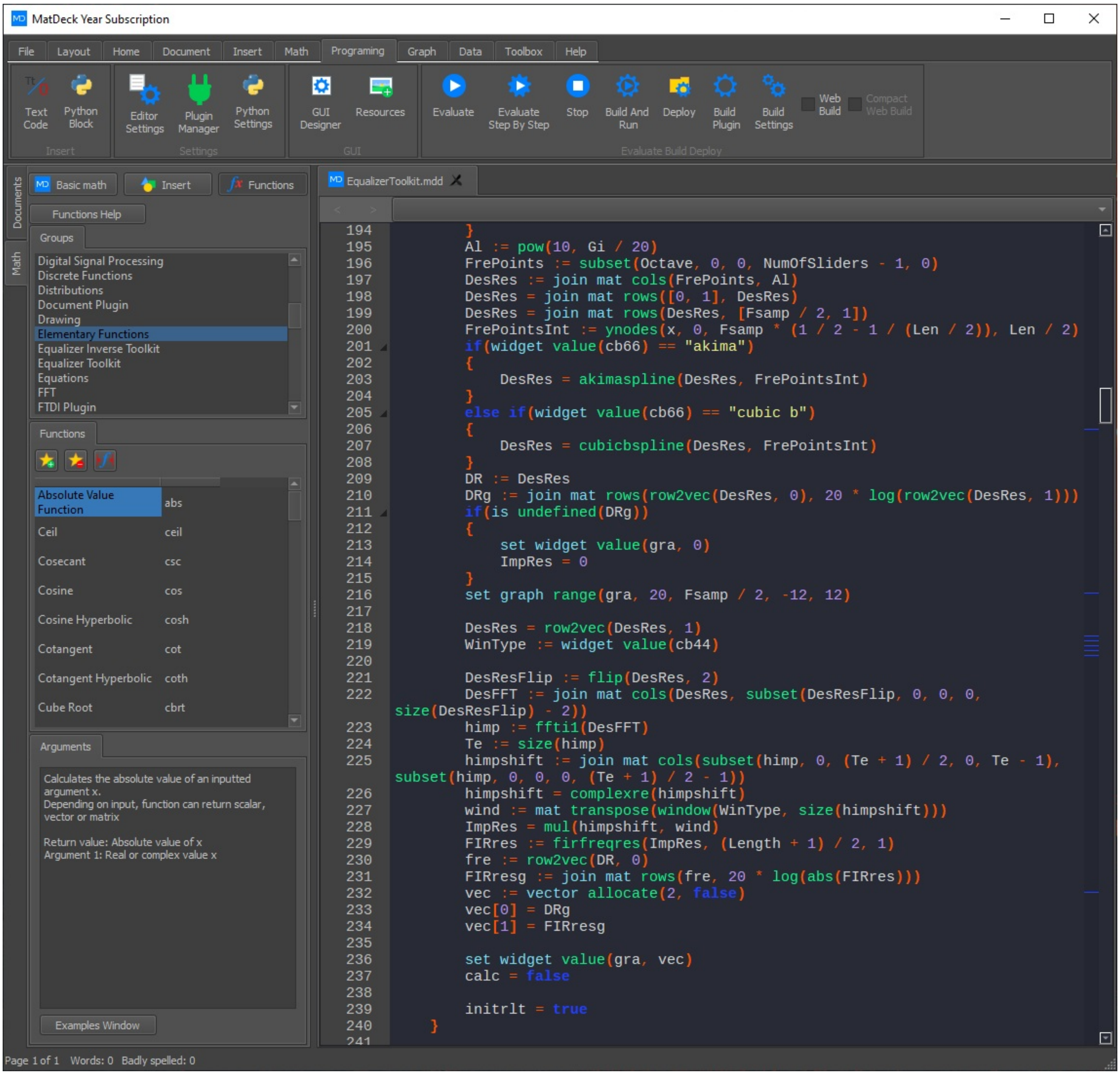
MD Script Programing

General

MD Products allows the integration of text editing, script language, the ability to generate GUIs, flowcharts, virtual instrumentation, data visualization, programing, and parallel processing all to be comprehensively done within its documents. Expressing programming ideas in MD Products takes less lines of code compared to others.



Digital Signal Processing
Discrete Functions
Distributions
Document Plugin
Drawing
Elementary Functions
Equalizer Inverse Toolkit
Equalizer Toolkit
Equations
FFT
FTDI Plugin

Absolute Value Function	abs
Ceil	ceil
Cosecant	csc
Cosine	cos
Cosine Hyperbolic	cosh
Cotangent	cot
Cotangent Hyperbolic	coth
Cube Root	cbrt

Calculates the absolute value of an inputted argument x.
Depending on input, function can return scalar, vector or matrix

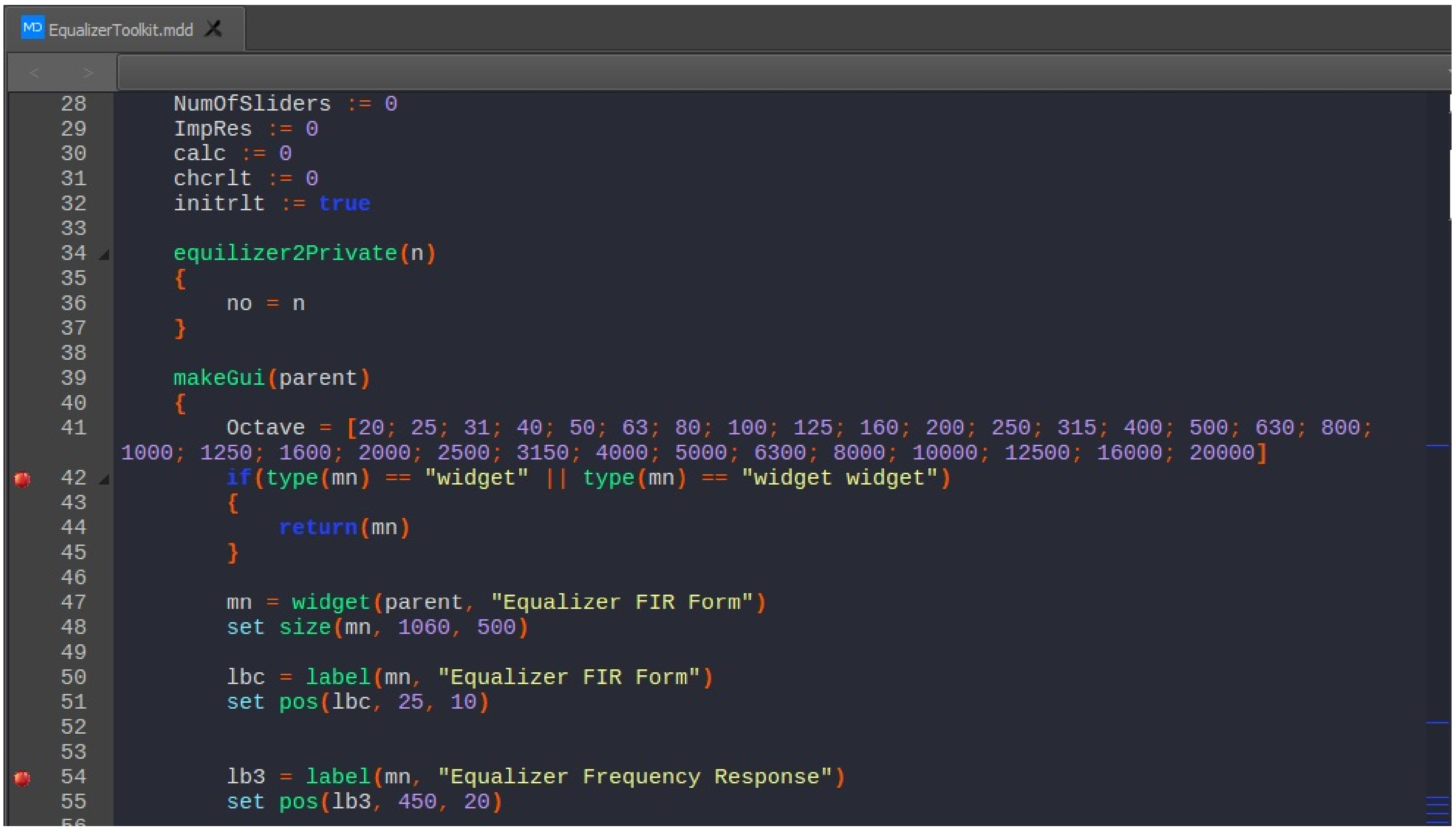
Return value: Absolute value of x
Argument 1: Real or complex value x

```
    }
    Al := pow(10, Gi / 20)
    FrePoints := subset(Octave, 0, 0, NumOfSliders - 1, 0)
    DesRes := join mat cols(FrePoints, Al)
    DesRes = join mat rows([0, 1], DesRes)
    DesRes = join mat rows(DesRes, [Fsamp / 2, 1])
    FrePointsInt := ynodes(x, 0, Fsamp * (1 / 2 - 1 / (Len / 2)), Len / 2)
    if(widget value(cb66) == "akima")
    {
      DesRes = akimaspline(DesRes, FrePointsInt)
    }
    else if(widget value(cb66) == "cubic b")
    {
      DesRes = cubicbspline(DesRes, FrePointsInt)
    }
    DR := DesRes
    DRg := join mat rows(row2vec(DesRes, 0), 20 * log(row2vec(DesRes, 1)))
    if(is undefined(DRg))
    {
      set widget value(gra, 0)
      ImpRes = 0
    }
    set graph range(gra, 20, Fsamp / 2, -12, 12)

    DesRes = row2vec(DesRes, 1)
    WinType := widget value(cb44)

    DesResFlip := flip(DesRes, 2)
    DesFFT := join mat cols(DesRes, subset(DesResFlip, 0, 0, 0,
size(DesResFlip) - 2))
    himp := ffti1(DesFFT)
    Te := size(himp)
    himpshift := join mat cols(subset(himp, 0, (Te + 1) / 2, 0, Te - 1),
subset(himp, 0, 0, 0, (Te + 1) / 2 - 1))
    himpshift := complexre(himpshift)
    wind := mat transpose(window(WinType, size(himpshift)))
    ImpRes = mul(himpshift, wind)
    FIRres := firfreqres(ImpRes, (Length + 1) / 2, 1)
    fre := row2vec(DR, 0)
    FIRresg := join mat rows(fre, 20 * log(abs(FIRres)))
    vec := vector allocate(2, false)
    vec[0] = DRg
    vec[1] = FIRresg

    set widget value(gra, vec)
    calc = false

    initrlt = true
  }
```

Page 1 of 1 Words: 0 Badly spelled: 0

Above we can see an example of the MatDeck Script IDE which is the perfect environment for programming and coding custom GUIs, applications and other large quantities of code. The IDE has features such as Breakpoints,Debugging and several different execution options.

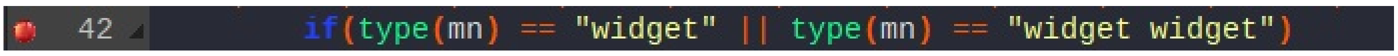
Breakpoints



EqualizerToolkit.mdd

```
    NumOfSliders := 0
    ImpRes := 0
    calc := 0
    chcrlt := 0
    initrlt := true

    equilizer2Private(n)
    {
        no = n
    }

    makeGui(parent)
    {
        Octave = [20; 25; 31; 40; 50; 63; 80; 100; 125; 160; 200; 250; 315; 400; 500; 630; 800;
1000; 1250; 1600; 2000; 2500; 3150; 4000; 5000; 6300; 8000; 10000; 12500; 16000; 20000]
        if(type(mn) == "widget" || type(mn) == "widget widget")
        {
            return(mn)
        }

        mn = widget(parent, "Equalizer FIR Form")
        set size(mn, 1060, 500)

        lbc = label(mn, "Equalizer FIR Form")
        set pos(lbc, 25, 10)


        lb3 = label(mn, "Equalizer Frequency Response")
        set pos(lb3, 450, 20)
```

Breakpoints in MD Products are selected lines in Python code where the IDE will stop compiling. To add a Breakpoint right click on the left-hand side of number in the programming line, to remove them need to click on the red dot. Breakpoints are illustrated by red dots next to the code line number.

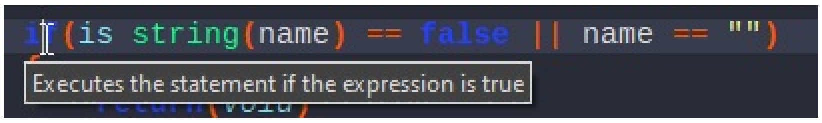


```
42        if(type(mn) == "widget" || type(mn) == "widget widget")
```

The IDE will compile each line of code until a Breakpoint is reached. Once, a Breakpoint has been reached, the user will have to evaluate the program. Similarly, the IDE will compile all the code after the Breakpoint until it reaches another Breakpoint. To reach a Breakpoint and execute the code, you will need to Evaluate the document.

Extended Tooltips

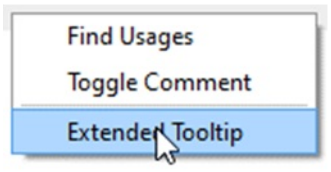
Extended Tooltips are another MatDeck Script IDE feature. They give the user information on any Python function that they hover there mouse over.



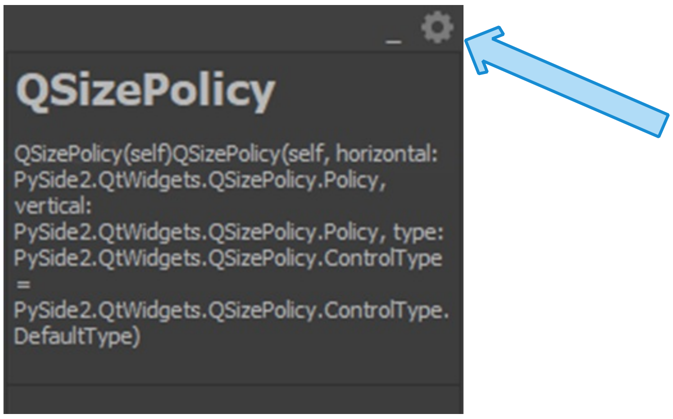
```
if(is string(name) == false || name == "")
```

Executes the statement if the expression is true

As you can see when we hover over the if() function, our tooltip appear giving us information on what the function does. We can also have the Extended Tooltip as a pop-up or even docked on the left-hand side of the IDE. To do this we need to left click on the IDE and select the Extended Tooltip Option.



Once the Extended Tooltip is on, you can change how the tooltip is present using the settings icon on the top right hand side of any Extended Tooltip pop-up.



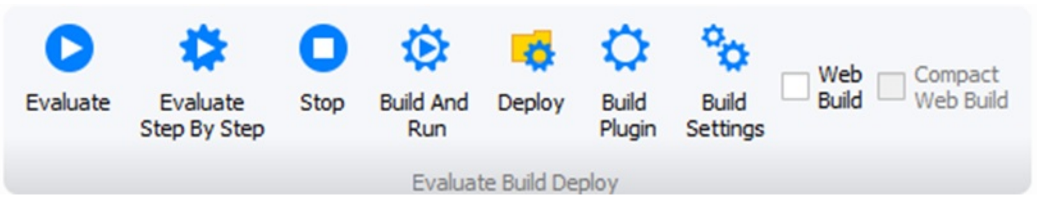
QSizePolicy

```
QSizePolicy(self)QSizePolicy(self, horizontal:
PySide2.QtWidgets.QSizePolicy.Policy,
vertical:
PySide2.QtWidgets.QSizePolicy.Policy, type:
PySide2.QtWidgets.QSizePolicy.ControlType
=
PySide2.QtWidgets.QSizePolicy.ControlType.
DefaultType)
```

Above we can see the Docking in dark mode, as mentioned before to change or close it you will need to click the settings icon on the top right hand side.

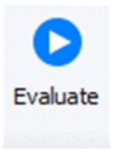
Programming Toolbar

In all of our IDE, we have several options to run and evaluate your MatDeck Script code.



Evaluate

The first option and most conventional method is the Evaluate button, it will run the MatDeck Script program without any interruptions or modifications.



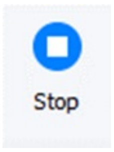
Please Note: The Evaluate button will offer debugging information such as any syntax error whereas Build and Run will only offer this information during the Build and Run and not during the runtime.

Evaluate Step By Step

The other method is to use the Evaluate Step By Step button. This will run a debugger on the MatDeck Script code while stopping at intervals where code blocks/functions or key points in your code. This is done to better help you locate and resolve any bugs or logic issues.

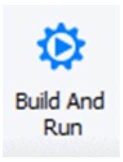
Stop

This is used to cease any kind of evaluation that is occurring on your MatDeck Script code. It also works to stop the debugger and the Evaluate Step By Step.



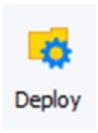
Build And Run

Evaluates the program as a sperate and independent application without creating a .exe file, it has the same effect as Evaluate but the program will not be run as a sub-program of MatDeck and the application will be allocated separate memory and run as the same speed as a .exe file, whereas Evaluate will run the program slower. However, Evaluate will start the program faster compared to Build and Evaluate, but the program itself will always be faster with Build and Run.



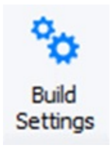
Deploy

The Deploy button will package your MatDeck Script code as a .exe application file. This will allow you to share your code as an independent application on an unlimited amount of windows devices. Please note that you will need to add any additional files your application needs using the Build Settings Button. This will not run the program



Build Settings

Build Settings is used to configure your code when it is packaged into a application. Here you will need to add any additional files that your code will call. This includes any images, MD files or any other file.



You can also choose whether or not you will deploy any MD Instruments as Images or as GUI Widgets.

MD Document

Below we can see the MD Document which is perfect for visual and interactive computing and much less code heavy. The screen shot itself is of the MatDeck Script Manual which you are reading now.

The main objectives of this manual are: give general tips and suggestions about how to program in MD Products, teach enough MatDeck script so that it is easy to do most data manipulations, analysis, and comparison, as well as to a provide firm knowledge foundation to learn more advance MD techniques.

Editing Code

There are three ways to edit script code and programming in MD Products: Math objects within a canvas, writing code in text mode, or the MD IDEs.

The first two options are used within a regular MD Document and MD IDEs are a dedicated code editing document. For short, simple and interactive mathematical and programing calculations, you should uset he

Canvas and Math objects (see Insert and Math tab)found within a MD document.

```
edit_code( )
{
    print("Hello world")
}
```

Make programm

```
edit_code( )
```

Call function

For more complex programing within a MD document, MD Products provide code editing via the text mode. It is enabled by clicking Text/Code icon in the Math Tab, or using **ctrl + i**. The code lines are numbered as seen here:

```
1 // Edit code here
2 varaiable := 5
```

In order to get back to text editing you must first click the Text/Code icon in the Math Tab, or use **ctrl + i**.

For more complex programing, MD Products provides the alternate option, MD IDEs. The IDEs are generated by using File-New and then the drop down tool bar next to it select the MatDeck Script option. The MatDeck IDE is dedicated for programming and it can contain only code as text .

MatDeck documents are evaluated every time after a **=** or a new line key is pressed in the canvas and you will get the results immediately after. Also you can click on the document, use **ctrl + e** or the Evaluate button to explicitly evaluate and execute codes within the document at any point in time.
MD documents and scripts can be compiled into executable applications. Behavior of such applications will be the same as in the source script but it will run much faster. For compiling processes, you should use the Build And Run button from the tool bar. You may be asked to set a development kit if one is not present.

Syntax

MatDeck Script is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters. Also keep in mind that script is executed from the left to the right and from the top to the bottom.

Data Types

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the types of values that can be represented and manipulated in a programming language.

MatDeck Script allows you to work with these data types:

- boolean
- integer
- double
- complex
- symbolic value
- string
- symbolic function
- unit
- vector

- matrix
- image
- expression
- equation
- interval
- object

MatDeck Script also defines the trivial data type undefined (void).

Composite data types like vectors, matrices, equations and intervals are composed of primitive data types and so, in canvas mode you need to enter its keyword before you can enter its data.

$\text{type}(\text{true}) = \text{"boolean"}$

$\text{type}(3) = \text{"integer"}$

$\text{type}(3.5) = \text{"double"}$

$\text{type}(4 + 7i) = \text{"complex"}$

$\text{type}(a) = \text{"symbolic value"}$

$\text{type}(\text{"a"}) = \text{"string"}$

$\text{type}\left(\sin(x)\right) = \text{"symbolic function"}$

$\text{type}(a + b) = \text{"expression"}$

$\text{type}\left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right) = \text{"matrix"}$

$\text{type}\left(\begin{bmatrix} 3 \\ 8 \end{bmatrix}\right) = \text{"vector"}$

$\text{type}(\text{void}) = \text{"undefined"}$

$\text{type}(\square) = \text{"undefined"}$

$\text{type}(m) = \text{"symbolic value"}$

$\text{type}(x == 3) = \text{"equation"}$

$\text{type}((\,3\,,5\,]) = \text{"interval"}$

Variables

Like many other programming languages, MatDeck Script has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Variable declaration and initialization

Before you use a variable you must declare it. Unlike many other languages, you don't have to tell MatDeck during variable declaration what type of value the variable will hold. Variables can hold a value of any data type. Variables will get its initial value type during variable declaration and it can be changed during the execution of a program.

Variables are declared with the **:=** operator as follows.

```
a:=7
name:="Your name"
type(a) = "integer"
type(name) = "string"
```

Storing a value in a variable is called **variable initialization**. You should do variable initialization at the time of variable creation. Also you can change the variable value later with the variable assignment operator **=**.

```
name = "My Name"
name = "My Name"
name = "My Name"
```

when entered variable assignment operator v= looks like = but bold

You can re-declare the same variable again but it is good practice to use variable assignment operator like in the example above.

Variable scope

The scope of a variable is the region of your code or document in which it is defined. MatDeck Script variables have two scopes:

- **Global** scope − A global variable has a global scope which means it can be defined anywhere in your code or document.
- **Local** scope − A local variable will be visible only within a function where it is defined. Function arguments are always local to that function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function argument with the same name as a global variable, you effectively hide the global variable. Take a look into the following example.

```
var:="global variable"
fn( )
{
   var:="local variable"
   return(var)                    fn( ) = "local variable"
}
```

Variable names

While naming your variables in MatDeck Script, keep the following rules in mind:

- You should not use any of the MatDeck Script reserved keywords as a variable name.
- Variable names should not start with a numeral (0-9). They must begin with a letter.
- Variable names are case-sensitive. For example, **Name** and **name** are two different variables.

Operators

An operator is a symbol that tells the an MD Product to perform specific mathematical or logical operation. MatDeck Script operators can also perform operations between different data types if it is possible.

MatDeck Script supports the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Compound Assignment Operators
- Subscript Operator

Arithmetic Operators

The following arithmetic operators are supported by MatDeck Script:

- + Adds two operands
- - Subtract second operant from the first
- * Multiplies both operands
- / Divides numerator by denominator
- % Division reminder

$2 \cdot 5 = 10$

$"I" + "am" = "I am"$

$"Cat" - "t" = \text{undefined}$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 2 = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Relational Operators

The following relational operators are supported by MatDeck Script:

- **==** Checks if the values of two operands are equal or not, if yes then condition becomes true.
- **!=** Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
- **>** Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
- **<** Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
- **>=** Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
- **<=** Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

$x > y = \text{false}$

$x < y = \text{false}$

$7 > 5 = \text{true}$

$\text{true} == 1 = \text{true}$

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} == \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \text{true} \quad \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} > \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} = \text{false}$$

Logical Operators

The following logical operators are supported by MatDeck Script:

- **&&** Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
- **||** Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
- **!** Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.

Bitwise Operators

The following bitwise operators are supported by MatDeck Script:

- **&** Bitwise AND
- **^** Bitwise HOR
- **|** Bitwise OR
- **<<** Bitwise Shift Left
- **>>** Bitwise Shift Right
- **~** Bitwise NOT

Compound Assignment Operators

- **+=** Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.
- **-=** Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
- ***=** Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
- **/=** Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.
- **%=** Modulus assignment
- **&=** Bitwise AND assignment
- **^=** Bitwise HOR assignment
- **|=** Bitwise OR assignment
- **<<=** Bitwise shift left assignment
- **>>=** Bitwise shift right assignment

Subscript Operator

[ ] gives access to the vector or matrix element

$$a := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad \begin{bmatrix} 5 \\ a \end{bmatrix}[1] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$a[0] = 1$$

$$a[0] = 5$$

$$a[0] = 5$$

$$a = \begin{bmatrix} 5 & 2 \\ 3 & 4 \end{bmatrix} \qquad \begin{bmatrix} 5 \\ a \end{bmatrix}[1] = \begin{bmatrix} 5 & 2 \\ 3 & 4 \end{bmatrix}$$

Operators Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.
Operators with the highest precedence appear at the top of the list, those with the lowest appear at the

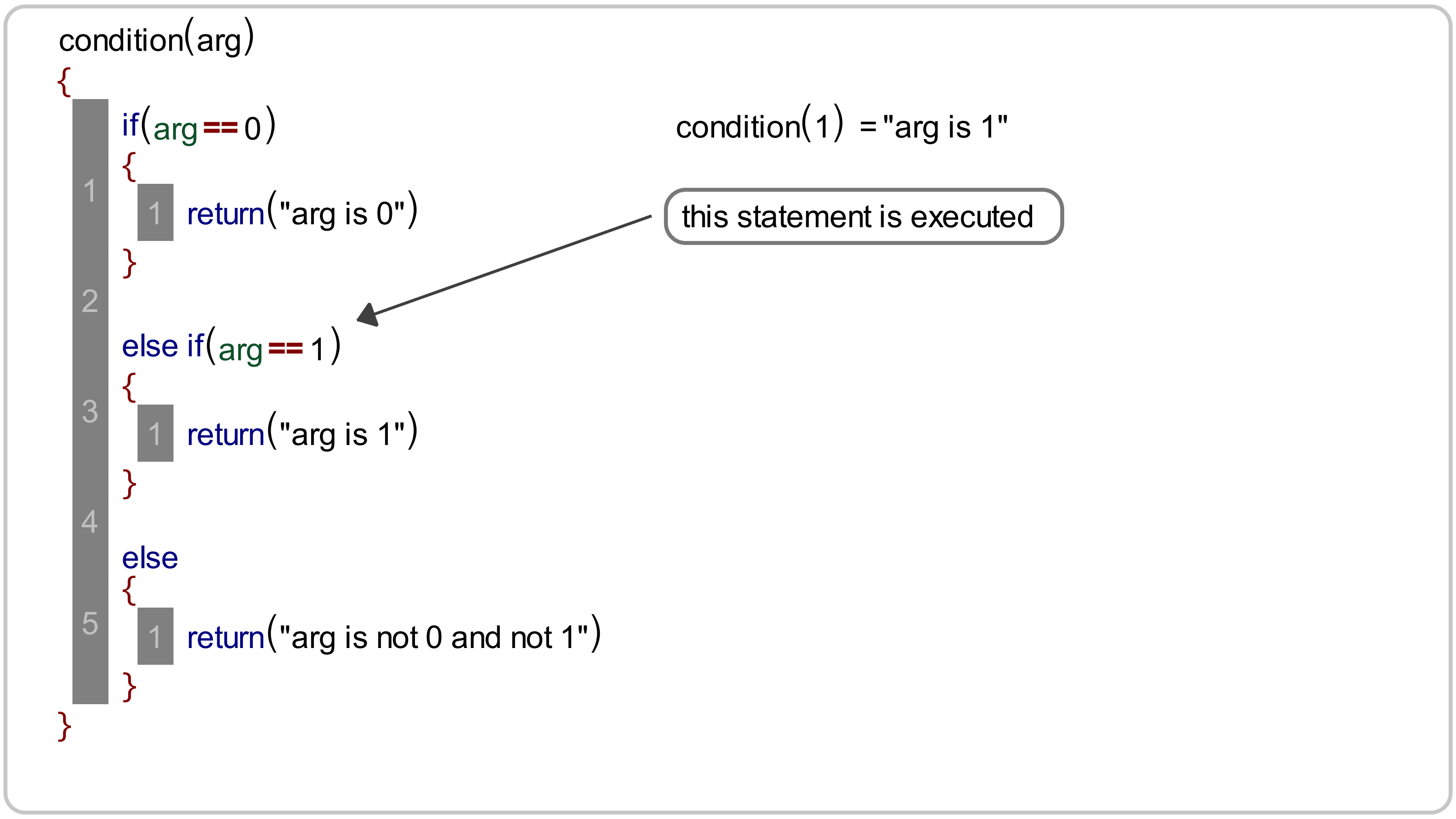
bottom. Within an expression, higher precedence operators will be evaluated first.

- **[ ]**
- **! ~ * &** (last two are dereference and address-of)
- *** / %**
- **+ -**
- **<< >>**
- **< > <= >=**
- **== !=**
- **&**
- **^**
- **|**
- **&&**
- **||**
- **+= -= *= /= %= <<= >>= &= ^= |=**
- **=**

Control Statements: if, else if and else

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions.
MatDeck Script supports conditional statements which are used to perform different actions based on different conditions. These statements are:

- if statement is the fundamental control statement that allows MatDeck Script to make decisions and execute statements conditionally.
- else if (can be used only after if)
- else (can be used only after if or else if)



```
condition(arg)
{
    if(arg == 0)
    {
        return("arg is 0")
    }

    else if(arg == 1)
    {
        return("arg is 1")
    }

    else
    {
        return("arg is not 0 and not 1")
    }
}
```

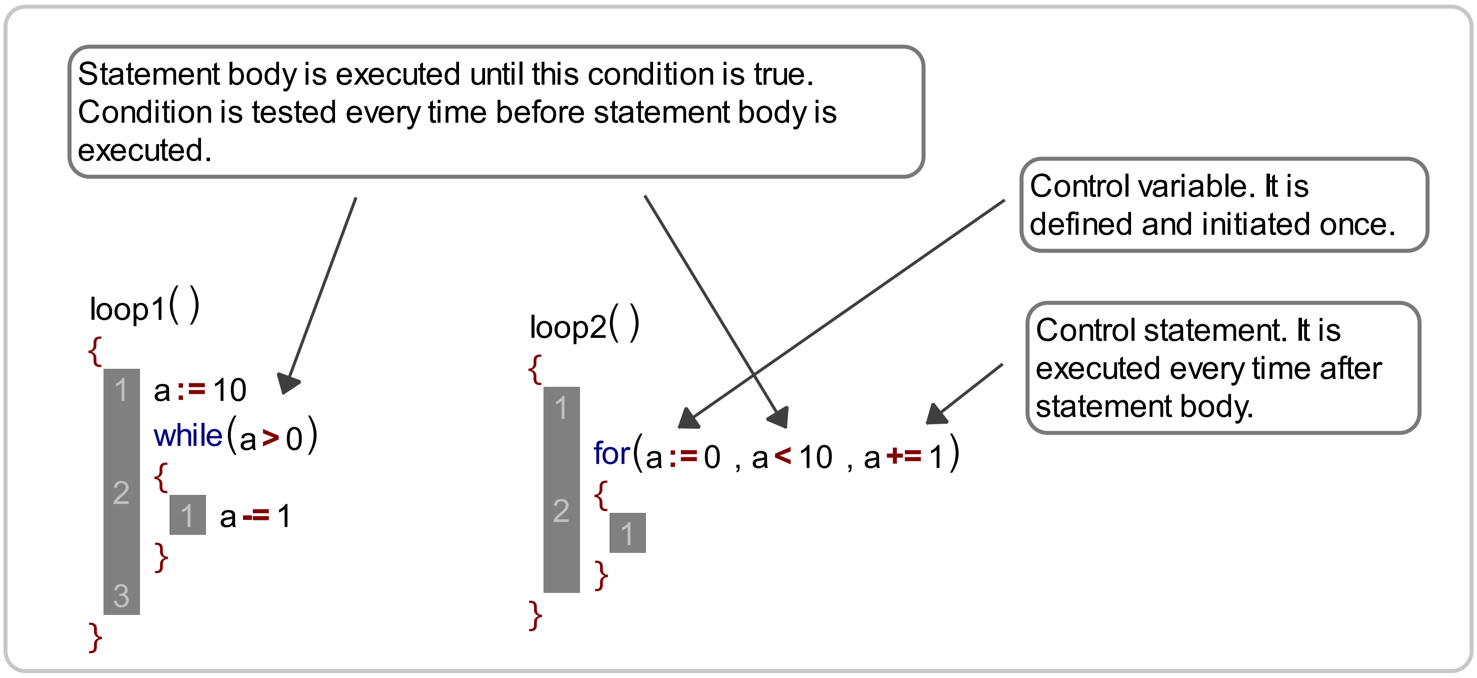
```
condition(1) = "arg is 1"
```

Loops

There may be a situation, when you need to execute a block of code several times. In general, statements are executed sequentially, the first statement is executed first, followed by the second, and so on.

MatDeck Script provides the following type of loops to handle looping requirements:

- **while** loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
- **for** loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable


Statement body is executed until this condition is true. Condition is tested every time before statement body is executed.
Control variable. It is defined and initiated once.
Control statement. It is executed every time after statement body.

```
loop1( )
{
    a := 10
    while(a > 0)
    {
        a -= 1
    }
}

loop2( )
{
    for(a := 0 , a < 10 , a += 1)
    {
    }
}
```

Loop Control

There may be a situation when you need to come out of a loop without reaching its bottom. There may also be a situation when you want to skip a part of your code block and start the next iteration of the loop.
To handle all such situations, MatDeck Script provides **break and continue statements**. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

- **break** statement is used to exit a loop early, breaking out of the enclosing curly braces.
- **continue** statement starts the next iteration of the loop and skip the remaining code block. When a **continue** statement is encountered, the program flow moves to the loop check expression immediately and if the condition remains true, then it starts the next iteration, otherwise the control comes out of the loop.

```
// Sum of numbers from 10 to 20
loopcontol()
{
  a := 0

  for(n := 0; n < 30; n += 1)
  {

    if(n < 10)
    {
      continue
    }
    a += n
    if(n >= 20)
    {
      break
    }
  }
  return(a)
}
myfn()
{

}
```

We can define a function with function arguments. Arguments are values passed to the function body.

```
myFirstFunction(arg1, arg2)
{
  return(arg1 + arg2)
}
```

Calling a function

To invoke a function somewhere later in the script, you would simply need to write the name of that function as shown in the following code.

$\text{myFirstFunction}(2\ ,\ 3) = 5$

$s := \text{myFirstFunction}(5\ ,\ 7)$

$s = 12$

Return statement

A MatDeck Script function can have an optional **return statement**. This is required if you want to return a value from a function. Functions without return statement will **return void** (undefined).

Classes

Classes and are often called user-defined types. A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

Class definition

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by pair of curly braces.

Define an object

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. In that process a class constructor (main function with the same name as class) is executed.

The public data members of objects of a class can be accessed using the direct member access operator **(.)**

Functions

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers to write modular code. Functions allow a programmer to divide a big program into a number of small and manageable functions. Also control and loop statements are available only in functions, if the code is used in the Canvas.

Defining a function

Before we use a function, we need to define it. To define a function in a new line, enter the function name and then follow with closed brackets **()**. After that, press the enter key and in the new line add a **{** bracket and press enter again. The MD Product will add the closing **}** bracket. The space between the **{}** brackets is called the function body.

```
class rectangle
{
  wid := 0
  hei := 0
  rectangle(w, h)
  {
    wid = w
    hei = h
  }

  surface()
  {
    return(wid * hei)
  }
}
```

```
r1 := rectangle(3 , 4)
r2 := rectangle(2 , 2)
r1 . surface( ) = 12
r2 . surface( ) = 4
r1.wid = 3
r1.hei = 4
r2.wid = 2
r2.hei = 2
```

Special Programming statements and functions

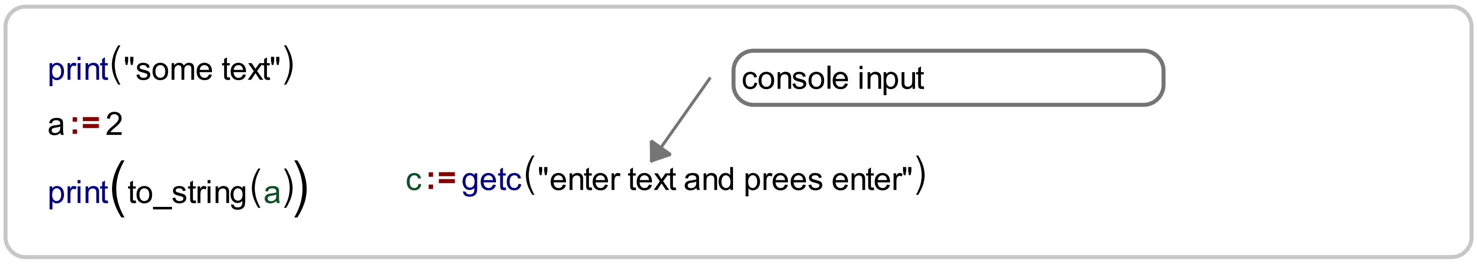
Plug-in name statement

Plug-in naming. For more see plug-in user manual.

```
plugin_name("my plugin")
```

Console input and output

All MatDeck scripts can be compiled into executable files. You can use the following functions to print to the console and get user input from the console.



```
print("some text")
a:=2
print(to_string(a))                c:=getc("enter text and prees enter")
```

In the document or script **getc()** will return its type argument as return value and **print()** will do nothing.

Programming Examples

- Minimum and maximum of vector (.mdd), (.pdf)
- Recursion (.mdd), (.pdf)
- Newton - Raphson method (.mdd), (.pdf)
- Secant method (.mdd), (.pdf)
- Class (.mdd), (.pdf)