	Represents the current hour.
CurrentMinute	CurrentMinute	Represents the current minute.
CurrentMonth	CurrentMonth	Represents the current month.
CurrentSecond	CurrentSecond	Represents the current second.
CurrentTime	CurrentTime	Represents the current time (HHMMSS).
CurrentYear	CurrentYear	Represents the current year.
CustomClose	CustomClose[N]	Constant which is customizable in the settings window of the chart (default: Close).
Cycle	Cycle(price)	Cycle Indicator.

D

CODE	SYNTAX	FUNCTION
Date	Date[N]	Reports the date of each closing bar loaded on the chart.
DATETOBARINDEX	DATETOBARINDEX(date)	Allows you to use a date for the drawing functions.
Day	Day[N]	Day number at the end of the N-th candle.
Days	Days[N]	Counter of days since 1900.
Days	TIMEFRAME(X Days)	Set the period to "X Days" for further calculations of the code.
DayOfWeek	DayOfWeek[N]	Day of the week of each closing bar.
DClose	DClose(N)	Close of the nth day before the current one.
Decimals	Decimals	Returns the number of decimals of the ticker.
DEMA	DEMA[N](price)	Double Exponential Moving Average.
DHigh	DHigh(N)	High of the n-th day before the current bar.
Dialog	DRAWTEXT(« text »,barindex,close,Dialog,Bold, 10)	Dialog font applied to text.
DI	DI[N](price)	Represents the Demand Index indicator.
DIminus	DIminus[N](price)	Represents the DI- indicator.
DIplus	DIplus[N](price)	Represents the DI+ indicator.

CODE	SYNTAX	FUNCTION
DivergenceCCI	DivergenceCCI[Div1,Div2,Div3,Div4]	Indicator for detecting discrepancies between price and the CCI.
DivergenceMACD	DivergenceMACD[Div1,Div2,Div3,Div4](close)	Indicator for detecting divergences between the price and the MACD.
DivergenceRSI	DivergenceRSI[Div1,Div2,Div3,Div4](close)	Indicator for detecting divergences between the price and the RSI.
DLow	DLow(N)	Low of the nth day before the current one.
DO	See FOR and WHILE	Optional instruction in FOR loop and WHILE loop to define the loop action.
DonchianChannelCenter	DonchianChannelCenter[N]	Middle channel of the Donchian indicator for N periods.
DonchianChannelDown	DonchianChannelDown[N]	Lower channel of the Donchian indicator for N periods.
DonchianChannelUP	DonchianChannelUp[N]	Upper channel of the Donchian indicator for N periods.
DOpen	DOpen(N)	Open of the nth day before the current one.
DOTTEDLINE	STYLE(DOTTEDLINE1/2/3/4, width)	Style applicable to the features of an object.
DOWNTO	See FOR	Instruction used in FOR loop to process the loop with a descending order.
DPO	DPO[N](price)	Detrended Price Oscillator.
DRAWARROW	DRAWARROW(x1,y1)	Draw an arrow pointing right at the selected point.
DRAWARROWDOWN	DRAWARROWDOWN(x1,y1)	Draw a down at the selected point.
DRAWARROWUP	DRAWARROWUP(x1,y1)	Draw an up arrow at the selected point.
DRAWBARCHART	DRAWBARCHART(open,high,low,close)	Draws a custom bar on the chart. Open, High, Low, and Close can be constants or variables.
DRAWCANDLE	DRAWCANDLE(open,high,low,close)	Draws a custom candlestick on the current barindex. Open, high, low, and close can be constants or variables.
DRAWELLIPSE	DRAWELLIPSE(x1,y1,x2,y2)	Draws an ellipse on the chart.
DRAWHLINE	DRAWHLINE(y1)	Draws a horizontal line on the chart at the selected point.
DRAWLINE	DRAWLINE(x1,y1,x2,y2)	Draws a line on the chart between the two selected points.
DRAWONLASTBARONLY	DEFPARAM DrawOnLastBarOnly = true	Parameter that lets you draw drawn objects on the last bar only.
DRAWPOINT	DRAWPOINT(x1,y1, optional size)	Draw a point on the chart.
DRAWRAY	DRAWRAY(x1,y1,x2,y2)	Draw a ray on the chart.
DRAWRECTANGLE	DRAWRECTANGLE(x1,y1,x2,y2)	Draws a rectangle on the chart.
DRAWSEGMENT	DRAWSEGMENT(x1,y1,x2,y2)	Draws a segment on the chart.
DRAWTEXT	DRAWTEXT("your text", x1, y1)	Adds a text box on the chart at at the selected point with your text.
DRAWTRIANGLE	DRAWTRIANGLE(x1, y1, x2, y2, x3, y3)	Draws a triangle on the chart.
DRAWVLINE	DRAWVLINE(x1)	Draws a vertical line on the chart.
DynamicZoneRSIDown	DynamicZoneRSIDown[N, M]	Lower band of the Dynamic Zone RSI indicator.
DynamicZoneRSIUp	DynamicZoneRSIUp[rsiN, N]	Upper band of the Dynamic Zone RSI indicator.
DynamicZoneStochasticDown	DynamicZoneStochasticDown[N]	Lower band of the Dynamic Zone Stochastic indicator.
DynamicZoneStochasticUp	DynamicZoneStochasticUp[N]	Upper band of the Dynamic Zone Stochastic indicator.

E

CODE	SYNTAX	FUNCTION
EaseOfMovement	EaseOfMovement[I]	Ease of Movement indicator.
ElderrayBearPower	ElderrayBearPower[N](close)	Elder ray Bear Power indicator.
ElderrayBullPower	ElderrayBullPower[N](close)	Elder ray Bull Power indicator.
ELSE	See IF/THEN/ELSE/ENDIF	Instruction used to call the second condition of If-conditional statements.

CODE	SYNTAX	FUNCTION
ELSIF	See IF/THEN/ELSE/ENDIF	Stands for Else If (to be used inside of conditional loop).
EMV	EMV[N]	Ease of Movement Value indicator.
ENDIF	See IF/THEN/ELSE/ENDIF	Ending Instruction of IF-conditional statement.
EndPointAverage	EndPointAverage[N](price)	End Point Moving Average.
EXP	EXP(a)	Mathematical Function "Exponential".
ExponentialAverage	ExponentialAverage[N](price)	Exponential Moving Average.

F – G – H

CODE	SYNTAX	FUNCTION
FILLCOLOR	PRINT x FILLCOLOR (r,g,b)	Sets the background color of the corresponding cell in the print table.
FractalDimensionIndex	FractalDimensionIndex[N](close)	Fractal Dimension Index indicator.
FOR/TO/NEXT	FOR i =a TO b DO a NEXT	FOR loop (processes all the values with an ascending (TO) or a descending order (DOWNTO)).
ForceIndex	ForceIndex(price)	Force Index indicator determines who controls the market (buyer or seller).
FLOOR	FLOOR(N, M)	Returns the largest number less than N with a precision of M digits after the decimal point.
GetTimeFrame	GetTimeFrame	Returns the number of seconds equivalent to the current time period (ex: 3600 for a one hour time period).
High	High[N]	High of the current bar or of the nth last bar.
Highest	Highest[N](price)	Highest price over a number of bars to be defined.
HighestBars	HighestBars[N]	Returns the candlestick offset of the last highest value.
HISTOGRAM	RETURN close STYLE(HISTOGRAM, lineWidth)	Apply the histogram style on the returned value.
HistoricVolatility	HistoricVolatility[N](price)	Historic Volatility (or statistic volatility).
Hour	Hour[N]	Represents the hour of each closing bar loaded in the chart.
Hours	TIMEFRAME(X Hours)	Sets the period to "X Hours" for further code calculations.
HullAverage	HullAverage[N](close)	Designates the Hull Average indicator.

I - J - K

CODE	SYNTAX	FUNCTION
IF/THEN/ENDIF	IF a THEN b ENDIF	Group of conditional instructions without second instruction.
IF/THEN/ELSE/ENDIF	IF a THEN b ELSE c ENDIF	Group of conditional instructions.
IntradayBarIndex	IntradayBarIndex[N]	Counts how many bars are displayed in one day on the whole data loaded.
IsSet	IsSet($MyArray[index])	Returns 1 if the index is defined in the array. Returns 0 if the index has not been defined.
INDEX	ANCHOR(TOPLEFT,INDEX,YSHIFT)	Define the point value of the object on the horizontal axis as a barindex value.
Italic	DRAWTEXT(« text »,barindex,close,Serif,Italic, 10)	Italic style to be applied to the text.
KeltnerBandCenter	KeltnerBandCenter[N]	Central band of the Keltner indicator of N periods.
KeltnerBandDown	KeltnerBandDown[N]	Lower band of the Keltner indicator of N periods.
KeltnerBandUp	KeltnerBandUp[N]	Upper band of the Keltner indicator of N periods.
KijunSen	KijunSen[T,K,S]	Returns the KijunSen value of the Ichimoku indicator.

L

CODE	SYNTAX	FUNCTION
LastSet	LastSet($MyArray)	Returns the highest defined index from the given Array.
LEFT	ANCHOR(LEFT,INDEX,YSHIFT)	Anchor to the left of the chart.
LINE	STYLE(LINE, lineWidth)	Standard line style.
LinearRegression	LinearRegression[N](price)	Linear Regression indicator.
LinearRegressionSlope	LinearRegressionSlope[N](price)	Slope of the Linear Regression indicator.
LOG	LOG(a)	Mathematical Function "Neperian logarithm" of a.
Low	Low[N]	Low of the current bar or of the nth last bar.
Lowest	Lowest[N](price)	Lowest price over a number of bars to be defined.
LowestBars	LowestBars[N]	Returns the candlestick offset of the last lowest value.

M

CODE	SYNTAX	FUNCTION
MACD	MACD[S,L,Si](price)	Moving Average Convergence Divergence (MACD).
MACDline	MACDLine[S,L,Si](price)	MACD line indicator.
MACDSignal	MACDSignal[S,L,Si](price)	MACD Signal line indicator.
MassIndex	MassIndex[N]	Mass Index Indicator applied over N bars.
MAX	MAX(a,b)	Mathematical Function "Maximum".
MedianPrice	MedianPrice	Average of the high and the low.
MIDDLE	ANCHOR(MIDDLE,INDEX,YSHIFT)	Anchoring in the middle of the chart.
MIN	MIN(a,b)	Mathematical Function "Minimum".
Minute	Minute	Represents the minute of each closing bar loaded in the chart.
Minutes	TIMEFRAME(X Minutes)	Sets the period to "X Minutes" for the following code calculations.
MOD	a MOD b	Mathematical Function "remainder of the division".
Momentum	Momentum[N]	Momentum indicator (close – close of the nth last bar).
MoneyFlow	MoneyFlow[N](price)	MoneyFlow indicator (result between -1 and 1).
MoneyFlowIndex	MoneyFlowIndex[N]	MoneyFlow Index indicator.
Monospaced	DRAWTEXT(« text »,barindex,close,Monospaced,Italic, 10)	Monospaced font applied to text.
Month	Month[N]	Represents the month of each closing bar loaded in the chart.
Months	TIMEFRAME(X Months)	Sets the period to "X Months" for the following code calculations.
NegativeVolumeIndex	NegativeVolumeIndex[N]	Negative Volume Index indicator.
NEXT	See FOR/TO/NEXT	Ending Instruction of FOR loops.
NOT	NOT a	Logical Operator NOT.

O

CODE	SYNTAX	FUNCTION
OBV	OBV(price)	On-Balance-Volume indicator.
ONCE	ONCE VariableName = VariableValue	Introduces a definition statement which will be processed only once.
Open	Open[N]	Open price of the current candlestick or of the nth previous candlestick.
OpenDay	OpenDay[N]	Opening day of the current candlestick or the nth previous candlestick.
OpenDayOfWeek	OpenDay[N]	Day of the week of the opening of the current candlestick or the nth previous candlestick.
OpenHour	OpenHour[N]	Opening time of the current candlestick or the nth previous candlestick.

CODE	SYNTAX	FUNCTION
OpenMinute	OpenMInute[N]	Opening minute of the current candlestick or the nth previous candlestick.
OpenMonth	OpenMonth[N]	Opening month of the current candlestick or the nth previous candlestick.
OpenSecond	OpenSecond[N]	Opening second of the current candlestick or the nth previous candlestick.
OpenTime	OpenTime[N]	Time (HHMMSS) of the opening of the current candlestick or the nth previous candlestick.
OpenTimestamp	OpenTimestamp[N]	UNIX opening timestamp of the current candlestick or the nth previous candlestick.
OpenWeek	OpenWeek[N]	Opening week of the current candlestick or the nth previous candlestick.
OpenYear	OpenYear[N]	Opening year of the current candlestick or the nth previous candlestick.
OR	a OR b	Logical OR Operator.

P - Q

CODE	SYNTAX	FUNCTION
PIPSIZE	PIPSIZE	Size of a pip (forex) PIPSIZE = POINTSIZE.
PIPVALUE	PIPVALUE	Value in €/$ of a pip (or point), PipValue=Pointvalue.
POINT	RETURN close STYLE(POINT, Width)	Apply the dot style on the returned value.
POINTSIZE	POINTSIZE	Size of a pip (or point): PIPSIZE = POINTSIZE.
POINTVALUE	POINTVALUE	Value in €/$ of a pip (or point), PipValue=Pointvalue.
PositiveVolumeIndex	PositiveVolumeIndex(price)	Positive Volume Index indicator.
POW	POW(N,P)	Returns the value of N at power P.
PriceOscillator	PriceOscillator[S,L](price)	Percentage Price oscillator.
PRINT	PRINT x	Displays the variable in its own window, useful for debugging.
PRTBANDSUP	PRTBANDSUP	Gives the value of the upper band of PRTBands.
PRTBANDSDOWN	PRTBANDSDOWN	Gives the value of the lower band of PRTBands.
PRTBANDSHORTTERM	PRTBANDSSHORTTERM	Gives the value of the short-term band of PRTBands.
PRTBANDMEDIUMTERM	PRTBANDSMEDIUMTERM	Gives the value of the long-term band of PRTBands.
PVT	PVT(price)	Price Volume Trend indicator.

R

CODE	SYNTAX	FUNCTION
R2	R2[N](price)	R-Squared indicator (error rate of the linear regression on price).
RANDOM	RANDOM(Min, Max)	Generates a random integer between Min and Max bounds (included).
Range	Range[N]	Range (High – Low) of the current candle.
Repulse	Repulse[N](price)	Repulse indicator (measure the buyers and sellers force for each candlestick).
RepulseMM	RepulseMM[N,Period,factor](price)	Moving Average line of the Repulse indicator.
RETURN	RETURN Result	Instruction returning the result.
RIGHT	ANCHOR(RIGHT,INDEX,YSHIFT)	Anchor to the right of the chart.
ROC	ROC[N](price)	Price Rate of Change indicator.
RocnRoll	RocnRoll(price)	Designates the RocnRoll indicator based on the ROC indicator.
ROUND	ROUND(a)	Mathematical Function "Round a to the nearest whole number".
RSI	RSI[N](price)	Relative Strength Index indicator.

S

CODE	SYNTAX	FUNCTION
SansSerif	DRAWTEXT(« text »,barindex,close,SansSerif,Italic, 10)	SansSerif font applied to text.
SAR	SAR[At,St,Lim]	Parabolic SAR indicator.
SARatdmf	SARatdmf[At,St,Lim](price)	Smoothed Parabolic SAR indicator.
Second	Second[n]	Returns the second of the closing bar n periods before the current bar.
Seconds	TIMEFRAME(X Seconds)	Sets the period to "X Seconds" for further code calculations.
SenkouSpanA	SenkouSpanA[T,K,S]	Returns the SenkouSpanA value of the Ichimoku indicator.
SenkouSpanB	SenkouSpanB[T,K,S]	Returns the SenkouSpanB value of the Ichimoku indicator.
Serif	DRAWTEXT(« text »,barindex,close,Serif,Italic, 10)	Serif font applied to text.
SIN	SIN(a)	Mathematical Function "Sine" ('a' argument in degrees).
SGN	SGN(a)	Mathematical Function "Sign of" a (it is positive or negative).
SMI	SMI[N,SS,DS](price)	Stochastic Momentum Index indicator.
SmoothedRepulse	SmoothedRepulse[N](price)	Smoothed Repulse indicator.
SmoothedStochastic	SmoothedStochastic[N,K](price)	Smoothed Stochastic indicator.
SQUARE	SQUARE(a)	Mathematical Function "a Squared".
SQRT	SQRT(a)	Mathematical Function "Squared Root" of a.
Standard	DRAWTEXT(« text »,barindex,close,Serif,Standard, 10)	Standard style applied to text.
STD	STD[N](price)	Statistical Function "Standard Deviation".
STE	STE[N](price)	Statistical Function "Standard Error".
STYLE	STYLE(dottedline, width)	Applies the *dottedline* style type with a *width* on an object.
Stochastic	Stochastic[N,K](price)	%K line of the Stochastic indicator.
Stochasticd	Stochasticd[N,K,D](price)	%D line of the Stochastic indicator.
Summation	Summation[N](price)	Sums a certain price over the N last candlesticks.
Supertrend	Supertrend[STF,N]	Super Trend indicator.

T

CODE	SYNTAX	FUNCTION
TAN	TAN(a)	Mathematical Function "Tangent" of a ('a' argument in degrees).
TEMA	TEMA[N](price)	Triple Exponential Moving Average.
TenkanSen	TenkanSen[T,K,S]	Returns the TenkanSen value of the Ichimoku indicator.
THEN	See IF/THEN/ELSE/ENDIF	Instruction following the first condition of "IF".
Ticks	TIMEFRAME(X Ticks)	Sets the period to "X Ticks" for further code calculations. See Multi-period instructions.
Ticksize	Ticksize	Minimum price variation of the instrument in the chart.
Time	Time[N]	Represents the closing time (HHMMSS) of each bar loaded in the chart.
TimeSeriesAverage	TimeSeriesAverage[N](price)	Temporal series moving average.
Timestamp	Timestamp[N]	UNIX date of the close of the nth previous candlestick.
TO	See FOR/TO/NEXT	Directional Instruction in the "FOR" loop.
Today	Today	Today's date (YYYYMMDD).
TOP	ANCHOR(TOP,INDEX,YSHIFT)	Anchor at the top of the chart.

CODE	SYNTAX	FUNCTION
TOPLEFT	ANCHOR(TOPLEFT,INDEX,YSHIFT)	Anchor at the top left of the chart.
TOPRIGHT	ANCHOR(TOPRIGHT,INDEX,YSHIFT)	Anchor at the top right of the chart.
TotalPrice	TotalPrice[N]	(Close + Open + High + Low) / 4.
TR	TR(price)	True Range indicator.
TriangularAverage	TriangularAverage[N](price)	Triangular Moving Average.
TRIX	TRIX[N](price)	Triple Smoothed Exponential Moving Average.
TypicalPrice	TypicalPrice[N]	Represents the Typical Price (Average of the High, Low and Close).

U – V – W

CODE	SYNTAX	FUNCTION
Undefined	a = Undefined	Sets the value of a variable to undefined.
Unset	unset($MyArray)	Resets the data in the table.
VALUE	ANCHOR(TOP, INDEX, VALUE)	Sets the value of the object point for the vertical axis to be a price.
Variation	Variation(price)	Difference between the close of the last bar and the close of the current bar in %.
ViMinus	ViMinus[N]	Bottom band of the Vortex indicator.
ViPLus	ViPlus[N]	Top band of the Vortex indicator.
Volatility	Volatility[S, L]	Chaikin volatility.
Volume	Volume[N]	Volume indicator.
VolumeAdjustedAverage	VolumeAdjustedAverage[N](Price)	Volume Adjusted Moving Average.
VolumeOscillator	VolumeOscillator[S,L]	Volume Oscillator.
VolumeROC	VolumeROC[N]	Volume of the Price Rate Of Change.
Weeks	TIMEFRAME(X Weeks)	Sets the period to "X Weeks" for further code calculations.
WeightedAverage	WeightedAverage[N](price)	Weighted Moving Average indicator.
WeightedClose	WeightedClose[N]	Average of (2 * Close), (1 * High) and (1 * Low).
WEND	See WHILE/DO/WEND	Ending Instruction of WHILE loop.
WHILE/DO/WEND	WHILE (condition) DO (action) WEND	WHILE loop.
WilderAverage	WilderAverage[N](price)	Represents Wilder Moving Average.
Williams	Williams[N](close)	Returns the %R of Williams indicator.
WilliamsAccumDistr	WilliamsAccumDistr(price)	Accumulation/Distribution of Williams Indicator.

X – Y – Z

CODE	SYNTAX	FUNCTION
XOR	a XOR b	Logical Operator exclusive OR.
XSHIFT	ANCHOR(TOP, XSHIFT,VALUE)	Defines the value of the object point for the horizontal axis as an offset.
Year	Year[N]	Year of the closing bar n periods before the current bar.
Years	TIMEFRAME(X Years)	Set the period to "X Year(s)" for further code calculations.
Yesterday	Yesterday[N]	Date of the day preceding the bar n periods before the current bar.
YSHIFT	ANCHOR(TOP, INDEX, YSHIFT)	Defines the value of the object point for the vertical axis as an offset.
ZigZag	ZigZag[Zr](price)	Represents the Zig-Zag indicator introduced in the Elliott waves theory.
ZigZagPoint	ZigZagPoint[Zp](price)	Represents the Zig-Zag indicator in the Elliott waves theory calculated on Zp points.

Other

CODE	FUNCTION	CODE	FUNCTION
+	Addition Operator.	<>	Difference Operator.
-	Subtraction Operator.	<	Strict Inferiority Operator.
*	Multiplication Operator.	>	Strict Superiority Operator.
/	Division Operator.	<=	Inferiority Operator.
=	Equality Operator.	>=	Superiority Operator.


ProRealTime