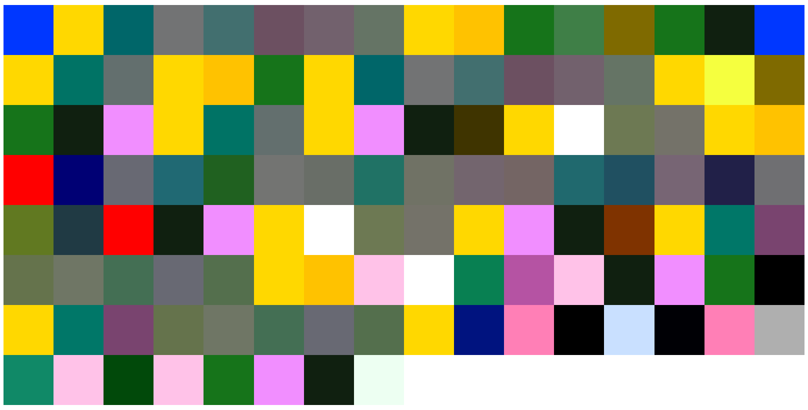
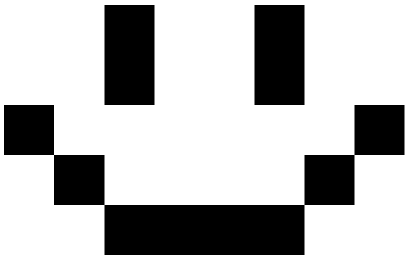
Pawet Language Specification

Version 0.1, 23/03/2023





Pawet created by Kristian Hansen
(Hansen Data Reality Ltd.)

Specification written by Kristian Hansen

Table of Contents

Introduction 3
List of Commands 4
Structure of a Pawet Program 7
Variables and Literals 8
  Variable Declaration 8
  Defining literals 9
  Variable and literal examples 10
Equations and operators 12
  Operator List 12
  Equation examples 13
Functions 15
Control Flow 18
  If-then-else statements 18
  Loops 20
Miscellaneous topics 22
  Importing other Pawet programs 22
  Printing to the Python console 22

Introduction

Welcome to Pawet. This specification is designed to lay out exactly what Pawet can do and what its features are and what its capabilities are. Please note this specification is subject to change and be updated as improvements and fixes progress on the language.

Pawet is an *interpreted* image-based programming language. The language is interpreted using a custom interpreter in *Python* (Python version 3.10). Each pixel of an image is a command, and the colour of the pixel in hexadecimal RGB (Red, Blue, Green) determines what the command is and the behaviour of the Pawet program. Pawet was inspired by the esoteric language *Piet*, which is another image-based programming language.

Pawet was a side project made by Kristian Hansen (Hansen Data Reality Ltd.) and designed to be open source and free. There is no true purpose to Pawet as it was made as a joke, but it is designed to be *Turing-complete*, fully scriptable, and purely functional (i.e. no objects or classes). The language syntax is inspired by the C-like languages and dabbles with syntax from many other languages.

The preferred image format of Pawet is Portable Network Graphics (PNG) images due to its lossless compression. Bitmaps will also be acceptable but are more file-size intensive. Pawet ignores the alpha colour channel of images so this is not needed for smaller image file sizes, and Pawet can take any n-dimensional image (i.e. width and height can be anything, as long as it fits your program!)

Image programs are read from left to right then up to down (as one would usually read a book or a worded document). This guide also assumes the reader has a basic knowledge and understanding of programming concepts and usage and is written with language that reflects that. The language is also under development and a work in progress so in some cases the expected behaviour might not be what is laid out in this specification. Please log an issue on the Github repository should you encounter unexpected behaviour.

If you are looking for the standard library documentation, please view the "Pawet Standard Library" document. This guide simply explains the syntax and behaviour of Pawet itself.

List of Commands

Below is a list of commands/abilities and their corresponding colour value. Please note this list and its corresponding colour value is subject to change as this project is developed more. A brief description is given for each instruction, which each will be covered more in detail later in this specification. This is mainly a reference point for users.

Instruction	Brief description	Hex Value	Colour
Skip/Whitespace	Blank spot with no code. Interpreter skips these	`0xffffff`	⬜
Statement End	Marks the end of a statement. Much like the semicolon (‘;’) in other languages.	`0x102010`	⬛
Bracket start	Marks the start of a bracket pair. For use in equations, function calls and definitions, and control flow conditions	`0x000000`	⬛
Bracket end	Marks the end of a bracket pair. See above.	`0xafafaf`	▨
String literal	Demarcates a string literal (equivalent of “”)	`0xff0000`	🟥
If	Marks the next brackets will be an if statement. See Control Flow for more.	`0x007f0e`	🟩
End if	Marks the end of an `if` statement body	`0x4cff00`	🟩
Then	Marks the next brackets and the following body is a ‘then’ statement (equivalent of `else if`)	`0xfaff3e`	🟨
End then	Marks the end of a ‘then’ statement	`0x1c1cff`	🟦
Else	Marks the following body as an else statement, the final fallback if any of the if or then statements’ conditions were false	`0x56ff94`	🟩
End else	Marks the end of an else statement	`0xffbe47`	🟧
True	Represents the ‘true’ literal/constant.	`0x7f6a00`	🟫
False	Represents the ‘false’ literal/constant	`0xff006e`	🟥
Function	Defines a function, See the Functions section for more	`0xb200ff`	🟪
End function	Marks the end of a function body	`0x57007f`	🟪
Variable/function name	Marks a group of pixels defining the name of a variable or function. See variables or functions for more info.	`0xffd800`	🟨
Escape	Escapes the next pixel value in a string, function/variable name, or literal in case a pixel matches the encapsulating pixels. See escaping for more info.	`0x7f0037`	🟥
Bool type	Represents the bool type. See types for more info	`0x0037ff`	🟦
Int type	Represents the int type. Can be any size	`0x7f3300`	🟫

Float type	Represents the floating-point number type	`0x404040`	⬛
String type	Represents the string type	`0x3f3400`	🟫
Void type	Represents no return type (for functions only)	`0xff7fed`	🟪
Return	Return statement (exits functions)	`0xff7f42`	🟧
Loop	Defines a loop. See control flow for more info	`0xffb584`	🟧
End loop	Marks the end of a loop body	`0x007f7f`	🟦
Comma	Acts as a separator when passing in/defining variables in functions or structs	`0x00a510`	🟩
Assignment	Assigns the right to the left, equivalent of =	`0xffc200`	🟨
Int literal	Defines that the pixels in between this will be an int literal	`0xffc2e8`	🟪
Float literal	Defines that the pixels in between this will be a float literal	`0xff7fb6`	🟪
Point	Marks a point. For use in float literals or struct property accessing (equivalent of .)	`0xc9e0ff`	🟦
Increment	Increments the variable (++)	`0x328bff`	🟦
Decrement	Decrements the variable (--)	`0x328b27`	🟩
Add	Addition operator (+)	`0x5c2589`	🟪
Subtract	Subtraction operator (-)	`0x108967`	🟩
Multiply	Multiplication operator (*)	`0x89403b`	🟫
Divide	Division operator (/)	`0x788956`	🟩
Modulo	Modulo (remainder) operator (%)	`0x896765`	🟫
Power	Puts a to the power of b (** in python)	`0x00137f`	🟦
Logical NOT	Negates a bool value	`0x3f7f47`	🟩
Bitwise AND	Bitwise AND operator (&)	`0x1f0052`	🟪
Bitwise OR	Bitwise OR operator (|)	`0x0b514e`	🟦
Bitwise Shift Left	Bitwise shift left operator (<<)	`0x4e4f51`	⬛
Bitwise Shift Right	Bitwise shift right operator (>>)	`0x787c51`	🟫
Logical AND	Logical AND operator (&&)	`0x7a296c`	🟪
Logical OR	Logical OR operator (||)	`0xf5ff3f`	🟨
Greater than	Greater than operator (>)	`0xae63ff`	🟪
Greater than or equal to	Greater than or equal to operator (>=)	`0xff0094`	🟥
Equals	Equals operator (==)	`0xadffe2`	🟩
Less than or equal to	Less than or equal to operator (<=)	`0xffd6a5`	🟧
Less than	Less than operator (<)	`0xff5111`	🟥

Not equal	Not equal operator (!=)	`0xff3538`	🟥
Break	Break keyword (for loops)	`0xffd9cc`	🟧
Continue	Continue keyword (for loops)	`0xff5eae`	🟪
End of program	Marks the end of the program (needed at the end)	`0xedfff2`	⬜
Print	Print to console statement	`0xf18eff`	🟪
Equation	Defines an equation. Needed for all operator functions.	`0x16741a`	🟩
Import	Import statement for including other Pawet image programs	`0x42f301`	🟩
Struct start	Defines the start of a struct definition	`0x4affff`	🟦
Struct end	Defines the end of a struct definition	`0xceff96`	🟩

Structure of a Pawet Program

Pawet is purely functional, meaning there are no objects, or object oriented concepts directly defined in the language. A Pawet program is a series of pixels with certain colour values representing data and commands, where the image is read from left to right and from top to bottom. A PNG image is the best recommended format for Pawet programs due to its lossless compression feature and versatility. Pawet uses the red, green, and blue channels of images – the alpha (or transparency) channel is ignored and will still treat a transparent pixel as its original colour value without alpha.

Programs can be padded with whitespace (`0xffffff` pixel value). Pixels with these values (with the exception of values within literal or variable declarations, more on this later) are ignored and skipped by the interpreter. This can be used to define blocks and 'lines' of Pawet code within an image and to enhance readability.

Pawet images can be of any dimensions, as long as your program fits in the image. Pawet is not limited to images that are square, or widths and heights in powers of two. For example, an image can be a 1024x1024 pixel image and will make no difference if the image was larger (e.g. 1029x2064 pixels) or smaller (e.g. 250x100 pixels), granted the program fits in the smaller image. There is a trade-off of readability and size with larger image dimensions, and conciseness with smaller images with little or no whitespace. This is up to the programmer to decide and Pawet does not have a bias or recommend either way.

A Pawet program is defined as everything between the first pixel of an image and the End of Program pixel (`0xedfff2` pixel value). This marks the end of the program and that everything after will be ignored by the interpreter, regardless of subsequent pixel values. It is **compulsory** for an End of Program pixel present in every Pawet image program as this helps the interpreter to shorten time to execute Pawet image parsing. Only one of these pixel values can be present in a single image – multiple programs cannot be in the same image (but other programs can be imported and used in the same image, more on this later on). An interpreter error will be thrown if the end of the image is encountered before the End of Program pixel.

Variables and Literals

Variable Declaration

In Pawet, variables are references that can change their value throughout the program and have their own scope (read on). Variables are also *typed*, meaning they can be any type in the list of strings, floats, Booleans, or integers. Variables can also be a structure type and hold values in that structure or an array of values of a certain type.

To declare a new variable, a type must first be stated. This can either be a float, integer, string, or bool type for simple type declarations. Structure and array definitions and usage will be covered later on in this specification document. The type is a single pixel value representing that type:

Type	**Hex value of colour**	**Colour**
`Float`	`0x404040`	⬛
`Int`	`0x7f3300`	🟫
`String`	`0x3f3400`	🟫
`Boolean`	`0x0037ff`	🟦

There is no need for a '`char`' type in Pawet as one can simply use a string type with one character in it as both are treated the same way in the interpreter anyway. Furthermore, `float` and `int` values can be as large as you want – there is no upper, lower, or precision limit for these values. `Strings` are also treated as arrays of characters and can be indexed in such ways (more on this later).

Following the type declaration pixel, the variable name is given. This is demarcated (surrounded) by pixels with the colour value of `0xffd800`. Between these two pixel values, any value can be supplied and the interpreter will take these colour values and construct them into a variable name. Variable names are encoded using ASCII and take the form of three characters per pixel (as there are three separate 8 bit numbers in an RGB value). For example, a variable with the name of 'sample' will have the pixel values of `0x73616d` and `0x706c65` as there are six characters in the word 'sample' where 's' corresponds to the ASCII value of `0x73`, 'a' corresponds to the ASCII value of `0x61` and so on and so forth. Variable names with a count of characters not divisible by three are padded with `0x00` (so just the letter ‘x’ in a variable name will be `0x000078`). All variable names have at least one pixel value between the two variable name marker pixels. Only ASCII characters are supported for variable names at this current stage.

If in between the two pixels marking a variable name should there be a pixel value which corresponds to the surrounding pixels (`0xffd800` in this case), an escape pixel value is available which cancels the following special action of the next pixel (equivalent of backslash in strings in some languages). The pixel value for escaping the next pixel is `0x7f0037`. This also works on itself, meaning you can escape the escape pixel value (equivalent of double backslash in strings in some languages).

Variable declarations always include an assignment pixel value after the type declaration and the variable name. Everything to the right of this assignment pixel before the end of statement pixel value is treated as the value you are trying to assign the variable to. The assignment pixel value is `0xffc200`. Variable declarations are always ended by an End of Statement pixel, which has the hex value of `0x102010`.

It is also possible to define a new variable and assign it to an existing variable, which will copy the value over. Simply place the variable name declaration after the assignment pixel value to do this.

Defining literals

Literals are used to define the same number, string, or Boolean, such as 0, 32, 3.141, “blarg”, or False. These appear in equations (more on this later) or on the right side of a variable assignment statement. Literals on their own are invalid and will throw an interpreter error. There are different literal types which match the possible variable types.

Boolean literals are values that can either be true or false. Because there are only two possible values for this type, a ‘true’ and ‘false’ pixel value is given without the need for literal definition pixels unlike for the remainder of the types. The pixel hex value that corresponds to the value of true is `0x7f7a00` and false being `0xff006e`.

For integer literals, any whole number can be provided. This is done using the int literal pixel value demarcations, which have a hex value of `0xffc2e8`. Pixel values between these determine the actual value of the literal and are handled in a similar way to variable names. Each byte of an integer literal is a separate R, G, or B channel value and three bytes together make a single pixel. Should an integer not use a full 24 bits or three bytes, the pixel value is padded with `0x00`. For example, the number 0 is treated as `0x000000`, the number 25 is treated as `0x000019` in a single pixel, and the number 49658234873 as an example is treated as `0x00000b` and `0x8fdc87` as two pixels between the integer literal demarcations. Pixels to the left have a larger effect on the size of the integer as it is treated as little endian. Integer literals can be any size and have no realistic upper limits.

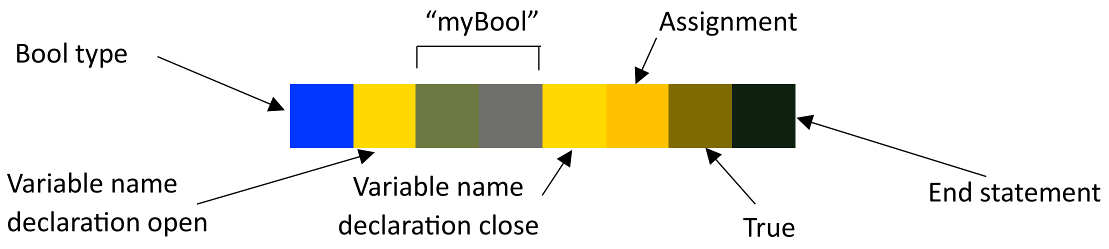
Float literals are parsed in a slightly different way and have two 'parts' to them. To define a float literal, surround the definition with the float literal pixel value on either side, whose hex value is `0xff7fb6`. Float literals have two parts: the first being the whole number value, and the second being the number after the decimal place (so the float literal 5.63 has a '5' part and a '63' part). The first part of the float literal is encoded as an integer and the second part of the literal is encoded as a string, separated by a point pixel value (`0xc9e0ff`). Both the whole number and decimal parts can be padded with `0x00` to create a whole pixel. For example, the float value 3.14159 would be encoded as `0x000003`, `0xc9e0ff` (the point pixel value), `0x003134`, and `0x313539` (first being the number 3 encoded as an integer, and the last two being the string '14159' encoded as ASCII). This is subject to change as the language develops.

String literals are treated in the same way as variable names, but with a different demarcation. Encode a string the same way as you would a variable name and surround it with the string literal pixel value of `0xff0000`. String literals can also be padded with `0x00` for instances where the string length is not divisible by 3. An example would be the string "abc123" becoming `0x616263` and `0x313233` in pixels.

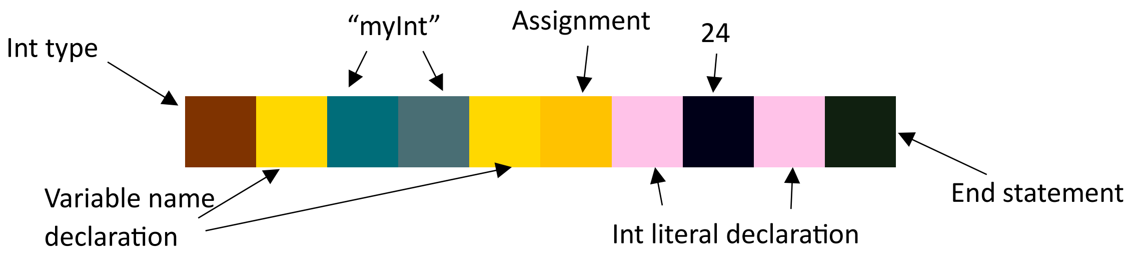
For negative float and integer literals, simply place the subtraction operator pixel value (`0x108967`) *before* the literal definition pixel value, not in between the literal definition pixels. It is only possible to do this with int and float types.

Variable and literal examples

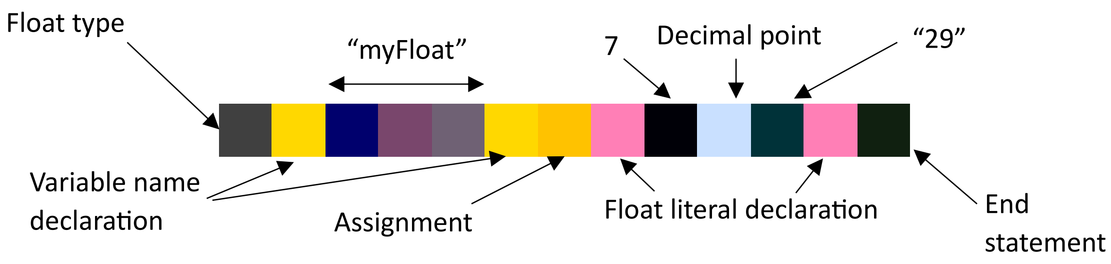
Defining a Boolean variable called 'myBool' and assigning it to the value of true:


"myBool"
Assignment
Bool type
Variable name declaration open
Variable name declaration close
True
End statement

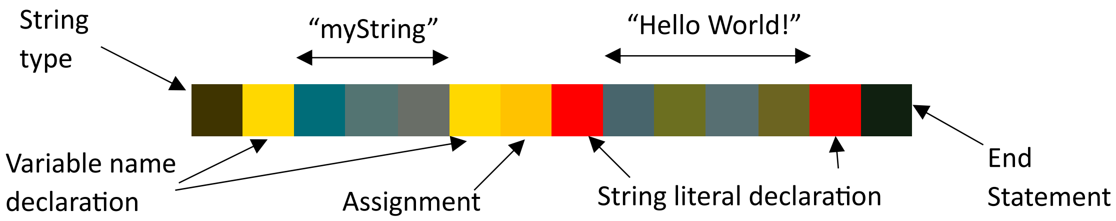
Defining an int variable called 'myInt' and assigning it to the value of 24:



Defining a float variable called 'myFloat' and assigning it to the value of 7.29:



Defining a string variable called 'myString' and assigning it to the value of "Hello world!"



Equations and operators

In Pawet; bitwise, logical, and mathematical operations are possible. However, all of these operations must be done in the special 'equation' context, demarcated by a certain pixel value in the same way variable names and literals are demarcated. The pixel value for the demarcation of equations is `0x16741a`. Equation contexts can appear almost anywhere, including after the assignment pixel for variable declarations, when calling functions, and return statements for functions. A rule of thumb is where you can call a function you can also create an equation context. On their own, equation contexts are invalid. Equations in a statement must also terminate after closing the equation context with an end of statement pixel value.

Equations are solely evaluated from left to right, completely ignoring the order of operations present in other languages and mathematical notation. To define your own order of operation, use brackets (opening bracket pixel value is `0x000000` and closing bracket pixel value is `0xafafaf`). Brackets can be nested and are executed individually from innermost to outermost. For example, the mathematical equation `a + b / c - d` will be evaluated in the order of `a + b` then `(a + b) / c` then `((a + b) / c) - d`. Pawet is designed this way to ensure an O(N) interpretation time and complexity of an equation.

Equations evaluate to a single value which can be of any type depending on the variables and literals used in the equation. For example if solely integers are used, then the equation will evaluate to an integer. Conversely, if the first variable is a string (e.g. `str1 + 3 + str2`), then the final result will be a string. This is the same for logical and comparative operators returning a Boolean value as well. Some operators can only be applied to certain types, such as bitwise operators only allow integers. Below is a list of operators that can be used in an equation context which also compares their equivalents in other languages and what types they support.

Operator List

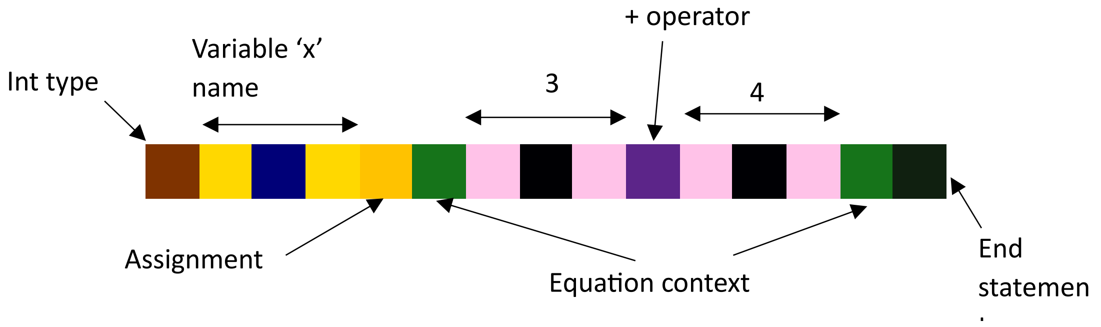
Operator	**Category**	**Hex value**	**Colour**	**Types supported**	**Equivalent in *C* languages**	**Equivalent in *Python*.**
Add	Multiple	`0x5c2589`	🟪	String, int, float	+	+
Subtract	Mathematical	`0x108967`	🟩	Int, float	-	-

Multiply	Mathematical	`0x89403b`	🟫	Int, float	*	*
Divide	Mathematical	`0x788956`	🟩	Int, float	/	/
Modulo	Mathematical	`0x896765`	🟫	Int, float	%	%
Power	Mathematical	`0x00137f`	🟦	Int, float	n/a	**
NOT	Logical	`0x3f7f47`	🟩	Bool	!	not
AND	Logical	`0x7a296c`	🟪	Bool	&&	and
OR	Logical	`0x0b514e`	🟩	Bool	||	or
Greater	Logical	`0xae63ff`	🟪	Int, float	>	>
Greater equal	Logical	`0xff0094`	🟥	Int, float	>=	>=
Equal	Logical	`0xadffe2`	🟩	All	==	==
Not equal	Logical	`0xff3538`	🟥	All	!=	!=
Less	Logical	`0xff5111`	🟧	Int, float	<	<
Less equal	Logical	`0xffd6a5`	🟨	Int, float	<=	<=
AND	Bitwise	`0x1f0052`	🟪	Int	&	&
OR	Bitwise	`0x0b514e`	🟩	Int	|	|
Shift left	Bitwise	`0x4e4f51`	⬛	Int	<<	<<
Shift right	Bitwise	`0x787c51`	🟩	Int	>>	>>

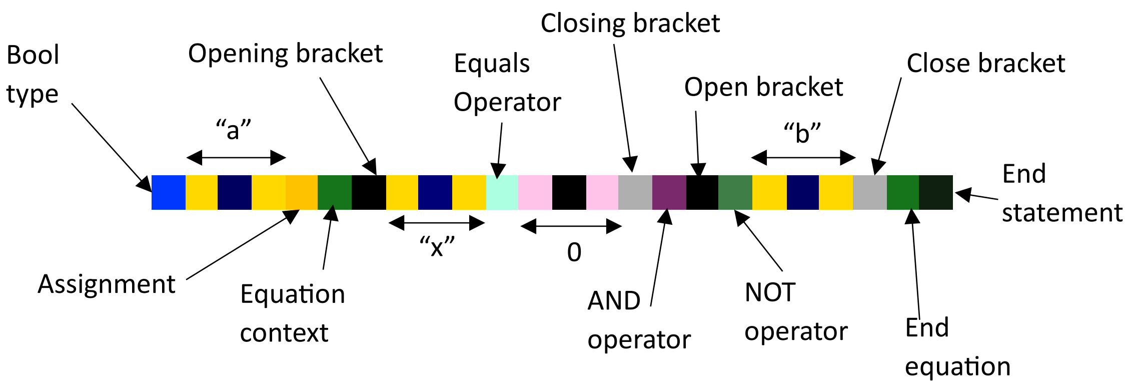
Note that all operators require two inputs with the exception of NOT. The NOT operator comes *before* the value or variable you want to invert. The subtraction operator can also be used to mark an int or float literal as below zero or negative. This also works for variable references but must still be done in the equation context.

Equation examples

Creates an int variable called 'x' and assigns it to the value of 3 + 4:



Creates a bool variable called ‘a’ and assigns it to the value of `(x == 0) && !b` (assuming ‘b’ is a bool):



An important note when interpreting equations using logical operators: lazy evaluation is not present. This means that should you have a Pawet snippet with the equivalent code in *Python* being `(a and b())`, the function call `b()` will *still* be evaluated even if the a variable evaluates to `false`.

Functions

Pawet supports user-defined functions to bundle code together for re-use. They are invoked in the same way as referencing a variable but have an additional set of values passed in between bracket pixel values. Functions in Pawet have a return type and typed function parameters (the function signature). Any code between the function definition pixel values (`0xb200ff` to start a function definition, and `0x57007f` to mark the end of a function body) will be bundled as part of the function and executed in a separate interpreter instance (with access to global variables) when called from any context.

The function syntax (visual examples shown further below) include the function definition pixel value, a return type pixel, then the function name (defined exactly the same way as you would define or reference a variable with a name), and then the function parameters enclosed in bracket pixel values. If a function does not return anything, then declare the function return type as the void type pixel value (which is `0xff7fed`). Functions cannot be defined from within functions – they can only be defined in the base context of the Pawet program and not in any control flow or function body.

The starting point of execution of a Pawet program is the first encountered statement, meaning that a 'main' function is not needed as the entry point of a Pawet Program. Pawet will also only define and store functions as it encounters them rather than upon execution. This means that functions definitions and body code have to appear ***before*** they are referenced/used in the program to ensure an O(N) interpretation complexity for Pawet programs. Therefore, function *a* which references function *b* which references function *a* again from within is not valid. Functions however can call themselves again (recursion) as long as an appropriate base case is implemented in the body as per standard programming practise.

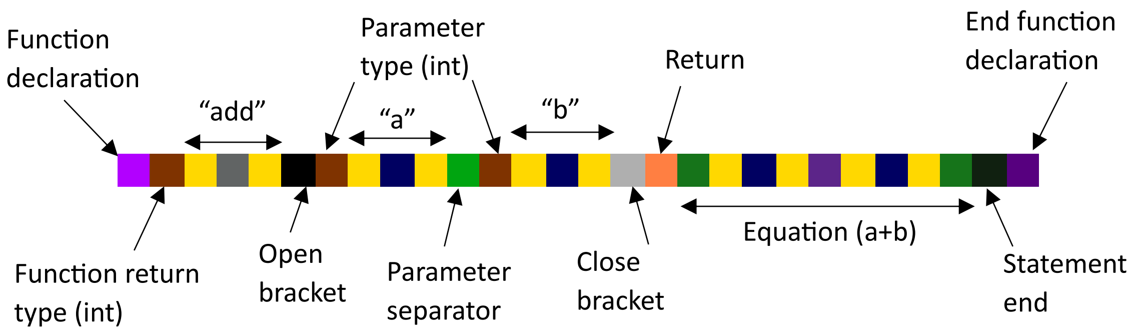
Functions have their own internal variable scope as well as access to the global variable scope. This means that variables declared and initialized within a function only exist for within that function body. Any global variables that have their values re-assigned within the function will update that variable in the global scope. Therefore, it is recommended that variables that you only need to use within a function are defined within a function body.

To programmatically exit a function during its execution, use a return statement pixel value. The hex value for this pixel is `0xff7f42`. This is used to return a value based on the function's return type and/or exit the function itself and return to the enclosing scope from where the function was called. Remember that the end of a return statement must be finished with an End of Statement pixel. Any return statements after the first encountered statement during interpretation (return statements in if-statements are only executed if the enclosing if body condition is true for example) are ignored and is considered dead code as it can never be reached.

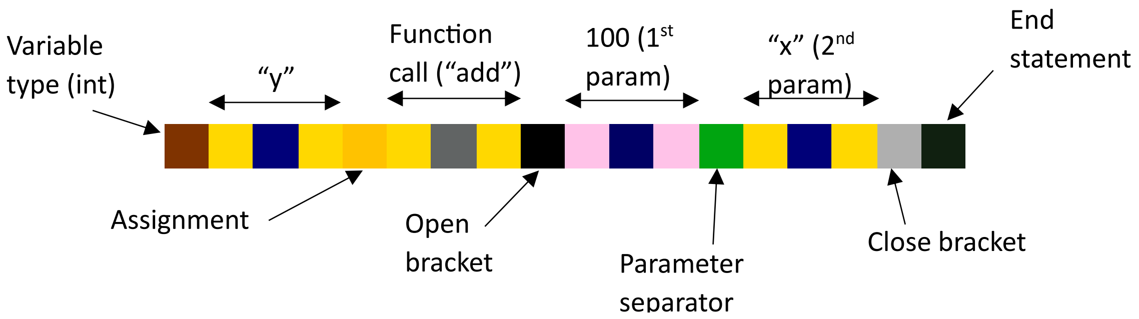
Function parameters are defined within the opening and closing bracket pixel values as a type pixel then the variable name and are separated by a parameter separator pixel value which acts like a comma in other text-based programming languages. The pixel hex value for the function parameter separator is `0x00a510`. For functions with only one parameter, the separator pixel is not needed.

Function definitions that have no parameters are simply declared with a bracket start and end pixel with no other pixels in between. Function parameters can be any supported type by Pawet, a structure type, or an array (more on structures and arrays further down) and are declared the same way you would declare a variable name (see examples below).

Example of a function named “add” which takes two integer parameters named ‘a’ and ‘b’ and returns the sum of them together:



Example of calling the previously defined function “add” with the parameters 100 and “x” (where x is a previously declared int variable) and assigning it to a new variable “y”:



Function parameters are passed by value when invoking, meaning no changes will be made to the original parameter or variable when calling a function. It is also possible to call a function inline within another function call as a parameter, as long as the syntax for the inline function call is correct. Function calls can be inserted virtually anywhere provided statements are closed if necessary or the syntax for a control flow body is correct and that the function exists as a definition.

A final important note is that functions are always global. There is currently no support for inner functions and thus a function call can be done anywhere in a program given that it is declared before calling. Function names cannot be duplicates either – they must have a unique name irrespective of the function return type and parameter signature. This also includes if any functions from a different program/library are imported as the function definitions of an external imported program are added to the function register of the current Pawet program. Pawet does not currently support function overloading.

Control Flow

If-then-else statements

In order for Pawet to be Turing complete, some form of control flow is needed. Therefore, Pawet supports if then else statements where you can conditionally control whether code is executed. The starting if statement block will evaluate the condition given and if that evaluates to true, it will interpret the contents of the body. Should this condition evaluate to false, the then statements are sequentially interpreted in order and will interpret the body if the condition for the then statement is true. Failing all of this, the interpreter will interpret the else body pixel code. ‘Then’ and ‘else’ statement bodies are completely optional to an if statement. Because there is an option to chain conditions in ‘them’ statements from an if statement, there is no need for `switch` statements and thus are not present in Pawet. If and then statement conditions *must* evaluate to a bool for it to be a valid condition.

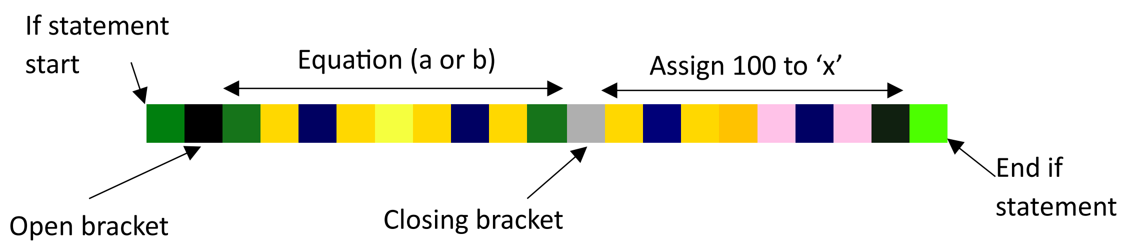
The syntax for an if statement is first the if statement pixel value, then an opening bracket pixel. After this, you can place a function call, a variable, or an equation into the condition for the if statement. The condition is evaluated by value and will not modify anything and uses the parent variable scope of the if statement. After the closing bracket is placed after the condition, anything after is interpreted separately until the interpreter encounters an end if statement pixel value. If statements can also be nested, as long as they are closed with the end if statement pixel. The variable scope applies to nested if statements, where an internal if statement has its own scope for variables and access to the parent scope variables.

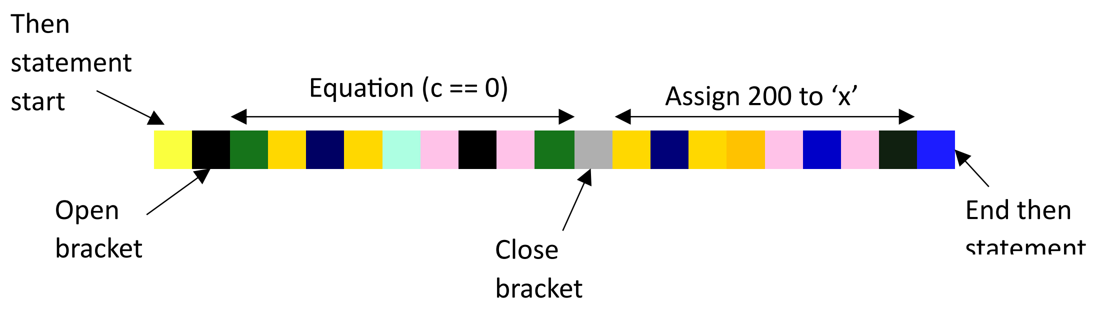
A then statement is very similar in syntax to an if statement – the only difference being a different pixel value for ‘then’ and that it is terminated with an end then pixel value. The conditions for following then statements are evaluated in order of appearance and the body is executed should this condition be true. Then statements use *lazy evaluation*, where following then statement conditions will not be evaluated at all should the previous then statement condition evaluate to `true` and that body executed.

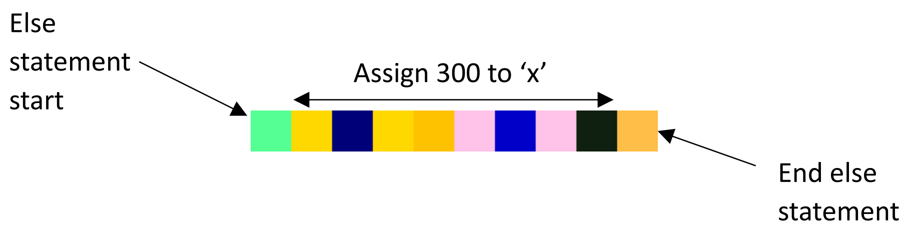
Should the if statement evaluate to false and any subsequent then statements also evaluate to `false`. If an else statement is present, this will then be executed. Else bodies act as the ‘fallback’ condition and are optional. Else statement syntaxes just require the else pixel value followed by the body code and then followed by the end else pixel value. There are no brackets or conditions for an else statement. See below for a table on all the colour values for these blocks.

Name	Description	Hex value	Colour
If	Starts the declaration of an if statement	`0x007f0e`	🟩
End if	Marks the end of an if statement body	`0x4cff00`	🟩
Then	Starts the declaration of a then statement	`0xfaff3e`	🟨
End then	Marks the end of a then statement body	`0x1c1cff`	🟦
Else	Starts the declaration of a fallback else statement	`0x56ff94`	🟩
End else	Marks the end of an else statement body	`0xffbe47`	🟧

Example of an if-then-else statement, where the if statement condition is an equation of a `or` b and sets the value of x to 100, the first and only then statement has a condition of `c == 0` and sets the value of x to 200, and finally the else statement sets the value of x to 300 given a and b are bool variables, and c is an integer variable (note the if, then, and else statements are split over multiple lines for readability):







Loops

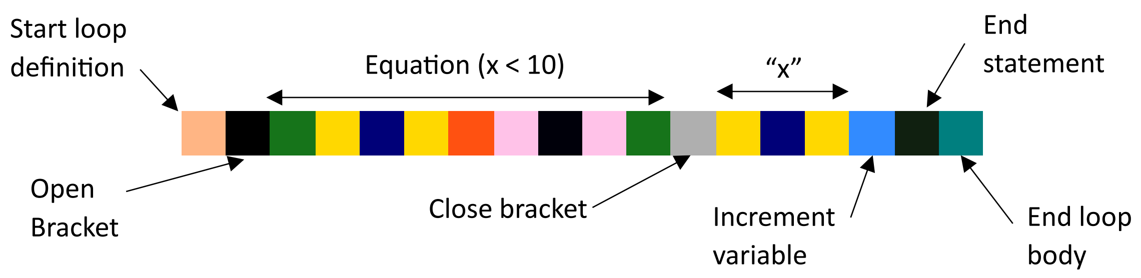
Pawet also supports basic loops as part of control flow. These follow the same syntax as an if statement but the condition is evaluated on every iteration and the loop body executed every iteration until the condition evaluates to false. The condition is evaluated *before* executing the body. There is only one type of loop present in Pawet at this stage, and is the equivalent of a `while` loop in other languages such as *Python* or *C*. There is no for, foreach, or do-while loop present in Pawet as these loop variants do appear in other languages. Therefore, you will need to define your variables before the loop definition in the case of iterating a certain amount of times (i.e. `for(int i = 0; i < 10; i++)`) and modify them in the loop body itself. Different variants of loops may be added later on as the language develops and evolves.

Loops are started with the loop pixel value (`0xffb584`) and terminated with the end loop pixel value (`0x007f7f`). Anything in between these pixels will be interpreted as the body of the loop. Loops can be nested too, which follow the same to nested if statement variable scopes. Loop bodies also have their own scope which exists for the current iteration, meaning that variables defined in a loop are redeclared on every iteration. Following the loop declaration pixel value, opening and closing bracket pixels must be straight after, where in between the brackets is the condition expression, which will be a function call, variable reference, a bool literal, or an equation.

It is possible to exit a loop using the break pixel value (`0xffd9cc`). Breaking a loop just requires this pixel and will exit the current loop when encountered. It is only possible to break out of the current loop and not any parent loops if the current loop is nested. No end of statement pixel after the break pixel is needed, however the loop still needs to be terminated with an end of loop pixel where appropriate.

Additionally, it also possible to skip the rest of the current loop iteration and move to the next iteration (much like the `continue` statement in other languages). After this pixel value (which is `0xff5eae`) is encountered, the rest of the loop is skipped, the condition re-evaluated, and if that is true then will perform another iteration of the current loop. The continue pixel will only skip the current iteration of the current loop, not affecting any parent loop bodies. Like with the break pixel, no end of statement pixel after the continue pixel is needed either.

Example of a loop that checks if the int variable ‘x’ is less than 10 and increments the variable ‘x’ in the body:



Miscellaneous topics

Importing other Pawet programs

It is possible to include other Pawet programs within your current Pawet program and use the variables, functions, and structures set out in the external programs. When another Pawet program is imported, all functions, variables, and structure definitions are included into the current Pawet program context and thus can be called and referenced from your current working program. This will be how to include the standard library programs of Pawet into your own Pawet code. Note that all code in the global scope of an imported Pawet program is also executed.

To import another program, use the import pixel value (`0x42f301`) followed by a string literal defining the relative path to your file. At the end of a string literal, an end of statement pixel value is needed. If you need an absolute path, set the import string literal as the full path to your external Pawet program. As the language develops, the methods and syntax for importing may change and improve, in addition to being able to use the standard Pawet library that comes with Pawet by default (such as math functions, file I/O functions, etc.). Use the "Pawet Standard Library" PDF document for further information of what programs and functions the standard library provides.

Printing to the Python console

Pawet includes a shorthand for printing values to the *Python* console (as the Pawet interpreter uses Python, and running a Pawet program requires Python and the terminal to use) as part of the default pixel instruction list. Surround your value with print pixel values (`0xf18eff`) and the literal (of any type), variable reference, function call, or equation to print the value of it to the *Python* console. Print statements need an end of statement pixel value after the second print pixel value to mark the statement as complete. Print statements can be called in any control flow or function body, or in the global program space as one would a variable assignment or standalone function call.