farique1 / msx-basic-dignified

master | Go to file | Add file | Code

Fred Rique and **Fred Rique** Dignif... ... on 17 Jul 101

Examples	v1.5 Python 3 and +	7 months ago
Images	v1.5 Python 3 and +	7 months ago
Versions	v1.5 Python 3 and +	7 months ago
.gitignore	v1.2	12 months ago
DignifieR.md	Dignifier v1.2	2 months ago
MSXBader....	Dignifier v1.2	2 months ago
README.md	Added DignifieR	2 months ago
changelog.md	v1.5 Python 3 and +	7 months ago
msxbadig.ini	test	2 months ago
msxbadig.py	v1.5 Python 3 and +	7 months ago

About

Convert modern styled MSX Basic to native format

#msx #msx-basic #basic #conversion #modernization

Readme

Languages

- **Python** 100.0%

README.md

MSX Basic Dignified

v1.5 - Python 3.8!

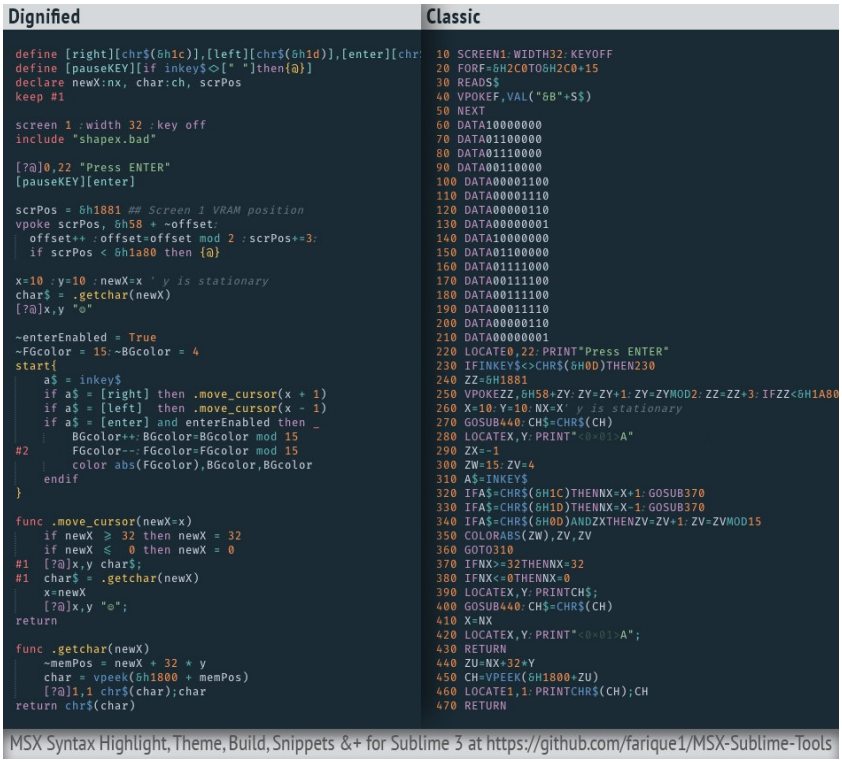
MSX Basic Dignified is a 'dialect' of MSX Basic using modern coding style and standards that can be composed on any text editor and converted to the traditional MSX Basic to be executed.

Dignified Basic can be written **without line numbers**, can be **indented** using TABs or spaces and have **broken lines**, can use variables with **long names**, have macros **defined** and external files **included**, has a kind of **function** construct, can use **unicode special characters** analogues to the MSX ASCII ones, as well as **line toggles**, **true** and **false** statements, **compound arithmetic** operators and more.

> To convert FROM the Classic Basic TO the Dignified format see **MSX Basic DignifieR**.

> I felt the need for something like this when I was redoing **Change Graph Kit**, an MSX program I did back in the day, just to see how much I could improve it. I coded on Sublime and used Excel, REM tags on branching instructions and a lot of patience to add and change line numbers and such, not pretty.
> Long after the **CGK** episode I discovered Tabmegx and also found Inliner (GitHub) during research for **MBD**, (NestorPreTer is yet another similar tool I found later). They were great sources of 'inspiration' but I wanted something even closer to the workflow I had been working on with other languages.

There is a syntax highlight, theme, build system, snippets and comment setting available for **Sublime Text 3** working with the **MSX Basic** and **MSX Basic Dignified** rules to help improve the overall experience.



> Please, be aware that **MBD** and its tools are by no mean finished products and are sort of expected to misbehave.

The Program

Standard behaviour

MSX Basic Dignified reads a text file containing the Dignified code and write back traditional **MSX Basic** in ASCII and/or tokenized format.
The Dignified code uses a `.bad` extension, the ASCII is given `.asc` and the tokenized is `.bas`.

> One of the flavours of **MSX Basic Tokenizer** must be present for tokenized output.

If **MSX Basic Tokenizer** is used, a list file (like the ones on assemblers) can be exported showing some statistics and the correspondence of the tokenized code with the ASCII one. A line from the `.mlt` file would be:

```
80da: ee80 7800 44 49 ef 50 49 f2 1f 41 31 41 59 2(
```

(more information on the **MSX Basic Tokenizer** page.)

If running from the build system on **MSX Sublime Tools**, the code execution can be monitored on **openMSX**; Basic errors are caught by **Sublime** and the offending lines will be tagged.

Unlike the traditional MSX Basic, instructions, functions and variables must be separated by spaces from alphanumeric characters. The MSX Syntax Highlight will reflect this and there are several settings to conform the spacing when the conversion is made. Dignified code should always end with a blank line.

Note: When saving a Basic program as ASCII from **openMSX** (`save"file",a`) the text encode used is the `Western (Windows 1252)` . This is the format **MSX Basic Dignified** uses to encode the `.asc` conversion and is the default encode of the `.bad` format as well. If **unicode translation** and special characters are used, the `.bad` format should be in the `UTF-8` format. The conversion will still generate a `Western (Windows 1252)` file with the correct special characters translated to the MSX ASCII format.

Features and usage

Run with: `msxbadig.py <source> [destination] args...`

> From now on, when showing code, usually the first excerpt is Dignified, followed by the program call and the traditional Basic output. For the sake of clarity only the arguments relevant to the current topic will be shown on the program call.

-

Directing the code flow is done with **labels**.
Labels are created using curly brackets `{like_this}` and can be used alone on a line to receive the code flow or on a branching (jump) instruction to direct the flow to the corresponding line label. They can only have letters, numbers and underscore and they cannot be only numbers. `{@}` points to its own line (abraço, Giovanni!).
A special kind of label can be used to create a concise closed loop. It is opened with `label{` and closed with an `}` (alone on a line). The opening label works like any regular label and the closing one will be replaced with `GOTO {label}`. Loop labels can be nested.
Labels marking a section are called Line Labels and labels on jump instructions (`GOTO` , `GOSUB` , etc) are called Branching Labels. Closed loop labels are calle Loop Labels.
A comment can be used after a line label using a `'` (`REM` is not supported) and it will stay after the conversion as an addition to the label or alone on the line, depending on the label conversion argument used.
Labels not following the naming convention, duplicated line labels, labels branching to inexistent line labels and loop labels not closed will generate an error and stop the conversion. Labels with illegal characters are higlighted when using the MSX Syntax Highlight.

```
{hello_loop}
print "press A to toggle"
if inkey$ <> "A" then goto {@}
hello{
        print "hello world"
        if inkey$ = "A" then goto {hello_loop}
}
```

```
msxbadig.py test.bad
```

```
10 PRINT "press A to toggle"
20 IF INKEY$<>"A" THEN 20
```

```
30 PRINT "hello world"
40 IF INKEY$="A" THEN 10
50 GOTO 30
```

- **Defines** are used to create aliases on the code that are replaced when the conversion is made. They are created with `define [name] [content]` where the `content` will replace the `[name]`. There can be as many as needed and there can be several on the same line, separated by commas: `define [name1] [content1], [name2] [content2], [name3] [content3]`.
  Duplicated **defines** will give an error and stop the conversion.
  A `[]` inside a **define** content is a **define** variable and will be substituted by an argument touching the closing **define** bracket. If there is content inside the variable bracket, the text will be used as default in case no argument is found.
  For instance, using `DEFINE [var] [poke 100,[10]]`, a subsequent `[var]30` will be replaced by `poke 100,30`; `[var]` alone will be replaced by `poke 100,10`.
  Variables passed to **defines** must be touching the closing bracket and be followed by an space, indicating the end of the variable content. A **define** that takes a variable but does not want one must also be followed by an space after the closing bracket, indicating no variable is given.
  Defines can be used as variables for other **defines** but they cannot be the same and a **define** with a variable as a variable for a **define** is a very finicky situation and should be used cautiously.
  `[?@]x,y` is a built in **define** that becomes `LOCATEx,y:PRINT`, it must be followed by an empty space.

  > `[...]` constructs inside `REM`, `DATA` or `""` will be identified as DEFINEs.

  ```
  define [ifa] [if a$ = ], [enter] [chr$(13)]
  define [pause][if inkey$<>[" "] goto {@}]
  ```

```
    [ifa]"2" then print "dois"
    [ifa]"4" then print "quatro"
    [?@]10,10 "dez"
    [pause][enter]
```

```
msxbadig.py test.bad
```

```
10 IF A$="2" THEN PRINT "dois"
20 IF A$="4" THEN PRINT "quatro"
30 LOCATE 10,10:PRINT "dez"
40 IF INKEY$<>CHR$(13) GOTO 40
```

Long name variables can be used on the Dignified code. They can only have letters, numbers and underscore and cannot be only numbers or have less than 3 characters. As is the case with the instructions, they must be separated by empty spaces from other alphanumeric characters or commands (the non alpha characters `~` , `$` , `%` , `!` , `#` on the variables can touch other commands).
When converted they are replaced by an associated standard two letter variables. They are assigned on a descending order from `ZZ` to `AA` and single letters and letter+number are never used. Each long name is assigned to a short name independent of type, so if `variable1` becomes `XX` so will `variable1$` become `XX$`
To use these variables they must be declared, there are two ways to do that:
-- On a `declare` line, separated by commas: `declare variable1, variable2, variable3` .
-- In place, preceded by an `~` : `~long_var = 3` .
An explicit assignment between a long and a short name can be forced by using a `:` when declaring a variable (only on a `declare` command): `declare variable:va` will assign `VA` to `variable` . Forcing declaration of the same variable to different short or long names will cause an error.
As variables are assigned independent of type, explicit type character (`$%!#`) cannot be used on a `declare` line.
When a long name variable is declared, a new two letter variable is assigned to it, redeclaring the variable will maintain the previous assignment. A warning will be given for repeated declarations of the same variable. Variables declared on `declare` lines have precedence over `~` declarations.
After it is assigned, a long name variable can be used without the `~` but a text without the `~` that has not been previously declared as a variable will be taken as a normal Basic instruction and will probably give a syntax error.
Reserved MSX Basic commands should not be used as variables names as they will be assigned and

converted.
Traditional one and two letters variables can be used normally alongside long names ones, just be aware that the letters at the end of the alphabet are being used up and they may clash with the hard coded ones.
The conversion (and the MSX Syntax Highlight) will catch illegal variables when declaring but it is not always perfect on `~` so keep an eye on them.
A summary of the long and short name associations can be generated on `REM` s at the end of the converted code.

```
declare food, drink:dk
if food$ = "cake" and drink = 3 then _
       ~result$ = "belly full":
       ~sleep = 10
endif
```

```
msxbadig.py test.bad
```

```
10 IF ZZ$="cake" AND DK=3 THEN ZY$="belly ful
```

Optional, with `msxbadig.py test.bad -vs`

```
20 ' ZZ-food, ZY-result, ZX-sleep, DK-drink
```

Proto-functions can be used to emulate the use of modern function definition and calls.
They are defined with `func .functionName(arg1, arg2, etc)` and must end with a `return` alone on a line.
The arguments can have default values as in `func .function(arg$="teste")` and the `return` can have return variables like `return ret1, ret2, etc` . The `return` and can also have a `:` before it or at the end of the line above to join them on the conversion.
The functions are called with `.functionName(arg1, arg2, etc)` and can be assigned to variables like `ret1, ret2 = .functionName(args)` . They can be separated by `:` as usual and can also come after a `THEN` or `ELSE` : `if a=1 then .doStuff() else .dontDoStuff()` .
The number of arguments and return variables must be the same and explicitly given except for an argument with a default value, in this case the default will be used if a empty space is passed on that position. To use a function call with empty argument positions and preserve the variable value, the default must be the variable name, eg: `func .applyColor(color1=color1, color2=color2)` can be called with `.applyColor(10,)` , `.applyColor(,20)` or even `.applyColor(,)` and the color not specified will be maintained.
Proto-functions cannot have multiple conditional returns (only the first `return` alone on a line will be parsed for variables and will signal the end of the function definition) but the conditional value can be established a priori in a variable on an `IF THEN ELSE` that can be passed on to the `return` .
Obviously there are no local variables on the MSX Basic (which can limit the usefulness of the proto-functions) but this can be simulated by using unique named variables inside the functions. **Proto-functions** can also be useful to apply different results to different variables at different points in the code.
When converting to traditional Basic, function

definitions are essentially **labels** so they cannot have the same name as one of them. A function call is a `gosub` to that label with the variables assigned before and after it accordingly.
If the arguments or return variable are the same between function calls or definitions they will not be equated on the conversion to avoid unnecessary repetition like `A$=A$` . The same way, an empty argument location with a default variable equated to itself will not be converted.
Different from a normal function, `func` definitions will not deviate the code flow from itself so they must be placed at a point unreachable by the normal code flow.
Everything on the same line as the function definition will be converted to a comment. `##` comments will be ignored as usual.
Upon conversion, function definitions and calls will follow the **label** configurations.

> Some known limitations:
> Function recursion (called inside itself) is not recommended.
> **Proto-function** definitions and calls cannot have the `_` line separator.
> `.xxx()` constructs inside `REM` , `DATA` or `""` will be identified as function calls.

```
ch$ = .getUpper("a")
print ch$
end
func .getUpper(up$)
       ch = asc(up$) - 32
return chr$(ch)
```

```
msxbadig.py test.bad
```

```
10 UP$="a":GOSUB 40:CH$=CHR$(CH)
20 PRINT CH$
30 END
40 ' {getUpper}
50 CH=ASC(UP$)-32
60 RETURN
```

- A single traditional MSX Basic line can span several lines on Dignified code using `:` or `_` breaks. They are then **joined** when converting to form a single line.
  Colons `:` can be used at the end of a line, to join the next one, or at the beginning of a line, to join it to the previous one, and are retained when joined. Their function is the same as on the traditional MSX Basic, separating different instructions.
  Underscores `_` can only be used at the end of a line and they are deleted when the line is joined. They are useful to join broken instructions like `IF THEN ELSE`, long quotes or anything that must form a single command on the converted code.
  `endif`s can be used (not obligatory, just Python it) but are for cosmetical or organisational purpose only. They must be alone on their lines and are removed upon conversion without any validation regarding their `IF`s. If they are not alone and are not part of a `DATA`, `REM` or `QUOTE` they will generate a warning but will not be deleted. `endif`s that are part of any of the previous commands but are alone on a line due to a line break will be deleted.
  Numbers at the start of a line will be removed, generate a warning. Numbers at the beginning of a line after an underscore break `_` will be preserved but numbers at the beginning of a line after a colon `:` break will be removed even if it is part of a `REM` (there is no need to break a `REM` line with an `:` anyway)
  All of the warning situations above are highlighted with the MSX Syntax Highlight.

```
if a$ = "C" then _
       for f = 1 to 10:
               locate 1,1:print f:
       next
       :locate 1,3:print "done"
       :end
endif
```

```
msxbadig.py test.bad
```

```
10 IF A$="C" THEN FOR F=1 TO 10:LOCATE 1,1:PR
```

- The Dignified code can use **exclusive comments** `##` that are stripped during the conversion.
  Regular `REM` s are kept. A tenacious bug (called My Own Regex Incompetency) prevents the `##` from being removed if there are any `"` after it. (this is so `##` are not removed inside quotes.)
  Block comments can also be used, they can be `''` or `###` , the text inside the first one will be kept while the text inside the later will be removed. They will toggle the block if they are alone on a line, open it if at the start or close when at the end of a line. Regular `##` lines inside a `''` block will be removed.

```
## this will not be converted
rem this will
' this also will
###
This will go
###
''
And this stays
''
```

```
msxbadig.py test.bad
```

```
10 REM this will
20 ' this also will
30 ' And this stay
```

-

Parts of the code can be tagged with **line toggles** to be removed on demand when converted. They have the format `#name` where `name` can be any combination of letters numbers and `_` and they can be kept by simply adding a `keep #name1 #name2 ...` on a line before them. `keep` can take none, one or more toggles on the same line, separated by spaces. Toggles are used at the start of a line (`#1 print "Hello"`) and also can be alone on a line to denote the start and end of a whole section to be removed, just like block comments. They can be useful to debug different code snippets without having to comment and uncomment them every time. Block toggles can be nested but cannot be interleaved. If they are not closed a warning will be given.

```
keep #2
#1 print "this will not be converted"
#2 print "this will"
print "this also will"
#3
print "This will go"
print "And so will this"
#3
```

```
msxbadig.py test.bad
```

```
10 PRINT "this will"
20 PRINT "this also will"
```

- The default text encoding of the `.bad` and `.asc` programs (`Western (Windows 1252)`) does not support the **special MSX ASCII characters**. They can be used by opening an ASCII file exported from the MSX with them but this is far from practical, ugly and takes a lot of space (the ☻ character is represented by `<0x01>A` for instance). By using `UTF-8` encoding on the `.bad` file, **unicode** characters similar to the MSX ones can be used on the Dignified code and be translated to their analogue counterparts when converting. This allows for a much cleaner and accurate presentation.
  The supported unicode characters are:

```
 Çüéâäàåçêëèïîì
£¥₧ƒáíóúñÑ
```

They were choosen for their similarities with the MSX ones but the actual shape will depend on the font used.

> For extreme cases, MSX Sublime Tools has a TrueType MSX screen 0 font available. A Screen 1 font is on its way.

```
print "┌──────┐"
print "│SAVING│"
print "└──────┘"
```

```
msxbadig.py test.bad -tr
```

```
10 PRINT "<0x01>X<0x01>W<0x01>W<0x01>W<0x01>W
20 PRINT "<0x01>VSAVING<0x01>V"
30 PRINT "<0x01>Z<0x01>W<0x01>W<0x01>W<0x01>W
```

- An external piece of Dignified code can be inserted anywhere using the **include** command.
  `include "code.bad"` will insert the contents of `code.bad` exactly where the `include` was and can even have its lines joined with the main code by using `:` or `_` on any of the files.

```
print "This is the main file."
'
include "help.bad"
'
print "This is the main file again."
```

```
msxbadig.py test.bad
```

```
10 PRINT "This is the main file."
20 '
30 PRINT "This is a helper code."
40 PRINT "Saved on another file."
```

```
50 '
60 PRINT "This is the main file again."
```

- **True** and **false** statements can be used with numeric variables, they will be converted to `-1` and `0` respectively and their variables can be treated as true booleans on `if`s and with `not` operators.

```
~variable = true
~condition = false
if condition then variable = not variable
```

```
msxbadig.py test.bad
```

```
10 ZZ=-1
20 ZY=0
30 IF ZY THEN ZZ=NOT ZZ
```

- **Shorthand and compound** arithmetic operators (`++` , `--` , `+=` , `-=` , `*=` , `/=` , `^=`) can be used and will be converted to normal MSX Basic operations. If the **unpack operator** (`-uo`) argument is used, the conversion will try to preserve the spaces used with the operators.

```
PX++ :PY --
LO+=20 :DI -= 10
```

```
msxbadig.py test.bad -uo
```

```
10 PX=PX+1:PY = PY - 1
20 LO=LO+20:DI = DI - 10
```

Configurable arguments

Arguments can be passed on the code itself, on `MSXBadig.ini` or through the command line with each method having a priority higher than the one before.

Files

-

Source file
The Dignified code file to be read.
ini: `source_file = [source]` arg: `<source>`

- *Destination file*
  The traditional MSX Basic code file to be saved.
  ini: `destin_file = [destination]` args: `[destination]`

  If no name is given, the *destination* file will be the *source* with a `.bas` or `.asc` extension accordingly.

- *Tokenize Tool*
  The tokenizer to use: **MSX Basic Tokenizer** (**MBT**) or **openMSX Basic (de)Tokenizer** (**oMBdT**).
  ini: `tokenize_tool = [b,o]` arg: `-tt <b,o>`
  Default: `b`
  `b` will call **MBT** and `o` will call **oMBdT**.

- *Output Format (tonekized format is only available if **MBT** or **oMBdT** are present)*
  The format of the converted output.
  ini: `output_format = [t,a,b]` arg: `-of <t,a,b>`
  Default: `b`
  Both formats are exported by default (`b`). A single format can be forced by using `a` (ASCII) or `t` (tokenized).

  > The ASCII format is always exported and will only be deleted if the tokenized is successfully saved.

- *Export List (only available if **MBT** is present, does not work with **oMBdT**)*
  Saves an `.mlt` list file similar to the ones exported by assemblers with the tokens alongside the ASCII code and some statistics.
  ini: `export_list = [0-32]` arg: `-el [0-32]`
  Default: `16`

  The `[0-32]` argument refer to the number of bytes shown on the list after the line number.
  The max value is `32` . If no number is given the default of `16` will be used. If `0` is given the list will not be exported.

- ***MSX Basic Tokenizer** Path*
  The path to the **MBT** installation.
  ini: `batoken_filepath = []`

- ***openMSX Basic (de)Tokenizer** Path*
  The path to the **oMBdT** installation.
  ini: `openbatoken_filepath = []`

Numbering

- *Starting line number*
  The number of the first line.
  ini: `line_start = [#]` arg: `-ls [#]` Default: `10`

- *Line step value*
  The line number increment amount.
  ini: `line_step = [#]` arg: `-lp [#]` Default: `10`

- *Add leading zeros*
  Line numbers can be padded with zeroes.
  ini: `leading_zeros = [true,false]` arg: `-lz`
  Default: `false`

```
{do_stuff}
        print "Say something"
        do$ = "Do something"
goto {do_stuff}
```

```
msxbadig.py test.bad -ls 5 -lp 5
```

```
5 ' {do_stuff}
10 PRINT "Say something"
15 DO$="Do something"
20 GOTO 5
```

```
msxbadig.py test.bad -ls 1 -lp 50 -lz
```

```
001 ' {do_stuff}
051 PRINT "Say something"
101 DO$="Do something"
151 GOTO 1
```

Labels
Labels are removed from the code as default but there are some options to left them for debugging or readability purposes.

- *Label conversions*
  Line labels can be left on the code on a `REM` line (`0`) or their names can be removed, leaving only a blank `REM` (and any comments they had) on its place (`1`). The default (`2`) removes the line altogether. If the lines are kept, branching labels points to them. If they are removed, the code flow is directed to the line mediately after where the label was.
  ini: `handle_label_lines = [0,1,2]` arg: `-ll <0,1,2>` Default: `2`

- *Show branching labels*
  Add a `:rem` at the end of lines with branching instructions showing the name of the labels used, making it easier to visualize the flow on the converted code.
  ini: `show_branches_labels = [true,false]` arg: `-sl` Default: `false`

```
{print_result}
        print "Result"
if r$ = "Result" goto {print_result} else goto {@}
```

```
msxbadig.py test.bad -ll 2
```

```
10 PRINT "Result"
20 IF R$="Result" GOTO 10 ELSE GOTO 20
```

```
msxbadig.py test.bad -ll 1
```

```
10 '
20 PRINT "Result"
30 IF R$="Result" GOTO 10 ELSE GOTO 30
```

```
msxbadig.py test.bad -ll 0 -sl
```

```
10 ' {print_result}
20 PRINT "Result"
30 IF R$="Result" GOTO 10 ELSE GOTO 30:' {print_re
```

Blank lines

Blank lines are stripped from the source by default but they can also be left on the converted code.
Blank lines after Dignified commands (define, declare,...) are always removed.
Extra lines can also be added close to labels for clarity and organization.

- *Keep blank lines*
  Do not erase blank lines on the converted code, convert them to `rem` s.
  ini: `keep_blank_lines = [true,false]` arg: `-bl`
  Default: `false`

- *Label gap*
  Add a blank line **before** or **after** a label. (has no effect if the line labels were removed)
  `b` Before, `a` after, `ba` before and after.
  ini: `label_gap = [b,a,ba]` arg: `-lg <b,a,ba>`
  Default: `none`

```
{print_result}
        PRINT "Result"

        GOTO {print_result}
```

```
msxbadig.py test.bad -bl -lg a
```

```
10 ' {print_result}
20 '
30 PRINT "Result"
40 '
50 GOTO 10
```

Spacing

Spacing and indentation are automatically removed from the converted code.
This behaviour can be configured, however, as follows.

- *Spaces around :*
  Add an space **before** or **after** the instruction separating character `:`. Has no effect if keeping original spacing.
  `b` Before, `a` after, `ba` before and after.
  ini: `colon_spaces = [b,a,ba]` arg: `-cs <b,a,ba>`
  Default: `none`

- *Amount of general spacing*
  The conversion automatically strips all blanks (**spaces** and **TAB**s) from the code (default `0`). This can be changed to 1 space throughout (`1`) or the original spacing used can be kept using `k`:
  ini: `general_spaces = [0,1,k]` arg: `-gs <0,1,k>`
  Default: `0`

- *Unpack operators*
  All spaces surrounding mathematical operators and punctuation (`+-=<>*/^\.,;`) are stripped by default. They can be kept as general spaces using:
  ini: `unpack_operators = [true,false]` arg: `-uo`
  Default: `false`

```
for f = 10  to 7  :read a:print a  : next
for f = 0   to 9  :read a:print a  : next
for f = 100 to 300:read a:print a+1: next
```

```
msxbadig.py test.bad -gs 0 -cs b
```

```
10 FORF=10TO7 :READA :PRINTA :NEXT
20 FORF=0TO9 :READA :PRINTA :NEXT
30 FORF=100TO300 :READA :PRINTA+1 :NEXT
```

Indentation
MBD, by default, also strips all indentation used on the Dignified code. This can be controlled with.

- *Keep indentation*
  Maintain the original indentations. **TAB**s are converted to a specified number of **spaces** (2 by default).
  ini: `keep_indent = [#]` arg: `-ki [#]` Default: `2`

```
for f = 1 to 200
        print "Beware"
         if f = 100 then print "Middle point reach
next
```

```
msxbadig.py test.bad -ki 3
```

```
10 FOR F=1 TO 200
20    PRINT "Beware"
30     IF F=100 THEN PRINT "Middle point reached!"
40 NEXT
```

Comments

The conversion uses `REM`s on certain occasions to keep some original formatting; they can be redefined as such:

- *New `REM` format*
  Sometimes `REM`s are created by **MBD**; they are applied as `'` (option `s`) but they can be told to use the regular form (option `rem`):
  ini: `new_rem_format = [s,rem]` arg: `-nr <s,rem>`
  Default: `s`

- *Convert all `REM`s*
  All **pre existing** `REM`s (not added by **MBD**) can be changed to maintain coherence along the converted code. The conversion will conform to the condition above.
  ini: `convert_rem_formats = [true,false]` arg: `-cr`
  Default: `false`

```
{start_tutorial}
        print "What tutorial?"
        ' Nevermid
```

```
msxbadig.py test.bad -nr rem -cr
```

```
10 REM {start_tutorial}
20 PRINT "What tutorial?"
30 REM Nevermid
```

General conversions

- *Capitalise*
  By default all text is **capitalised** with the exception of texts inside `""` , `{labels}` , `REM` s and `DATA` s.
  The original case can be maintained by using:
  ini: `capitalize_all = [true,false]` arg: `-nc`
  Default: `true`

- *Convert `?` to `PRINT`*
  `?` as `PRINT` are left alone on the conversion, they can be told to become `PRINT` with:
  ini: `convert_interr_to_print = [true,false]` arg: `-cp` Default: `false`

- *Strip adjacent `THEN` / `ELSE` or `GOTO` s*
  MSX Basic doesn't need both `THEN` or `ELSE` and `GOTO` if they are adjacent. The converted code can be told to strip `THEN` (option `t`), `GOTO` (option `g` , default) or they can all be left alone (option `k`)
  ini: `strip_then_goto = [t,g,k]` arg: `-tg <t,g,k>`
  Default: `g`

```
{at_last}
        ? "What?"
        if me$ = "Huh?" then goto {at_last} else g
        ?:? "Not here."
{i_agree}
        ? "That is (almost) all folks."
```

```
msxbadig.py test.bad -nc -cp -tg t
```

```
10 ' {at_last}
20 print "What?"
30 if me$="Huh?" then 10 else 50
40 print:print "Not here."
50 ' {i_agree}
60 print "That is (almost) all folks."
```

Misc

-

Long variable summary
A summary of all the long name variables used on the code along with their associated classic short names can be generated on `REM` lines at the end of the converted code and the amount shown per line can be set. If used without number argument the amount shown per line is `5`.
ini: `long_var_summary = [#]` arg: `-vs [#]` Default: `5`

- *Verbose*
  Set the level of feedback given by the program.
  `0` show nothing, `1` errors, `2` errors and warnings, `3` errors, warnings and steps and `4` errors, warnings, steps and details.
  ini: `verbose_level = [#]` arg: `-vb <#>` Default: `3`

- *Supress info comments*
  Do not add information about **MSX Basic Dignified** at the top of the converted code.
  ini: `rem_header = [true,false]` arg: `-rh` Default: `false`

- *Running from build system*
  On build systems, like the one in Sublime, errrors and warning messages demand the file name to be on the offending log line. To declutter the log on a run from a console window this information is omitted unless told so.
  arg: `-frb` Default: `false`

- *Monitor Execution (only works with `-frb`)*
  If running from the build system on **MSX Sublime Tools** this will trigger the monitoring of the code execution on **openMSX**.
  ini: `monitor_execution = [true,false]` arg: `-exe` Default: `false`

- *Use the `.ini` file*
  Tells if the `.ini` file settings should be used or not, allowing it to be disabled without being moved or deleted.
  ini: `use_ini_file = [true,false]` Default: `true`

- *Write the `.ini` file*
  Rewites the `.ini` file in case its is missing.
  arg: `-ini` Default: `false`

- *Help*
  Help is available using:
  ```
  msxbadig.py -h
  ```

Technicalities and warnings

Yes, they are one and the same

MSX Basic Dignfied was made on a Mac OS with Python 2.7.10 and eventually ported to Python 3.8.1.
This was the first Python program I made; it has been through a lot of iterations but it is still a hot mess of ideas, experiments and alternative (bad) coding.
Use with care and caution.