Macro Creation Tutorial

Part 1: Basic Programming Concepts

Klipper macros are quite powerful, but at the same time can be limited in their scope. The idea for macros is to be able to make some basic decisions and act on them by substituting various values with other values. To understand macros, though, we must first understand some programming basics

What is a function?

A function is a bit of code that produces output based on the input. Typically decisions are made inside of a function via expressions that determine what the function should do. Functions can also be called methods, routines, procedures. They all do basically the same thing: Take a block of code and give it a name so it can be reused to produce output based on input.

What is a literal?

A literal is a value that is effectively hard coded and cannot be changed throughout the execution of a program. For instance, a integer value of `42` is a literal. The value of `42` is literally `42`.

What is a comment?

The intended use of a comment is for the person writing code to leave notes for themselves or anyone else should they come back and try to understand the code later. Comments are denoted by a special character or set of characters specific to each programming language. In the case of Klipper config files and macros, there are two such characters used to denote a comment. `#` and `;` . When Either of these appear on a line, anything after them will be ignored.

```
#Move the toolhead to the middle of the bed
G1 X150 Y150

#Move the toolhead up slightly
G1 Z1
```

Comments can also be beneficial for debugging or changing code around. If you have a bit of code that might not be working right or you want try to make it do something else, but don't want to delete the original bit in case it doesn't work, you can "comment out" that bit of code

```
#Move the toolhead to the middle of the bed
```

```
#G1 X150 Y150

#Move the toolhead up slightly
G1 Z1
```

In the example above, the newly commented line will be ignored when it comes time to execute the macro.

What is a variable?

A variable is a value that can change throughout the execution of a program. Variables are typically represented by names or letters. The name is associated with a value because the value can change. When the variable name is referenced during the execution of a block of code, the name is substituted for the value of the variable at the time the expression is executed. So if we take our literal `42` and assign it to a variable called `e` whenever `e` is referenced, the value `42` will be used.

What is an assignment?

Giving a value to a variable is done by assignment. This can be a literal value, the value of another variable, or the result of an expression. An assignment is done by setting the variable equal to a value. In Jinja2 syntax, this is done with

```
{ set x = 42 }
```

What is an expression?

An expression is a programming command that does something. A simple example would be a gcode command. `M105` is a gcode expression that reports the various temperatures. A variable assignment is also an expression. A comparison of two values is an expression. In the simplest terms, an expression is a bit of code that produces a result or value.

What is an object?

An object is a collection of variables, states, functions, and possibly more objects. Objects are used to organize and access entities in a program. For instance, in Klipper macros, there is a `printer` object which has another "fan" object attached to it. The fan has a "speed" variable which is used to both get and set the fan speed. To get the fan speed, one would use

```
printer.fan.speed
```

What is a parameter?

A parameter is a value that is used as the input for the execution of a function. In gcode, the function would be the gcode command and the parameters would be the axis positions and speeds.

```
G1 X100 Y24 Z1 F2000
```

In other programming languages like python, functions are represented by a name and the parameters are enclosed in parenthesis after the name.

```
action_respond_info("String Literal")
```

What is a conditional?

Similarly, expressions can also evaluate two things. Conditionals answer a `Yes` or `No`, `True` or `False`, `1` or `0` question by evaluating one variable or literal against another variable or literal. This is “boolean” logic and it is at the core of all programming.

```
Is 15 equal to 12?
False

Is EXTRUDER_TEMP > 100?
True
```

Similarly, functions can return a value after they execute and that value can be compared in an expression as well.

There say you have a function called `EXTRUDER_PERCENT_TO_TARGET()` and it takes two parameters. It returns the percentage of the `CURRENT_TEMP` to the `TARGET_TEMP`
`Is EXTRUDER_PERCENT_TO_TARGET( CURRENT_TEMP, TARGET_TEMP ) > 70? False`
What is a macro?

A macro is a stored set of commands that can be called from a single gcode command. Macros can call other macros or even themselves. Macros can be treated like functions. For all intents and purposes, a klipper macro is a function.

What is a Jinja?

Jinja is an engine for python (Klipper’s primary language) that can take a block of text and read programming elements out of it. In the case of gcode macros those programming elements are applied as a command template.

A command template could be looked at like a paragraph of fill in the blanks. Like Mad Libs meets Choose Your Own Adventure. When the gcode macro is called, the entirety of the command template is read in and evaluated. The gcode that is expressed back to klipper is the result of whatever decisions were made inside of that macro.

What is delayed gcode?

Delayed gcode is a macro that cannot be called directly, but is instead called and executed from a timer at a set interval. Delayed gcode can run once or repeatedly

Part 2: Anatomy of a Macro

Defining a macro.

Now that some core concepts are out of the way, we will make a common start print macro that is called by the slicer when a print starts. It will move the nozzle to the corner of the bed and draw a priming line.

In the config file we first need to define the macro. Macros are prefixed with `gcode_macro` followed by a space and then then name of the macro.

```
[gcode_macro start_print]
```

The next thing that is needed is the actual gcode to be executed. After the macro declaration, we need to define a `gcode:` section for the macro. Once the `gcode:` section has been added, we can put the gcode into the template.

The actual gcode commands *must* be indented otherwise the config file will fail to load. You can see in this macro, each line has been commented so that it's purpose is understood.

```
[gcode_macro start_print]
gcode:
        G92 E0                                          # Reset
Extruder
        G1 Z2.0 F3000                                   # Move Z Axis to
travel height
        G1 X0.1 Y20 Z0.2 F5000.0                # Move to start
position
        G1 X0.1 Y200.0 Z0.2 F1500.0 E15 # Draw the first line
        G1 X0.4 Y200.0 Z0.2 F5000.0  # Move to side a little
        G1 X0.4 Y20 Z0.3 F1500.0 E30            # Draw the second line
        G92 E0                                          # Reset
Extruder
        G1 Z2.0 F3000                                   # Move Z Axis up
to travel height
```

We can then call the macro at any time from the terminal or from a gcode file by simply adding `start_print`. This could also be included in the slicer's start-gcode section so that it is called when the machine starts executing the

file. When the file is being read by Klipper or Octoprint or whatever, it will come to the line `start_print` and that command will be substituted for the set of commands in the macro.

Part 3: Parameters and Basic Conditionals

If the axes aren't homed or the nozzle is not to temp, it will fail like anything else because of safety checks.

What if we wanted to incorporate some of that into my `start_print` macro? We would need to get values from the slicer, such as extruder and bed temperature back to our macro somehow.

We can use parameters for this.

To add a parameter for a macro, all we have to do is reference it. Within Jinja2 templates, programmatic actions need to be encapsulated within curly braces. In this case, we are simply substituting the name of the parameter for it's value. Parameters passed to a gcode macro are contained within the `params` collection. Due to how parameters are handled in the gcode parser, they must be capitalized when referencing them in a macro.

```
[gcode_macro start_print]
gcode:
        M109 S{ params.TOOL_TEMP }          # Heat the tool to
temperature and wait
        G92 E0                                          # Reset
Extruder
        G1 Z2.0 F3000                               # Move Z Axis to
travel height
        G1 X0.1 Y20 Z0.2 F5000.0            # Move to start
position
        G1 X0.1 Y200.0 Z0.2 F1500.0 E15 # Draw the first line
        G1 X0.4 Y200.0 Z0.2 F5000.0  # Move to side a little
        G1 X0.4 Y20 Z0.3 F1500.0 E30        # Draw the second line
        G92 E0                                          # Reset
Extruder
        G1 Z2.0 F3000                               # Move Z Axis up
to travel height
```

From now on, whenever `start_print` is called, `{ params.TOOL_TEMP }` will be replaced with whatever value the `tool_temp` parameter is assigned.

```
start_print tool_temp=200
```

If our slicer supports variables in its gcode templates like Slic3r variants, for instance, we could substitute the 200 for whatever that variable is in the slicer. So in the start gcode, we could put something like

```
start_print tool_temp=[first_layer_temperature]
```

when the gcode file is written out, the slicer substitutes the variable `[first_layer_temperature]` for the temperature that was specified in the slicer settings for that particular filament. This would produce gcode in the output file similar to

```
start_print tool_temp=200
```

When the macro is called, the 200 is being assigned to the `tool_temp`, parameter. When the `tool_temp` variable is referenced, it is substituted for 200.

```
M109 S200                               # Heat the tool to temperature and wait
```

This still doesn't solve the problem of axes that aren't homed. We could just call G28 at the start of the macro, but that would result in the axes homing again even if they are already homed. With macros, we can actually check to see if we need to home by using a conditional and referencing the toolhead object.

In Jinja2, a conditional is prefixed with `{%` followed by the type of conditional to use followed by the expression to be evaluated closed with `%}` In this case, we would want to use an `if` statement. An `if` statement is a conditional that compares two things and results in a `true` or "false" answer. If the resulting answer is `true` then the code in the `if` statement is executed, if it is `false` then the code is ignored or the `else` is executed. More on that later.

Klipper has certain "virtual" objects exposed to the macro ecosystem so that this sort of thing can be accomplished. In our macro example here, we are looking to figure out which axes are currently homed. To get there we need to reference the `printer` object. The `printer` object has a field called `homed_axes` which is a string of characters that represent each axis that is currently homed. So `XY` would mean both X and Y are homed, but Z is not.

The `printer.homed_axes` object will always contain the axes in the order of `XYZ` so to check to see if all 3 axes are homed, we merely need to make sure the value of `printer.homed_axes` is equal to `XYZ`. Since we want to perform an action if they are *not* homed we need to use the *not equal* comparision operator, `!=` .

In plain English, we are trying to say "If all 3 axes are not homed, home them"

In Jinja2, we express that as

```
{% if printer.homed_axes != 'XYZ' %}
        G28                    #Home All Axes
{% endif %}
```

The condition is typed on the first line encased in the curly braces with percent signs. The code to execute is on the next line. Finally, the `{% endif %}` tag is added to close the statement. Anything in between the `if` and `endif` is executed if the `if` statement evaluates `true`.

So when the macro is called, if one or more of the printer's axes are not homed, G28 will be called to home them.

```
[gcode_macro start_print]
gcode:
        M109 S{params.TOOL_TEMP}            # Heat the tool to
temperature and wait

        {% if printer.homed_axes != 'XYZ' %}
                G28
    #Home All Axes
        {% endif %}

        G92 E0                                                  # Reset
Extruder
        G1 Z2.0 F3000                                  # Move Z Axis to
travel height
        G1 X0.1 Y20 Z0.2 F5000.0            # Move to start
position
        G1 X0.1 Y200.0 Z0.2 F1500.0 E15 # Draw the first line
        G1 X0.4 Y200.0 Z0.2 F5000.0  # Move to side a little
        G1 X0.4 Y20 Z0.3 F1500.0 E30        # Draw the second line
        G92 E0                                                  # Reset
Extruder
        G1 Z2.0 F3000                                  # Move Z Axis up
to travel height
```

Part 4: Default Parameters

One drawback to supplying a parameter arbitrarily in a macro is that it can be referenced without a value. This could result in undesirable behavior or and error at runtime if it is not handled. One way to check to see if a parameter exits is simply to evaluate it.

```
{% if PA %}
{% if params.PA %}
```

Gcode parameters are accessed in macros through either directly referencing the parameter name or by using the `params` collection. There are some nuances to using the `params` collection in that, it will only have values that are passed in through the gcode command. Whereas directly referencing the parameter name can allow us to get a default value for 'default_parameter_PA'

Jinja affords us the use of certain filters for things like rounding and typing (more on that later). There is also a `default()` filter which can be used instead of declaring a `default_parameter_PA`.

```
{ params.PA|default(.06) }
```

So if we pass the a parameter through for PA and we want to set pressure advance in our start print macro, we can check for the parameter and then call the `SET_PRESSURE_ADVANCE` command.

```
SET_PRESSURE_ADVANCE ADVANCE={ params.PA|default(.06) }
```

Alternatively you can give it a default value using the `default_parameter` config option for the gcode macro.

```
[gcode_macro start_print]
default_parameter_PA: 0.06
gcode:
```

For this example, we are adding a `Pressure Advance` setting to our start print macro so that the PA can be set based on a particular filament

```
[gcode_macro start_print]
default_parameter_PA: 0.06
gcode:
        M109 S{params.TOOL_TEMP}            # Heat the tool to
temperature and wait

        {% if printer.homed_axes != 'XYZ' %}
                G28
    #Home All Axes
        {% endif %}

        SET_PRESSURE_ADVANCE ADVANCE={PA}

        G92 E0                                                  # Reset
Extruder
        G1 Z2.0 F3000                                  # Move Z Axis to
travel height
        G1 X0.1 Y20 Z0.2 F5000.0            # Move to start
position
```

```
            G1 X0.1 Y200.0 Z0.2 F1500.0 E15 # Draw the first line
            G1 X0.4 Y200.0 Z0.2 F5000.0  # Move to side a little
            G1 X0.4 Y20 Z0.3 F1500.0 E30        # Draw the second line
            G92 E0                                                  # Reset
Extruder
            G1 Z2.0 F3000                                   # Move Z Axis up
to travel height
```


Part 5: Data Types, Assignments, Casting, and Scope

Data Types

Previously we have discussed variables and objects and "types" have been mentioned. In the world of programming, everything that has a value has a type. The type refers to how that value is stored in memory and how it is used within the program. Individual characters and words are called `strings`. By default, any parameters being passed into a macro are created as strings. Many values accessed within a macro are also `string` types.

When using a `string` in a comparison or assignment, the value *must* be encapsulated in quotes. Single or double quotes are allowable to be used as long as the beginning quote matches the end quote. So

```
{% if printer.homed_axes != 'XYZ' %}
```

and

```
{% if printer.homed_axes != "XYZ" %}
```

are both valid, but

```
{% if printer.homed_axes != "XYZ' %}
```

is not. String comparisons are *always* case-sensitive, meaning

```
'My Value' is not equal to 'my value'
```

In addition to `string`, there are also `float`, and `int` types (plus many more, but that is another topic).

A `float` is any numerical value that is not a whole number (has a decimal point). It is called `float` as shorthand for "floating point". Floating point means that the number of digits before and after the decimal point varies. This is really important for the program but not so much for us. All we should concern ourselves with is that it is not a whole number.

If we need only need whole numbers we can use an `int` data type.

Assignments

Sometimes it may be necessary to create a variable inside of our macro that is not being supplied by a parameter, or if a parameter is supplied and we want to access it as particular type for the scope of our macro.

To create a new variable, we use the Jinja2 `set` expression. So if we wanted to create a variable called “toolTemp” and have it be the value of the parameter ‘TOOL_TEMP’ we would do

```
{% set toolTemp = params.TOOL_TEMP %}
```

This would create a a variable called `toolTemp` and it would be equal to the value of the TOOL_TEMP parameter *at the time* the `set` operation was performed. As previously mentioned, parameters are always a `string`, so before we can do any math or comparisions on it, we need to make it into something a bit more numeric.

Casting

Casting refers to changing one type to another. Going from an `int` to a `float` is fine. An `int` value of 42 cast as a `float` would yield `42.000..`. Going from a `float` to an `int`, however, “truncates” the digits after the decimal point, so `42.9999` would become `42` in many cases you will want to round the number up to the nearest whole number *before* casting.

So how do we cast types? Jinja2 has what are called filters. Filters are applied by supplying a | (pipe) followed by the filter type. So to cast our TOOL_TEMP parameter `string` as an `int`, we would do

```
params.TOOL_TEMP|int
```

in order to round a `float` to the nearest whole number in order to cast it as an `int` we need to apply the `round` filter the round filter takes a parameter that specifies how many digits after the decimal point to round the number to. So to get the variable `a` rounded to the nearest whole number and cast it to `int`, the filters would look like this

```
a|round(0)|int
```

Why are types important? Say in our start print macro we want to preheat our bed and hotend at the same time but we want to add a sort of wait timer at the end so the bed temperature has time to “soak in” to the entire bed.

We want this time to be calculated based on the temperature the bed was when the start print macro was called vs the target temperature of the print. If we do something like

```
{% set dwell = params.BED_TEMP * 3 %}
```

If our BED_TEMP parameter is `10` then `BED_TEMP * 3` would be `101010` because the `*` operator when applied to a string value duplicates that string value a number of times. So to multiply bed temp by 3, we have to first cast it as an `int`.

```
{% set dwell = params.BED_TEMP|int * 3 %}
```

Scope

Any variable we create within a macro only exists within that macro and can only be accessed within that macro. This is called scope. There are some cases where a variable can only be accessed from within a control structure (such a for loop) and those will be discussed later.

Bringing it together

So let us apply the delay to our macro. If my starting bed temperature is already at the target temperature because we have been preheating things, we can say that we only want to “soak it” for 1 minute.

If the bed temperature at the start of the macro execution is less than 20 degrees below the target, we want to “soak it” for 5 minutes.

```
{% set target  = params.BED_TEMP|int %}
{% set current = printer.heater_bed.temperature %}

{% if current < target - 20 %}
       G4 P{ 5 * 60 * 1000 }       #Milliseconds to dwell
{% else %}
       G4 P{ 1 * 60 * 1000 }
{% endif %}
```

Part 6: Collections

What is it?

Programming languages have long supported basic collections in the form of arrays. An array is a single variable that represents a location in memory that has been divided up into chunks that can be accessed with an index. So if for instance we needed to store 3 points of data, like a single RGB color, we could do so by making am array that contains 3 elements. The order of the colors in

this array would be Red then Green then Blue. Accessing the value for each is done by specifying the index for each.

```
red   = myColor[0]
green = myColor[1]
blue  = myColor[2]
```

With arrays, it is also possible to specify additional dimensions. If we wanted to store multiple RGB color values in a single array, like for a bitmap image, we could define that array with a second dimension that represents a row of pixels and the color values for each column pixel in that row. A third dimension could be added that holds all of the rows together.

```
                                        Row
                                        |  Column
                                        |  |  Color
                                        V  V  V
pixel = bitmapArray[2][1][2]
```

Each programming language has its own specific way of dealing with things. It is beyond the scope of this tutorial to go into great detail with those things as, in many ways, arrays are rather primitive by todays standards. It is important to mention them though because they are the underpinnings for more advanced “collection” types. Many high level programming languages support these in various forms and python is no exception.

Jinja2 supports 3 collection types out of the box: List, Tuple, and Dictionary. By extension, Python supports these collection types as well since it is written on top of Python and there is a lot of overlap.

Lists and Tuples

Lists and tuples are an ordered series of values stored in a single variable, much like the array discussed before. They are more advanced than an array, however, as they contain additional methods and functionality beyond assignment and access.

In various programming languages, we have what are called Mutable and Immutable objects. Mutable, is one of those :atin words smart people like to use. It means something along the line of “subject to change”. Mutable objects can be changed once they have been created. Immutable types, cannot be changed.

A List object in Jinja2 is a mutable type, this means it can be changed after it has been created. Tuples on the other hand cannot be changed after they are created. Both types are iterable and ordered in that, when you access them the element at position [*n*] will always be the same.

Defining a list is as simple as providing a comma separated list of values that are wrapped in square brackets.

```
{% set myList = [ 2, 4, 6, 8, 10] %}
```

Tuples are declared the same way, but use parentheses

```
{% set myTuple = ( 2, 4, 6, 8, 10) %}
```

Dictionaries

Dictionaries are a little more advanced. These types use *key-value* pairs to represent their data elements. Each element in the dictionary has 2 items associated with it, they key and the value. They are very useful in that you can have a single variable representing a number of different things with a number of different values.

Collections can also be collections of other collections or types and objects organized into a large data structure.

Why is any of this important?

Klipper exposes the `printer` object to the macro system. This object is very useful for obtaining various values related to the current state of the printer. It has been referenced several times already in this tutorial, but up until now it hasn't really been explained.

Part 7: The Printer Object

Many Klipper modules have what is called a *get_status() wrapper* these wrappers are functions that report the state of the module. For instance, the `toolhead` get_status wrapper reports things like `position`, `max_velocity`, and `max_accel`. The get_status wrapper is added to the `printer` object when Klipper starts up. By navigating the `printer` object's hierarchy, we can obtain all manner of useful information that can be used in macros.

The tree structure is a mish-mash of types organized in various collections. These collections frequently have more collections nested within them. It is quite complex.

I wrote a macro to help search for values within the printer tree (Example: Search Printer Objects) since I can never remember the exact names of anything.

You can see by the output of it, the `printer` object carries booleans, floats, strings, tuples, and even a list of lists.

```
printer.firmware_retraction.retract_length                : 0.75
printer.firmware_retraction.unretract_extra_length        : 0.0
printer.firmware_retraction.unretract_speed               : 20.0
printer.firmware_retraction.retract_speed                 : 30.0
printer.probe
printer.probe.last_query                                              :
False
printer.bed_mesh
printer.bed_mesh.mesh_max                                             :
(275.0, 275.0)
printer.bed_mesh.profile_name                                         :
default
printer.bed_mesh.mesh_min                                             :
(25.0, 25.0)
printer.bed_mesh.probed_matrix :
       [[-0.16875, -0.16375, -0.151875, -0.165625, -0.168125],
       [-0.1125, 0.04125, -0.063125, -0.05, -0.13875],
       [0.020625, -0.015, -0.009375, -0.04625, -0.013125],
       [0.023125, 0.095625, 0.0475, 0.165625, -0.041875],
       [0.070625, 0.06375, 0.043125, 0.01625, -0.03375]]
```

The `printer` object is the cornerstone to writing useful macros because of the data contained within it.

Say we don't really want to have put that default pressure advance number in there since it is already defined on the `[extruder]`. We can reference that value from within our macro like so.

```
[gcode_macro start_print]
       gcode:
              {% if not params.PA %}
                     {% set PA =
printer.configfile.settings.extruder.pressure_advance %}
              {% endif %}

              M109 S{params.TOOL_TEMP}            # Heat the tool to
temperature and wait

              {% if printer.homed_axes != 'XYZ' %}
                     G28
       #Home All Axes
              {% endif %}

              SET_PRESSURE_ADVANCE ADVANCE={PA}

              G92 E0                                                # Reset
Extruder
```

```
              G1 Z2.0 F3000                                # Move Z Axis to
travel height
              G1 X0.1 Y20 Z0.2 F5000.0            # Move to start
position
              G1 X0.1 Y200.0 Z0.2 F1500.0 E15 # Draw the first line
              G1 X0.4 Y200.0 Z0.2 F5000.0  # Move to side a little
              G1 X0.4 Y20 Z0.3 F1500.0 E30        # Draw the second line
              G92 E0                                                # Reset
Extruder
              G1 Z2.0 F3000                                # Move Z Axis up
to travel height
```

It should be noted, the `configfile` tree contains both `config` and `settings` branches. Both trees will have all the values that were read from the `config` file the last time klipper read it from the disk. The `settings` tree will have all of the currently configured values as *typed* values where the `config` tree will have them as `strings` only.

It should also be noted that you cannot change any of the values found in the `printer` object. The only way to change anything is if there is an available gcode to do so.

Our start print macro is going to start looking cluttered pretty quickly, especially if we start doing comparisons and math on various values obtained from the printer object. One thing we can do to help with this is to assign the printer object or a specific section of it to a local variable that can then be referenced instead. This can be done anywhere in the macro prior to its first use, but to keep things concise, I personally prefer to put it at the top.

```
[gcode_macro start_print]
       gcode:
              {% set config = printer.configfile.settings %}

              {% if not params.PA %}
                     {% set PA = config.extruder.pressure_advance %}
              {% endif %}
```

This is especially helpful when macros begin to get more complex like when comparing the position of the toolhead to the axis maximum

```
       {% if printer.toolhead.position.z > (
printer.toolhead.axis_maximum.z - 40 ) %}
```

This could be made a bit tidier by assigning the position and axis limit values to a variable and then evaluating them.

```
[gcode_macro end_print]
```

```
gcode:
       {% set axismax = printer.toolhead.axis_maximum %}
       {% set pos     = printer.toolhead.position     %}

       #Move toolhead away from finished print
       {% if pos.z <= ( axismax.z - 40 ) %}
              G1 X10 Y10 Z{ pos.z + 40 }
       {% else %}
              G1 X10 Y10 Z{ axismax.z }
       {% endif %}
```