A Retro-style BASIC Interpreter

Table of Contents

Refresh

Introduction

This is a simple interactive BASIC interpreter written in Python 3.

The interpreter is a homage to the home computers of the late 1970s and early 1980s, and when started, presents the user with an interactive prompt (`>`) typical of computers of the time. Commands to run, list, save and load BASIC programs can be entered at the prompt as well as new program statements.

No specific BASIC dialect has been implemented. It is a generalised version typical of those of the time mentioned and avoids machine specific instructions, such as those concerned with sound and graphics.

There is reasonable level of error checking. Syntax errors will be picked up and reported by the scanner (lexical analyser) as statements are entered. Runtime errors will highlight the cause and the line number of the offending statement.

The interpreter environment can be invoked as follows:

```
$ python basicui.py
```

Operators

A limited number of arithmetic expressions are provided. Addition and subtraction have the lowest precedence, but this can be changed with parentheses.

- **+** - Addition
- **-** - Subtraction
- ***** - Multiplication
- **/** - Division
- **MOD** (or **%**) - Modulo

```
> 10 PRINT 2 * 3
> 20 PRINT 20 / 10
> 30 PRINT 10 + 10
> 40 PRINT 10 - 10
> 50 PRINT 15 MOD 10
> RUN
6
2.0
20
0
```

```
5
>
```

Additional numerical operations may be performed using numeric functions (see below).

Note also that `+` also performs string concatenation, and `*` as a string replicator.

Commands

Programs may be listed using the **LIST** command:

```
> LIST
10 LET I = 10
20 PRINT I
>
```

The list command can take arguments to refine the line selection listed

`LIST 50` Lists only line 50.

`LIST 50-100` Lists lines 50 to 100, inclusive.

`LIST 50 100` Also Lists lines 50 through 100 inclusive, almost any delimiter should work here.

`LIST -100` Lists from the start of the program through line 100 inclusive.

`LIST 50-` Lists from line 50 to the end of the program.

A program is executed using the **RUN** command:

```
> RUN
10
>
```

A program may be saved to disk using the **SAVE** command. Note that the full path must be specified within double quotes:

```
> SAVE "C:\path\to\my\file"
Program written to file
>
```

The program may be re-loaded from disk using the **LOAD** command, again specifying the full path using double quotes:

```
> LOAD "C:\path\to\my\file"
Program read from file
>
```

When loading or saving, the .bas extension is assumed if not provided. If you are loading a simple name (alpha / numbers only) and in the working dir, quotes can be omitted:

```
> LOAD myprogram
```

will load regression.bas from the current working directory.

Individual program statements may be deleted by entering their line number only:

```
> 10 PRINT "Hello"
> 20 PRINT "Goodbye"
> LIST
10 PRINT "Hello"
20 PRINT "Goodbye"
> 10
> LIST
20 PRINT "Goodbye"
>
```

The program may be erased entirely from memory using the **NEW** command:

```
> 10 LET I = 10
> LIST
10 LET I = 10
> NEW
> LIST
>
```

The program can be renumbered using the **RENUM** command:

The default renumbering starts at line 10 and increases in steps of 10. However, arguments can be used to customise the renumbering.

`RENUM` Renumbers from 10 in steps of 10.

`RENUM 100` Renumbers from 100 in steps of 10.

`RENUM 200-5` Renumbers from 200 in steps of 5.

`RENUM 200 5` Also renumbers from 200 in steps of 5.

`RENUM 1000-` Renumbers from 1000 in steps 0f 10.

`RENUM -20` Renumbers from 10 in steps of 20.

Finally, it is possible to leave the BASIC programming environment with the **EXIT** command:

```
> EXIT
c:\
```

On occasion, it might be necessary to force termination of a program and return to the command level environment, for example, because it is caught in an infinite loop. This can be done by using *Ctrl-C* to force the program to stop:

```
> 10 PRINT "Hello"
> 20 GOTO 10
> RUN
"Hello"
"Hello"
"Hello"
...
...
<Ctrl-C>
Program terminated
> LIST
10 PRINT "Hello"
20 GOTO 10
>
```

Programming Language Constructs

Statement Structure

As per usual in old-school BASIC, all program statements must be prefixed with a line number which indicates the order in which the statements will be executed. There is a renumber command that allows all line numbers to be modified. A statement may be modified or replaced by re-entering a statement with the same line number:

```
> 10 LET I = 10
> LIST
10 LET I = 10
> 10 LET I = 200
> LIST
10 LET I = 200
>
```

Multiple statements may appear on one line separated by a colon:

```
> 10 LET X = 10: PRINT X
```

NOTE: Currently inline loops are NOT supported

```
10 FOR I = 1 to 10: PRINT I: NEXT
```

will need to be decomposed to individual lines.

Variables

Variable types follow the typical BASIC convention. *Simple* variables may contain either strings or numbers (the latter may be integers or floating point numbers). Likewise *array* variables may contain arrays of either strings or numbers, but they cannot be mixed in the same array.

Note that all keywords and variable names are case insensitive (and will be converted to upper case internally by the scanner). String literals will retain their case however. There is no inherent limit on the length of variable names or string literals, this will be dictated by the limitations of Python. The range of numeric values is also dependent upon the underlying Python implementation.

Note that variable names may only consist of alphanumeric characters and underscores. However, they must all begin with an alphabetic character. For example:

- `MY_VAR`
- `MY_VAR6$`
- `VAR77(0, 0)`

are all valid variable names, whereas:

- `5_VAR`
- `66VAR$`
- `66$`

are all invalid.

Numeric variables have no suffix, whereas string variables are always suffixed by `$`. Note that `I` and `I$` are considered to be separate variables. Note that string literals must always be enclosed within double quotes (not single quotes). Using no quotes will result in a syntax error.

Array variables are defined using the **DIM** statement, which explicitly lists how many dimensions the array has, and the sizes of those dimensions:

```
> REM DEFINE A THREE DIMENSIONAL NUMERIC ARRAY
> 10 DIM A(3, 3, 3)
```

Note that the index of each dimension always starts at *zero*, but for compatibility with some basic dialects the bounds of each dimension are expanded by one to enable element access including the length given by the `len` function. So in the above example, valid index values for array *A* will be *0*, *1*, *2* or *3* for each dimension. Arrays may have a maximum of three dimensions.

Numeric arrays will be initialised with each element set to zero, while string arrays will be initialised with each element set to the empty string `""`.

As for simple variables, a string array has its name suffixed by a `$` character, while numeric arrays do not have a suffix. An attempt to assign a string value to a numeric array or vice versa will generate an error.

Array variables with the same name but different *dimensionality* are treated as the same. For example, using a **DIM** statement to define *I(5)* and then a second **DIM** statement to define *I(5, 5)* will result in the second definition (the two dimensional array) overwriting the first definition (the one dimensional array).

Array values may be used within any expression, such as in a **PRINT** statement for string values, or in any numerical expression for number values. However, you must be specific about which array element you are referencing, using the correct number of in-range indexes. If that particular array value has not yet been assigned, then an error message will be printed.

```
> 10 DIM MYARRAY(2, 2, 2)
> 20 LET MYARRAY(0, 1, 0) = 56
> 30 PRINT MYARRAY(0, 1, 0)
> RUN
56
> 30 PRINT MYARRAY(0, 0, 0)
> RUN
Empty array value returned in line 30
>
```

As in all implementations of BASIC, there is no garbage collection. This is not unreasonable since all variables have global scope.

Program Constants

Constants may be defined through the use of the **DATA** statement. They may consist of numeric or string values and are declared in a comma separated list:

```
> 10 DATA 56, "Hello", 78
```

These values can then later be assigned to variables using the **READ** statement. Note that the type of the value (string or numeric) must match the type of the variable, otherwise an error message will be triggered. Therefore, attention should be paid to the relative ordering of constants and variables. Once the constants on a **DATA** statement have been used by a **READ** statement, the next **READ** statement will move to the **DATA** statement with the next higher line number, if there are no more **DATA** statements before the end of the program an error message will be displayed. This is to ensure that the program is not left in a state where a variable has not been assigned a value, but nevertheless an attempt to use that variable is made later on in the program.

Normally each **DATA** statement is consumed sequently by **READ** statements however, the **RESTORE** statement can be used to override this order and set the line number of the **DATA** statement that will be used by the next **READ** statement. If the *line-number* used in a **RESTORE** statement does not refer to a **DATA** statement an error will be displayed.

The constants defined in the **DATA** statement may be consumed using several **READ** statements or several **DATA** statements may be consumed by a single **READ** statement.:

```
> 10 DATA 56, "Hello", 78
> 20 READ FIRSTNUM, S$
> 30 PRINT FIRSTNUM, " ", S$
> 40 READ SECONDNUM
> 50 PRINT SECONDNUM
> 60 DATA "Another "
> 70 DATA "Line "
> 80 DATA "of "
> 90 DATA "Data"
> 100 RESTORE 10
> 110 READ FIRSTNUM, S$, SECONDNUM, A$, B$, C$, D$
> 120 PRINT S$," ",A$,B$,C$,D$
> RUN
56 Hello
78
Hello Another Line of Data
>
```

It is a limitation of this BASIC dialect that it is not possible to assign constants directly to array variables within a **READ** statement, only to simple variables.

Comments

The **REM** statement is used to indicate a comment, and occupies an entire statement line. It has no effect on execution:

```
> 10 REM THIS IS A COMMENT
```

Stopping a Program

The **STOP** statement may be used to cease program execution. The command **END** has the same effect.

```
> 10 PRINT "one"
> 20 STOP
> 30 PRINT "two"
> RUN
one
>
```

A program will automatically cease execution when it reaches the final statement, so a **STOP** may not be necessary. However, a **STOP** *will* be required if subroutines have been defined at the end of the program, otherwise execution will continue through to those subroutines without a corresponding subroutine call. This will cause an error when the **RETURN** statement is processed and the interpreter attempts to return control back to the caller.

Assignment

Assignment may be made to numeric simple variables (which can contain either integers or floating point numbers) and string simple variables (string variables are distinguished by their dollar suffix). The interpreter will enforce this division between the two types:

```
> 10 LET I = 10
> 20 LET I$ = "Hello"
```

The **LET** keyword is also optional:

```
> 10 I = 10
```

Array variables may also have values assigned to them. The indices can be derived from numeric expressions:

```
> 10 DIM NUMS(3, 3)
> 20 DIM STRS$(3, 3)
> 30 LET INDEX = 0
> 40 LET NUMS(INDEX, INDEX) = 55
> 50 LET STRS$(INDEX, INDEX) = "hello"
```

Attempts to assign the wrong type (number or string) to a numeric or string array, attempts to assign a value to an array by specifying the wrong number of dimensions, and attempts to assign to an array using an out of range index, will all result in an error.

Printing to Standard Output

The **PRINT** statement is used to print to the screen (or to a file, see File I/O below):

```
> 10 PRINT 2 * 4
> RUN
8
> 10 PRINT "Hello"
> RUN
Hello
>
```

Multiple items may be printed by providing a semicolon separated list. The items in the list will be printed immediately after each another, so spaces must be inserted if these are required:

```
> 10 PRINT 345; " hello "; 678
> RUN
345 hello 678
>
```

A blank line may be printed by using the **PRINT** statement without arguments:

```
> 10 PRINT "Here is a blank line:"
> 20 PRINT
> 30 PRINT "There it was"
> RUN
Here is a blank line:

There it was
>
```

A print statement terminated by a semicolon will not include a CR/LF.

Unconditional Branching

Like it or loath it, the **GOTO** statement is an integral part of BASIC, and is used to transfer control to the statement with the specified line number:

```
> 10 PRINT "Hello"
> 20 GOTO 10
> RUN
Hello
Hello
Hello
...
```

Subroutine Calls

The **GOSUB** statement is used to generate a subroutine call. Control is passed back to the program at the next statement after the call by a **RETURN** statement at the end of the subroutine:

```
> 10 GOSUB 100
> 20 PRINT "This happens after the subroutine"
> 30 STOP
> 100 REM HERE IS THE SUBROUTINE
> 110 PRINT "This happens in the subroutine"
> 120 RETURN
> RUN
This happens in the subroutine
This happens after the subroutine
>
```

Note that without use of the **STOP** statement, execution will run past the last statement of the main program (line 30) and will re-execute the subroutine again (at line 100).

Subroutines may be nested, that is, a subroutine call may be made within another subroutine.

A subroutine may also be called using the **ON-GOSUB** statement (see Conditional Branching, below).

Loops

Bounded loops are achieved through the use of **FOR - NEXT** statements. The loop is controlled by a numeric loop variable that is incremented or decremented from a start value to an end value. The loop terminates when the loop variable reaches the end value. The loop variable must also be specified in the **NEXT** statement at the end of the loop.

```
> 10 FOR I = 1 TO 3
> 20 PRINT "hello"
> 30 NEXT I
> RUN
hello
hello
hello
>
```

Loops may be nested within one another.

The **STEP** statement allows the loop variable to be incremented or decremented by a specified amount. For example, to count down from 5 in steps of -1:

```
> 10 FOR I = 5 TO 1 STEP -1
> 20 PRINT I
> 30 NEXT I
> RUN
5
4
3
2
1
>
```

Note that the start value, end value and step value need not be integers, but can also be floating point numbers. If the loop variable was previously assigned in the program, its value will be replaced by the start value.

After the completion of the loop, the loop variable value will be the *end value* + *step value* (unless

Conditionals

Conditionals are implemented using the **IF - THEN - ELSE** statement. The expression is evaluated and the appropriate statements executed depending upon the result of the evaluation. If a positive integer is supplied as the **THEN** or the **ELSE** statement, a jump will be performed to the indicated line number.

Note that the **ELSE** clause is optional and may be omitted. In this case, the **THEN** branch is taken if the expression evaluates to true, otherwise the next statement is executed.

Conditional branching example:

```
> 10 REM PRINT THE GREATEST NUMBER
> 20 LET I = 10
> 30 LET J = 20
> 40 IF I > J THEN 50 ELSE 70
> 50 PRINT I
> 60 GOTO 80
> 70 PRINT J
> 80 REM FINISHED
> RUN
20
>
```

The following code segment is equivalent to the segment above:

```
> 10 REM PRINT THE GREATEST NUMBER
> 20 LET I = 10
> 30 LET J = 20
> 40 IF I > J THEN PRINT I ELSE PRINT J
> 80 REM FINISHED
> RUN
20
>
```

A **THEN** or **ELSE** can be supplied multiple statements if they are separated by a colon `:`.

```
> 10 REM PRINT THE GREATEST NUMBER
> 20 LET I = 10
> 30 LET J = 20
> 40 IF I > J THEN LET L = I : PRINT I ELSE LET L = J : PRINT J
> 50 PRINT L
> 80 REM FINISHED
> RUN
20
20
>
```

Note that should an **IF - THEN - ELSE** stmt be used in a **THEN** code block or multiple **IF - THEN - ELSE** statements are used in either a single **THEN** or **ELSE** code block, the block grouping is ambiguous and logical processing may not function as expected. There is no ambiguity when single **IF - THEN - ELSE** statements are placed within **ELSE** blocks.

Ambiguous:

```
> 100 IF I > J THEN IF J >= 100 THEN PRINT "I > 100" ELSE
                                  PRINT "Not clear which **IF** this belongs to"
```

Not ambiguous:

```
> 100 IF I < J THEN PRINT "I is less than J" ELSE IF J > 100 THEN
                                  PRINT "I > 100" ELSE PRINT "J <= 100"
```

Allowable relational operators are:

- '=' (equal, note that in BASIC the same operator is used for assignment)
- '<' (less than)
- '>' (greater than)
- '<=' (less than or equal)
- '>=' (greater than or equal)

- '<>' / '!=' (not equal)

The logical operators **AND** and **OR** are also provided to allow you to join two or more expressions. The **NOT** operator can also be given before an expression.

`=` and `<>` can also be considered logical operators. However, unlike **AND** or **OR** they can't be used to join more than two expressions.

Inputs		***AND***	***OR***	***=***	****<>* / !=***
FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

Input	**NOT**
TRUE	FALSE
FALSE	TRUE

Example:

```
> 10 A = 10
> 20 B = 20
> 30 IF NOT A > B AND B = 20 OR A >= 5 THEN 60
> 40 PRINT "Test failed!"
> 50 STOP
> 60 PRINT "Test passed!"
> RUN
Test passed!
```

Expressions can be inside brackets to change the order of evaluation. Compare the output when line 30 is changed:

```
> 30 IF NOT A > B AND (B = 20 OR A >= 5) THEN 60
> RUN
Test failed!
```

ON GOTO, ON GOSUB

The **ON** *expr* **GOTO | GOSUB** *line1, line2,...* statement will call a subroutine or jump to a line number in the list of line numbers corresponding to the ordinal value of the evaluated *expr*. The first line number corresponds with an *expr* value of 1. *expr* must evaluate to an integer value. If *expr* evaluates to less than 1 or greater than the number of provided line numbers execution continues on the next statement without making a subroutine call or jump:

```
> 20 LET J = 2
> 30 ON J GOSUB 100,200,300
> 40 STOP
> 100 REM THE 1ST SUBROUTINE
> 110 PRINT "J is ONE"
> 120 RETURN
> 200 REM THE 2ND SUBROUTINE
> 210 PRINT "J is TWO"
> 220 RETURN
> 300 REM THE 3RD SUBROUTINE
> 310 PRINT "J is THREE"
> 320 RETURN
> RUN
J is TWO
>
```

It is also possible to call a subroutine depending upon the result of a conditional expression using the **IFF** function (see Ternary Functions, below). In the example below, if the expression evaluates to true, **IFF** returns `1` and the subroutine is called, otherwise **IFF** returns `0` and execution continues to the next statement without making the call:

```
> 10 LET I = 10
> 20 LET J = 5
> 30 ON IFF (I > J, 1, 0) GOSUB 100
> 40 STOP
> 100 REM THE SUBROUTINE
> 110 PRINT "I is greater than J"
> 120 RETURN
> RUN
I is greater than J
>
```

Ternary Functions

As an alternative to branching, Ternary functions are provided.

- **IFF** `(x, y, z)` - Evaluates `x` and returns `y` if true, otherwise returns `z`. *(Note: `y` and `z` are expected to be numeric.)*
- **IF$** `(x, y$, z$)` - As above, but `y$` and `z$` are expected to be strings.

```
> 10 LET I = 10
> 20 LET J = 5
> 30 PRINT IF$(I > J, "I is greater than J", "I is not greater than J")
> 40 K = IFF(I > J, 20, 30)
> 50 PRINT K
> RUN
```

```
I is greater than J
20
```

User Input

The **INPUT** statement is used to solicit input from the user (or read input from a file, see File I/O below):

```
> 10 INPUT A
> 20 PRINT A
> RUN
? 22
22
>
```

The default input prompt of '`?`' may be changed by inserting a prompt string, which must be terminated by a semicolon, thus:

```
> 10 INPUT "Input a number - "; A
> 20 PRINT A
> RUN
Input a number - 22
22
>
```

Multiple items may be input by supplying a comma separated list. Input variables will be assigned to as many input values as supplied at run time. If there are more input values supplied than input variables, excess commas will be left in place. Conversely, if not enough input values are supplied, an error message will be printed and the user will be asked to re-input the values again.

Furthermore, numeric input values must be valid numbers (integers or floating point).

```
> 10 INPUT "Num, Str, Num: ": A, B$, C
> 20 PRINT A, B$, C
> RUN
Num, Str, Num: 22, hello!, 33
22 hello!33
>
```

A mismatch between the input value and input variable type will trigger an error, and the user will be asked to input the values again.

It is a limitation of this BASIC dialect that it is not possible to assign constants directly to array variables within an **INPUT** statement, only simple variables.

File Input / Output

Data can be read from or written to files using the **OPEN**, **FSEEK**, **INPUT**, **PRINT** and **CLOSE** statements.

When a file is opened using the syntax **OPEN** "*filename*" **FOR INPUT | OUTPUT | APPEND AS** *#filenum* [**ELSE** *linenum*] a file number (*#filenum*) is assigned to the file, which if specified as the first argument of an **INPUT** or **PRINT** statement, will direct the input or output to the file.

If there is an error opening a file and the optional **ELSE** option has been specified, program control will branch to the specified line number, if the **ELSE** has not been provided an error message will be displayed.

If a file is opened for **OUTPUT** which does not exist, the file will be created, if the file does exist, its contents will be erased and any new **PRINT** output will replace it. If a file is opened for **APPEND** an error will occur if the file doesn't exist (or the **ELSE** branch will occur if specified). If the file does exist, any **PRINT** statements will add to the end of the file.

If an input prompt is specified on an **INPUT** statement being used for file I/O (i.e. *#filenum* is specified) an error will be displayed.

The **FSEEK** *#filenum*, *filepos* statement will position the file pointer for the next **INPUT** statement.

The **CLOSE** *#filenum* statement will close the file.

```
> 10 OPEN "FILE.TXT" FOR OUTPUT AS #1
> 20 PRINT #1,"0123456789Hello World!"
> 30 CLOSE #1
> 40 OPEN "FILE.TXT" FOR INPUT AS #2
> 50 FSEEK #2,10
> 60 INPUT #2,A$
> 70 PRINT A$
> RUN
Hello World!
>
```

Numeric Functions

Several numeric functions are provided, and may be used with any numeric expression. For example, the square root function, **SQR**, can be applied expressions consisting of both literals and variables:

```
> 10 LET I = 6
> 20 PRINT SQR(I - 2)
> RUN
2.0
>
```

Available numeric functions are:

- **ABS** `(x)` - Calculates the absolute value of `x`
- **ATN** `(x)` - Calculates the arctangent of `x`
- **COS** `(x)` - Calculates the cosine of `x`, where `x` is an angle in radians
- **EXP** `(x)` - Calculates the exponential of `x` (i.e. e^x), where *e* = 2.718281828
- **INT** `(x)` - Rounds numbers down to the lowest whole integer less than or equal to `x`
- **LOG** `(x)` - Calculates the natural logarithm of `x`
- **MAX** `(x, y[, z]...)` - Returns the greatest value from a list of expressions
- **MIN** `(x, y[, z]...)` - Returns the least value from a list of expressions

```
> 10 PRINT MAX(-2, 0, 1.5, 4)
> 20 PRINT MIN(-2, 0, 1.5, 4)
> RUN
> 4
> -2
```

- **PI** - Returns the value of π (i.e. 3.1415926...)
- **POW** `(x, y)` - Calculates x^y
- **RND** `(mode)` - Psuedorandom number generator. The behavior is different depending on the value passed. If the value is positive, the result will be a new random value between 0 and 1 (including 0, but not 1). If the value is negative, it will be rounded down to the nearest integer and used to reseed the random number generator.

Pseudorandom sequences can be repeated by reseeding with the same number. Generates a pseudo random number N, where *0 <= N < 1*. Can be reset using the **RANDOMIZE** instruction with an optional seed value: e.g.

```
> 10 RANDOMIZE 100
> 20 PRINT RND(1)
> RUN
0.1456692551041303
>
```

Random integers can be generated by combining **RND** and **INT**: e.g.

```
> 10 PRINT INT(RND(1) * 6) + 1
> RUN
3
> RUN
6
>
```

Seeds may not produce the same result on another platform.

- **RNDINT** `(lo, hi)` - Generates a pseudo random integer N, where *lo <= N <= hi*. Uses the same seed as above.
- **ROUND** `(x)` - Rounds number to the nearest integer
- **SIN** `(x)`- Calculates the sine of `x`, where `x` is an angle in radians
- **SQR** `(x)` - Calculates the square root of `x`
- **TAN** `(x)` - Calculates the tangent of `x`, where `x` is an angle in radians

String Functions

Some functions are provided to help you manipulate strings. Functions that return a string have a *'$'* suffix like string variables.

NOTE: For compatibility with older basic dialetcs, all string indexes are 1 based.

The functions are:

- **ASC** `(x$)` - Returns the character code for `x$`. `x$` is expected to be a single character. Note that despite the name, this function can return codes outside the ASCII range.
- **CHR$** `(x)` - Returns the character specified by character code `x`.
- **INSTR** `(x$, y$[, start[, end]])` - Returns position of `y$` inside `x$`, optionally start searching at position `start` and end at `end`. Returns `0` if no match found.
- **LEN** `(x$)` - Returns the length of `x$`.
- **LOWER$** `(x$)` - Returns a lower-case version of `x$`.
- **MID$** `(x$, y[, z])` - Returns part of `x$` starting at position `y`. If `z` is provided, that number of characters is returned, if omitted the entire remaider of the string is returned.
- **LEFT$** `(x$, y)` - Returns the left-most `y` characters from string `x$`. If `y` * exceeds the length of `x$`, the entire string will be returned.
- **RIGHT$** `(x$, y)` - Returns the right-most `y` characters from string `x$`. If `y` exceeds the length of `x$`, the entire string will be returned.

- **STR$** `(x)` - Returns a string representation of numeric value `x`.
- **UPPER$** `(x$)` - Returns an upper-case version of `x$`.
- **VAL** `(x$)` - Attempts to convert `x$` to a numeric value. If `x$` is not numeric, returns `0`.
- **TAB** `(x)` - When included in a **PRINT** statement *print-list*, it specifies the position `x` on the line where the next text will be printed. If the specified position `x` is less than the current print position a newline is printed and the print location is set to the specified column. If the **TAB** function is used anywhere other than on a **PRINT** statement, it will return a string containing `x` spaces with no CR/LF.

Examples for **ASC**, **CHR$** and **STR$**

```
> 10 I = 65
> 20 J$ = CHR$(I) + " - " + STR$(I)
> 30 PRINT J$
> 40 PRINT ASC("Z")
RUN
A - 65
90
```

Strings may also be concatenated using the '+' operator:

```
> 10 PRINT "Hello" + " there"
> RUN
Hello there
```

Strings may be repeated using the '*' operator:

```
> 10 PRINT "Hello " * 5
> RUN
Hello Hello Hello Hello Hello
```

Example Programs

A number of example BASIC programs have been supplied:

- *factorial.bas* - A simple BASIC program to take a number, *N*, as input from the user and calculate the corresponding factorial *N!*.
- *test.bas* - A program to checks the features of the programming language to verify that the parser is working properly.

The following are examples of games typical of the time. The BASIC code has been altered where required to conform to this particular dialect, otherwise they are as found, principally on the internet. No guarantee is made concerning their freedom from errors.

- *oregon.bas* - A port of The Oregon Trail. This is a text based adventure game, originally developed 1971.
- *eliza.bas* - A port of the early chatbot, posing as a therapist, originally created 1964. This particular BASIC version has its origins in a version of the program developped in 1973.
- *startrek.bas* - A port of the 1971 Star Trek text based strategy game.
- *adventure.bas* - A port of the text based adventure game. The game was originally created in 1975 and further expanded in 1976 and 1977 and many variations exist. This is a reduced version based on the original theme

Informal Grammar Definition

ABS(*numerical-expression*) - Calculates the absolute value of the result of *numerical-expression*.

ASC(*string-expression*) - Returns the character code of the result of *string-expression*.

ATN(*numerical-expression*) - Calculates the arctangent value of the result of *numerical-expression*.

CHR$(*numerical-expression*) - Returns the character specified by character code of the result of *numerical-expression*.

CLOSE *#filenum* - Closes an open file.

COS(*numerical-expression*) - Calculates the cosine of the result of *numerical-expression*.

DATA(*expression-list*) - Defines a list of string or numerical values.

DIM *array-variable*(*dimensions*) - Defines a new array variable.

EXIT - Exits the BASIC environment.

EXP(*numerical-expression*) - Calculates the exponential value of the result of *numerical-expression*.

FOR *loop-variable* = *start-value* **TO** *end-value* [**STEP** *increment*] - Bounded loop.

FSEEK *#filenum*, *filepos* - Positions the file input pointer to the specified location within the open file, the next **INPUT** *#filenum* will read starting at file position *filepos*.

GOSUB *line-number* - Subroutine call.

GOTO *line-number* - Unconditional branch.

IF *expression* **THEN** *line-number* | *basic-statement(s)* [**ELSE** *line-number* | *basic-statement(s)*] - Conditional.

IFF(*expression*, *numeric-expression*, *numeric-expression*) - Evaluates *expression* and returns the value of the result of the first *numeric-expression* if true, or the second if false.

IF$(*expression*, *string-expression*, *string-expression*) - Evaluates *expression* and returns the value of the result of the first *string-expression* if true, or the second if false.

INPUT [*#filenum*, | *input-prompt*;] *simple-variable-list* - Processes user or file input presented as a comma separated list.

INSTR(*main-string-expression*, *sub-string-expression*[, *start-numeric-expression*[, *end-numeric-expression*]]) - Returns position of first *sub-string-expression* inside *main-string-expression*, optionally start searching at position given by *start-numeric-expression* and optionally ending at position given by *end-numeric-expression*. Returns `-1` if no match found.

LEFT$(*string-expression*, *char-count*) - Takes the result of *string-expression* and returns the left-most *char-count* characters. If *char-count* exceeds string length the entire string is returned.

LEN(*string-expression*) - Returns the length of the result of *string-expression*.

[**LET**] *variable* = *numeric-expression* | *string-expression* - Assigns a value to a simple variable or array variable.

LIST - Lists the program.

LOAD *filename* - Loads a program from disk.

LOWER$(*string-expression*) - Returns a lower-case version of the result of *string-expression*.

LOG(*numerical-expression*) - Calculates the natural logarithm value of the result of *numerical-expression*.

NEW - Clears the program from memory.

NEXT *loop-variable* - See **FOR** statement.

MAX(*expression-list*) - Returns the greatestt value in *expression-list*.

MID$(*string-expression*, *start-position*[, *end-position*]) - Takes the result of *string-expression* and returns part of it, starting at position *start-position*, and ending at *end-position*. *end-position* can be omitted to get the rest of the string. If *start-position* or *end-position* are negative, the position is counted backwards from the end of the string.

MIN(*expression-list*) - Returns the least value in *expression-list*.

ON *expression* **GOSUB | GOTO** *line-number1, line-number2, ...* - Conditional subroutine call | branch - Program flow will be transferred either through a **GOSUB** subroutine call or a **GOTO** branch to the line number in the list of line numbers corresponding to the ordinal value of the evaluated *expr*. The first line number corresponds with an *expr* value of `1`. *expr* must evaluate to an integer value.

OPEN "*filename*" **FOR INPUT | OUTPUT | APPEND AS** *#filenum* [**ELSE** *linenum*] - Opens the specified file. Program control is transferred to *linenum* if an error occurs otherwise continues on the next line.

PI - Returns the value of π.

POW(*base*, *exponent*) - Calculates the result of raising the base to the power of the exponent.

PRINT [*#filenum*,]*print-list* - Prints a semicolon separated list of literals or variables to the screen or to a file. Included CR/LF by default, but this can be suppressed by ending the statement with a semicolon.

RANDOMIZE [*numeric-expression*] - Resets random number generator to an unpredictable sequence. With optional seed (*numeric expression*), the sequence is predictable.

READ *simple-variable-list* - Reads a set of constants into the list of variables.

REM *comment* - Internal program documentation.

RETURN - Return from a subroutine.

RESTORE *line-number* - Sets the line number that the next **READ** will start loading constants from. *line-number* must refer to a **DATA** statement.

RIGHT$(*string-expression*, *char-count*) - Takes the result of *string-expression* and returns the right-most *char-count* characters. If *char-count* exceeds string length, the entire string is returned.

RND(*mode*) - For mode values >= 0 generates a pseudo random number *N*, where *0 <= N < 1*. For values < 0 reseeds the PRNG.

RNDINT(*lo-numerical-expression*, *hi-numerical-expression*) - Generates a pseudo random integer *N*, where *lo-numerical-expression <= N <= hi-numerical-expression*

ROUND(*numerical-expression*) - Rounds *numerical-expression* to the nearest integer.

RUN - Runs the program.

SAVE *filename* - Saves a program to disk.

SIN(*numerical-expression*) - Calculates the sine of the result of *numerical-expression*.

SQR(*numerical-expression*) - Calculates the square root of the expression.

STOP - Terminates a program.

STR$(*numerical-expression*) - Returns a string representation of the result of *numerical-expression*.

TAN(*numerical-expression*) - Calculates the tangent of the result of *numerical-expression*.

UPPER$(*string-expression*) - Returns an upper-case version of the result of *string-expression*.

VAL(*string-expression*) - Attempts to convert the result of *string-expression* to a numeric value. If it is not numeric, returns `0`.

Architecture

The interpreter is implemented using the following Python modules:

- **basicui.py** - This module provides the user interface. It allows the user to input program statements and to execute the resulting program. It also allows the user to run commands, for example to save and load programs, or to list them.
- **tokens.py** - This defines the tokens that are produced by the scanner (lexical analyser). The class mostly defines token categories and provides a simple token pretty-printing method.
- **scanner.py** - This implements the lexical analyser. Lexical analysis is performed on each statement as the programm is loaded from file or entered from the user interface.
- **program.py** - This class implements an actual basic program, which is represented as a dictionary. Dictionary keys are the statement line numbers and the corresponding value is the list of tokens that make up the corresponding statement. Statements are executed by calling the parser to parse one statement at a time. This class maintains a program counter, an indication of which line number should be executed next. The program counter is incremented to the next line number in sequence, unless an executed a statement has resulted in a branch. The parser indicates this by signalling to the program object by returning a message object.
- **parser.py** - This class implements a parser for individual BASIC statements. Since the parser is based on the processing of individual statements, it uses a sends Msg object (from the message module) to indicate when program level actions are required, such as recording the return address following a subroutine jump. The parser maintains a symbol table (implemented as a dictionary) in order to record the value of variables as they are assigned. This approach is inefficient because each statement must be parsed every time it is encountered including those enclosed in a loop. However, it does mean that a program can be freely edited without the need for a separate 'compilation' process.
- **message.py** - A simple data object that allows the parser to signal a change in control flow. This could be as a result of the line just parsed including a jump (GOTO or conditional branch), a subroutine call (GOSUB), loop evaluation or program termination (STOP).

Unresolved Issues and Limitations

- Decimal values less than one must be expressed with a leading zero (i.e. `0.34` rather than `.34`)
- User input values cannot be directly assigned to array variables in an **INPUT** or **READ** statement
- Strings representing numbers (e.g. `"10"`) can actually be assigned to numeric variables in **INPUT** and **READ** statements without an error, Python will silently convert them to integers.