

# GOTHIC

## VIRTUAL TABLETOP

### **V 6.9 "The Macro Update"**

#### Compendium and Macro Reference

|                               |    |
|-------------------------------|----|
| <u>Compendium</u>             | 2  |
| <u>Weapons</u>                | 3  |
| <u>Armour</u>                 | 4  |
| <u>Items</u>                  | 5  |
| <u>Spells</u>                 | 6  |
| <u>Abilities</u>              | 7  |
| <u>Tables</u>                 | 8  |
| <u>Custom Categories</u>      | 9  |
| <u>Campaign Properties</u>    | 10 |
| <u>Macros</u>                 | 11 |
| <u>Macro Script Editor</u>    | 15 |
| <u>Functions</u>              | 16 |
| <u>Script Local Variables</u> | 26 |
| <u>Token Variables</u>        | 27 |
| <u>BBcode / RichText</u>      | 28 |

# Compendium: General Information

Compendium campaign files (weapons.json, armour.json, spells.json, etc.) are defined using [JSON](#).

Your compendium files can be found in your campaign's folder:

**Windows:**

```
%APPDATA%\Godot\app_userdata\Gothic_VTT\local\<CAMPAIGN_NAME>
```

**Linux :**

```
~/.local/share/godot/app_userdata/Gothic_VTT/local/<CAMPAIGN_NAME>
```

You can create a .json file in any text editor (notepad, notepad++, kate, emacs, vim, etc.)

Default compendium categories: [abilities](#), [armour](#), [items](#), [spells](#), [tables](#), and [weapons](#).

**Note:** To add more categories and enable/disable compendium categories, see the [Campaign Properties](#) section.

Terms to be familiar with:

## Dictionaries – {}

Dictionaries are comma-separated key:value pairs. Keys should be strings, and values can be strings or numbers.

## Strings – “”

Strings are any character you can type on your keyboard surrounded by double quotes.

- “this is a string ”
- “TH15 15 4L50 4 5TR1N6!!!@#\*#()\$@#\$@!”
- “69” <= that is a string
- not a string

## Numbers

- 3
- -45
- 420.69
- “4” <= that is a string, not a number

## Arrays – []

Arrays are comma-separated lists of Dictionaries, Strings, Numbers, or Arrays.

# Weapons

The weapons.json campaign file is where you can define the rpg system's default weapons and all of their properties. These weapons can then be added to tokens (via the character sheet > weapons tab, or the compendium > armour tab) and then edited if desired.

## For those familiar with .json files:

The weapons.json file should contain an array of dictionaries, with a separate dictionary for each weapon. Keys should be strings, and values can be strings or numbers.

## For those unfamiliar with .json files:

Below is what a 'weapons.json' file could look like:

```
[
  {
    "name": "Shortsword",
    "damage": "1d6",
    "damage type": "Piercing",
    "range": "5ft",
    "properties": "finesse, light",
    "price in gold": 5,
  },
]
```

Inside each pair of curly brackets '{}' should be a comma separated list of keys and values. Keys should be strings, and values can be strings or numbers.

Keys and their corresponding values should be separated by a colons ':'

In the example above, "name" is the key, and "Shortsword" is the value. Keys should always have a value attached to them. If you want the value to be *nothing*, use an 'empty string,' (a pair of quotes "" with nothing inside).

**Note:** Each dictionary must contain a "name" key, and it's a good idea for each dictionary to contain the exact same keys.

**Note:** If your weapon dictionary contains any of the following keys: script, text\_color, or base\_color, then adding the weapon to a token will automatically create a corresponding macro.

# Armour

The armour.json campaign file is where you can define an rpg system's default armour and all of their properties. This armour can then be added to tokens (via the character sheet > armour tab, or the compendium > armour tab) and then edited if desired.

## For those familiar with .json files:

The armour.json file should contain an array of dictionaries, with a separate dictionary for each piece of armour. Keys should be strings, and values can be strings or numbers.

## For those unfamiliar with .json files:

Below is what a 'armour.json' file could look like:

```
[
  {
    "name": "Plate",
    "AC": 18,
    "requirements" : "STR 15",
    "stealth": "Disadvantage",
  },
]
```

Inside each pair of curly brackets '{}' should be a comma separated list of keys and values. Keys should be strings, and values can be strings or numbers.

Keys and their corresponding values should be separated by a colons ':'

In the example above, "name" is a key, and "Plate" is it's corresponding value. Keys should always have a value attached to them. If you want the value to be *nothing*, use an 'empty string,' (a pair of quotes "" with nothing inside). The 'damage' key has 'nothing' as its value.

**Note:** Each dictionary must contain a "name" key, and it's a good idea for each dictionary to contain the exact same keys.

# Items

The items.json campaign file is where you can define an rpg system's default item list. Items can then be added to tokens via the items tab in the compendium.

## For those familiar with .json files:

The items.json file should contain an array of dictionaries, with a separate dictionary for each item. Keys should be strings, and values can be strings or numbers.

## For those unfamiliar with .json files:

Below is what a 'items.json' file could look like:

```
[
  {
    "name": "Potion of Healing",
    "description": "Restores 2d4 + 2 health on use.",
    "weight": 0.5,
  },
]
```

Inside each pair of curly brackets '{}' should be a comma separated list of keys and values.

**Note:** An Item dictionary entry *must* contain a "name" and "description" key. "weight" is optional.

# Spells

The spells.json campaign file is where you can define an rpg system's default spell list. Spells can then be added to tokens (via the character sheet > spells tab, or the compendium > spells tab).

## For those familiar with .json files:

The spells.json file should contain an array of dictionaries, with a separate dictionary for each spell. Keys should be strings, and values can be strings or numbers.

## For those unfamiliar with .json files:

Below is what a 'spells.json' file could look like:

```
[
  {
    "name": "Fireball",
    "script" : "damage = roll(8, 6);
                show(damage);
                show(\"Dexterity save for half damage\");
                link_compendium(\"spells\", \"Fireball\");",
    "sort" : "Spells-300",
    "base_color" : "343434",
    "text_color" : "d4d4d4",
    "description" : "Room explodes, everyone dies.",
  },
]
```

Inside each pair of curly brackets '{} ' should be a comma separated list of keys and values.

**Note:** When adding a spell to a token, a macro is made automatically. Each spell dictionary must contain the following keys: "name", "script", and "description". The keys "sort", "text\_color", and "base\_color" can also be provided if desired. "sort" will determine what hotbar tab the spell macro will show up in, and the number following the hyphen determines how it will be sorted relative to other macros on that tab. text\_color and base\_color will change the color of the macro button, and can be any hexadecimal color, e.g. FFFFFFFF (white), 12C3F0 (whatever color this is).

The value of the 'description' key will appear in chat when running link\_compendium() on a spell. The 'description' field supports BBCode text formatting, see the [BBcode / Rich Text](#) section for more.

# Abilities

The abilities.json campaign file is where you can define an rpg system's default abilities, feats, traits, and other unique character features. Abilities can be added to tokens (via the character sheet > abilities tab, or the compendium > abilities tab).

## For those familiar with .json files:

The abilities.json file should contain an array of dictionaries, with a separate dictionary for each ability. Keys should be strings, and values can be strings or numbers.

## For those unfamiliar with .json files:

Below is what a 'abilities.json' file could look like:

```
[
  {
    "name":"Example Ability",
    "description":"Ability Description goes here.",
    "value":6,
    "max":9,
  },
]
```

**Note:** Each dictionary must contain the following keys: “name”, “description”, “value”, and “max”.

The keys “value” and “max” refer to the number of uses, and the maximum amount of uses of the ability a token has left. If these are not applicable to your system, leave them both as 0's.

# Tables

The tables.json campaign file is where you can define an rpg system's commonly used tables. Tables can be rolled on in macros, or in chat by using the `'/t'` or `'/table'` command followed by the name (or partial name) of the table.

For those familiar with .json files:

The tables.json file should contain a **dictionary** of dictionaries. The dictionary keys should be the name of the table.

For those unfamiliar with .json files, follow this example:

```
{
  "Male Names" :
  {
    "roll": "1d10",
    "1": "Mark",
    "2": "James",
    "3": "Jared",
    "4": "Evan",
    "5": "Liam",
    "6": "Brian",
    "7": "Matthew",
    "8": "Nick",
    "9": "Jordan",
    "10": "Preston",
  },
  "Female Names" :
  {
    "roll": "2d10 + 1",
    "3-4": "Agnes",
    "5-7": "Alice",
    "8-9": "Barbara",
    "10": "Beatrice",
    "11": "Calliope",
    "12-13": "Cora",
    "14": "Diane",
    "15": "Edith",
    "16": "Elizabeth",
    "17-21": "Florence",
  },
}
```

**Note:** Each table dictionary **must** contain a "roll" key defining what dice expression to roll. It should also contain one key/value pair for each possible roll result (first example table), or ranges that cover all possible results (second example table).

# Custom Compendium Categories

Adding a custom compendium category is as simple as creating a new new .json file and adding it to the [campaign\\_properties.json](#) compendium load order.

Some operating systems hide file extensions by default, so the easiest way to make sure you are creating a proper .json file is to just copy an existing compendium file (like 'weapons.json') and paste it in the same folder, renaming the file when prompted.

Custom compendium files should be an array [] of entries which are dictionaries {}.

The only required fields for an entry in a custom compendium category are "name" and "description". You can add any number of custom fields if you wish. When a custom compendium entry is linked in chat (from a macro), the entry's 'description' field will be displayed.

The 'description' field supports [BBcode / Rich Text](#).

Here is an example compendium file "boats.json":

```
[
  {"name": "Raft", "description": "..."},
  {"name": "Canoe", "description": "..."},
  {"name": "Brig", "description": "..."},
]
```

Here is what your 'campaign\_properties.json' file might look like if you want to view 'boats.json' in the in-game compendium:

```
{
  "default_map": "default_map",
  "init_sort": false,
  "compendium": [
    "boats.json",
    "weapons.json",
    "armour.json",
    "items.json",
    "spells.json",
    "abilities.json",
    "tables.json",
  ]
}
```

<----- NEW!

# Campaign Properties

In your campaign folder, you will find a file named 'campaign\_properties.json'. (To find your campaign folder, see the text highlighted in green in the [Compendium: General Information](#) section.)

```
{
  "default_map": "default_map",
  "init_sort": false,
  "compendium": [
    "weapons.json",
    "armour.json",
    "items.json",
    "spells.json",
    "abilities.json",
    "tables.json",
  ]
}
```

## “default\_map”

This is the name of the map you want to load when you first load up a campaign. This can be changed in-game by navigating to the 'Maps' tab of the left panel, then clicking on the gray star icon in the bottom left of the map you want to set as default. The current default map will have a gold star.

## “init\_sort”

This controls whether the initiative tracker will sort from low to high, or high to low by default. You can change this in-game by opening the initiative tracker and clicking 'Highest' or 'Lowest'.

## “compendium”

- This is where you enable/ disable compendium categories from appearing in the in-game compendium, change the order of compendium tabs, or add custom compendium categories.
- To add a custom category, see [Custom Compendium Categories](#).
- To change the order in which the categories appear in the in-game compendium by moving the lines around. In this example, the in-game compendium tabs would be sorted (from left to right) weapons, armour, items, spells, abilities, tables. To put the 'tables' tab first and the 'weapons' tab last, swap “weapons.json”, and “tables.json”, in the list.

# Macros

[Function List](#)

[Macro Script Local Variables](#)

[Token Variables](#)

[If-Else Blocks](#)

The image shows a macro editor interface with the following elements and numbered callouts:

- 1**: Name field containing "Example Macro".
- 2**: Sort field containing "Example-0".
- 3**: Button color preview field containing "Button Preview".
- 4**: Base color field containing "770000".
- 5**: Text color field containing "00ff00".
- 6**: "EDIT MACRO" button.
- 7**: "X" button to delete the macro.

## [1. Macro Name](#)

This is the name of your macro, and the text that will be shown on the generated button on that token's hotbar.

## [2. Macro Sort Name](#)

This controls what tab your macro will appear in on the hotbar, and in what order.

## [3. Macro button color preview](#)

Displays what the macro button will look like on the token's hotbar.

## [4. Base Color](#)

The base color of the generated macro button on the token's hotbar. Colors should be in hexadecimal. If no color is provided, a default color is used.

## [5. Text Color](#)

Text color of the generated macro button on the token's hotbar. Colors should be in hexadecimal. If no color is provided, a default color is used.

## [6. Macro Editor](#)

Opens a text box for you to type macro scripts into.

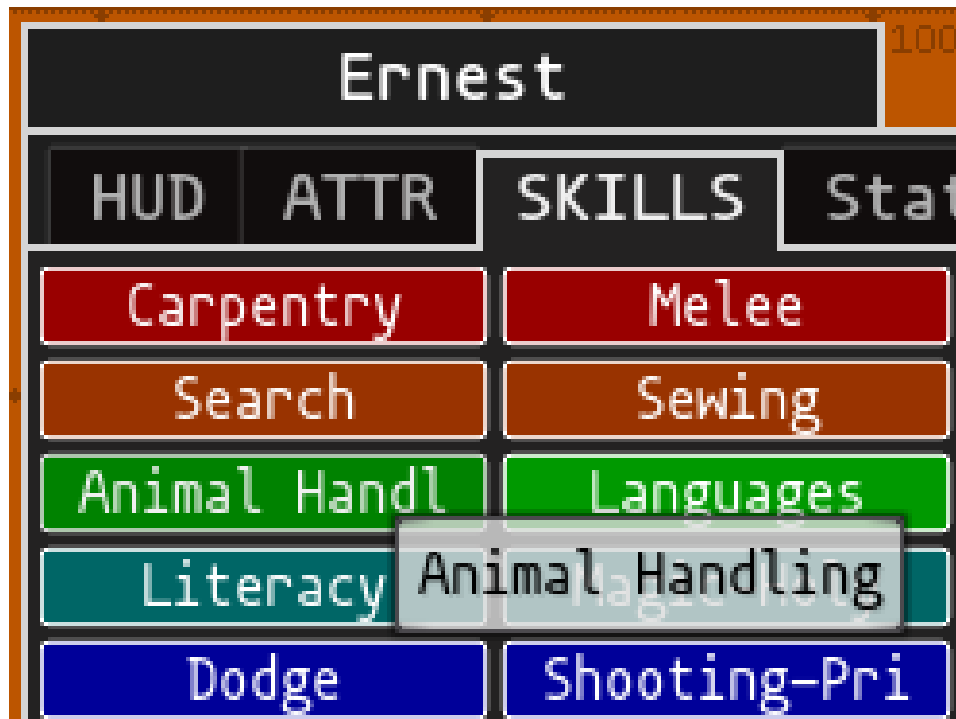
## [7. Delete](#)

Deletes the macro.

## Macro Name

This can be anything, but ideally it should be a short word or description of what the macro does.

Long macro names will be cut off on the hotbar, but hovering your cursor over the macro button will always show the full macro name.



# Macro Sort Name

Text in this box will be used as the name of a new tab on the token's hotbar. If multiple macros share the same Macro Sort Name, they will appear in the same tab, sorted alphabetically by [Macro Name](#). Alternatively, you can append a hyphen (-) followed by any whole number to manually sort macros within individual tabs. Macros with small numbers following the hyphen will appear in the top rows, and the largest numbers will be in the final rows.

*Note: If multiple macros share the same sort name and number, they will all appear next to eachother, sorted alphabetically.*

Consider the two macro names and sort values below:

NameAlphaSortTestx

NameBravoSortTestx

Example Token

HUDTestStatuses

AlphaBravo

They will appear on the hotbar sorted alphabetically (right):

If we wanted them to appear in the opposite order we could add a hyphen followed by a sort number to their sort fields like this:

NameAlphaSortTest-420x

NameBravoSortTest-69x

Example Token

HUDTestStatuses

BravoAlpha

Resulting in macro 'Bravo' (with the smallest number) appearing first, followed by macro 'Alpha':

## Macos Base + Text Color

Colors are described using hex codes. Hex codes are made up of 6 hexadecimal (0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f) values. The first two control the amount of red in the color, the next two the amount of green, and the last two the amount of blue. There are numerous hex color pickers that can be found with a simple internet search.

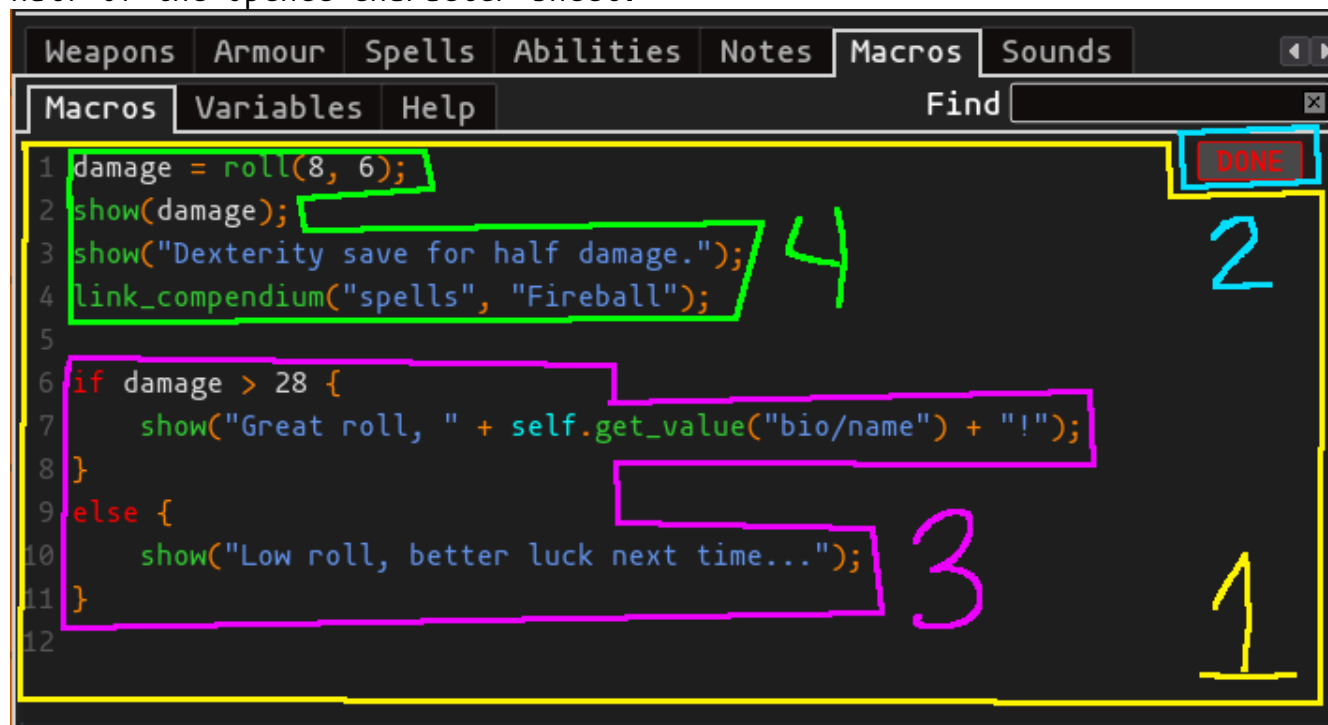
The **Text Color** changes the button's text and border color.

The **Base Color** changes the button's main color.



# Macro Script Editor

After clicking 'Edit Macro', a large text field will replace the bottom half of the opened character sheet.



1. The area where you will type macros.

2. This button closes the script editor. Any changes you've made to the macro script will remain, but are still unsaved.<sup>A</sup> To save all changes to macros, click the floppy disc button on the top right of the character sheet.

3. [If-Else blocks](#)

4. [Macro Functions](#)

## Notes:

- All lines should end with a semicolon ';'. (the exception being lines 6, 8, 9, and 11 in the above example. If <comparison> {} and else {} do not need semicolons after them, but any line inside curly brackets '{} ' does.)
- Indentation is not required anywhere, including If-Else blocks.

A. You can also close the script editor by clicking the blue floppy disc icon, which will immediately save the character sheet as well.

# Macro Functions

|                                           |                                                |
|-------------------------------------------|------------------------------------------------|
| <a href="#"><u>abs</u></a>                | <a href="#"><u>max</u></a>                     |
| <a href="#"><u>.add_to_initiative</u></a> | <a href="#"><u>min</u></a>                     |
| <a href="#"><u>.add_to_token</u></a>      | <a href="#"><u>pick_token</u></a>              |
| <a href="#"><u>ceil</u></a>               | <a href="#"><u>.play_music</u></a>             |
| <a href="#"><u>evaluate_roll</u></a>      | <a href="#"><u>.play_sound</u></a>             |
| <a href="#"><u>.execute_macro</u></a>     | <a href="#"><u>.remove_from_initiative</u></a> |
| <a href="#"><u>floor</u></a>              | <a href="#"><u>roll</u></a>                    |
| <a href="#"><u>get_digit</u></a>          | <a href="#"><u>roll_table</u></a>              |
| <a href="#"><u>.get_input</u></a>         | <a href="#"><u>round</u></a>                   |
| <a href="#"><u>.get_value</u></a>         | <a href="#"><u>.set_aura</u></a>               |
| <a href="#"><u>is_valid_variable</u></a>  | <a href="#"><u>.set_status</u></a>             |
| <a href="#"><u>link_compendium</u></a>    | <a href="#"><u>.set_value</u></a>              |
| <a href="#"><u>link_macro</u></a>         | <a href="#"><u>show</u></a>                    |
| <a href="#"><u>link_roll</u></a>          | <a href="#"><u>.stop_sound</u></a>             |
| <a href="#"><u>link_table</u></a>         |                                                |

## 'Dot Functions' & 'self'

Certain function names are preceeded by a period, and highlighted in green. These are called 'Dot Functions,' and they're special. In order to be used in a macro script, these functions need to be attached to a [Token Variable](#).

One example of a token variable is the variable '[self](#)'. [self](#) is a token variable that refers to the token whose character sheet the macro is on.

When you want to use a dot function, you attach it to a Token Variable like this:

```
self.add_to_initiative(16);  
  
self.set_value("bio/name", "Fred Flintstone");  
  
show(self.get_value("attributes/strength/value"));
```

This first line would add the token running the macro ([self](#)) to initiative with a value of 16, and the second line would set that same token's name to "Fred Flintstone". The third line will display the token's name (which is now 'Fred Flintstone') in chat.

See the [Token Variable](#) section for more information.

## `abs(x)`

Returns value of *x* without regard to its sign.

- `abs(2.1) => 2.1`
  - `abs(-2.1) => 2.1`
- 

## `.add_to_initiative(x)`

Must immediately follow a [Token Variable](#).

Adds the token (the one represented by the token variable) to the initiative list, where 'x' should be the token's speed, initiative roll, or equivalent.

- `self.add_to_initiative(12);`
  - `self.add_to_initiative(roll(1, 20, "keep_highest", 1));`
  - ~~`add_to_initiative(12);`~~
- 

## `.add_to_token("category", "entry_name")`

Must immediately follow a [Token Variable](#).

Adds the specified campaign compendium entry to the token.

"category" should be replaced with one of the following:

```
"weapons"  
"armour"  
"spells"  
"abilities"  
"items"
```

"entry\_name" should be a valid entry in the category.

- `self.add_to_token("weapons", "M16");`
- `self.add_to_token("items", "Health Potion");`
- ~~`add_to_token("spells", "Fireball");`~~

## ceil(x)

Returns the smallest integer greater than or equal to x.

- `ceil(2.1) => 3`
  - `ceil(67.000000001) => 68`
  - `ceil(-2.1) => -2`
- 

## evaluate\_roll(`roll_var`, `"roll_option"`, `arg1`, ... , `argn`)

This function allows you to evaluate any roll (stored in a macro local variable) using most of the roll functions. The two roll functions you cannot use are 'explode' or 'reroll'

`roll_var` should be a macro local variable storing the result of a dice roll.

`"roll_option"` should be a valid roll option (not 'explode' or 'reroll')

`arg1` to `argn` should be appropriate arguments for the used roll option.

---

## .execute\_macro(`"macro_name"`, `arg1`, ... , `argn`)

Must immediately follow a Token Variable.

This function allows you to run another macro from any macro.

`"macro_name"` should be the name of a macro function that exists on the token.

You may optionally pass any number of arguments following the macro name that will automatically stored in the macro script variables `"arg0"` through `"arg(n-1)"`. See Macro Script Local Variables for more

- `self.execute_macro("Fireball");`
  - `execute_macro("Fireball");`
- 

## floor(x)

Returns the greatest integer less than or equal to x.

- `floor(2.1) => 2`
- `floor(67.000000001) => 67`
- `floor(-2.1) => -3`

## get\_digit(*number*, *place*)

*number* should be either a macro local variable or any specified number.

*place* should be the location of the desired digit, e.g. 1 for ones digit, 10 for the 10's digit, and so on.

- get\_digit(34, 1) => 4
  - get\_digit(34, 10) => 3
  - get\_digit(34, 100) => 0
- 

## .get\_input(*"prompt"*, *"prefix"*, *"suffix"*)

Must immediately follow a [Token Variable](#).

This function creates a prompt on the hotbar allowing the user to provide input to a macro at runtime. The user can type into a text box, or choose an option from a drop-down menu.

*"prompt"* should be a string containing a set of square brackets '[]'.

The brackets should either be:

- Empty: "[]"
- Contain a default value: "[3]"
- Contain a list of values: "[strength, dexterity, intelligence]"

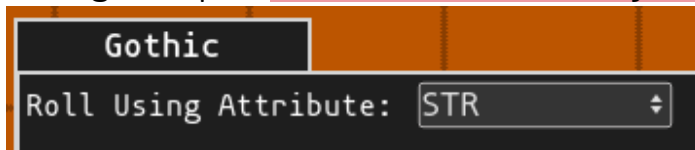
Note:

By default, any input received in this way is evaluated as if it were a numerical value, or script variable. If you are getting an input that should be a string such as a token name, prefix the open square bracket with 'str', see the last example below.

*"prefix"* is an optional argument that represents a string to prepend to the supplied information.

*"suffix"* is an optional argument that represents a string to append to the supplied information.

- self.get\_input("Take Damage: []")
- self.get\_input("Use Attribute: [STR, DEX, CON, INT, WIS, CHA]")



- self.get\_input("New Name: str[]", "Sir ", ", Esq.")  
OUTPUT => "Sir <INPUT>, Esq."

## `.get_value("path/to/value")`

Must immediately follow a [Token Variable](#). Retrieves a value from the token's character sheet.

"path/to/value" must be surrounded by quotes. A list of valid paths can be found on the Macros' "Help" tab. You can copy a path to your clipboard by clicking an item in the list.

- `self.get_value("bio/name");` => Gets the token's name.
  - `self.get_value("attributes/strength/value")` => 16
  - `get_value("bio/name")`
- 

## `is_valid_variable(variable)`

Returns 'true' if the variable has a value assigned to it in the current macro script. Can be used to check if runtime args (e.g. `arg0`) is present.

`Variable` should be the name of a local or runtime variable.

---

## `link_compendium("category", "entry", "table_roll")`

Posts a hyperlink in chat that, when clicked, displays the contents of the corresponding compendium entry's "description" field. If linking a table, you supply the roll value to print the table result mapped to that roll.

- `link_compendium("bestiary", "goblin");` => "A tiny evil creature."
  - `link_compendium("tables", "dungeon loot", 17);` => 1d6 gems
- 

## `link_macro(token_reference, "macro_name", args)`

Posts a hyperlink in chat that, when clicked, runs the a macro on the supplied token.

`token_reference` should be a valid [Token Variable](#), such as `self`.

"macro\_name" should be in quotes ("), and should be the name of a macro the token has.

`args` can be a comma-separated list of macro runtime arguments.

Used in concert with [is\\_valid\\_variable\(\)](#)

- `link_macro(self, "Fireball");`
- `link_macro(self, "Take Damage", 5, "goblin spear");`

**link\_roll("visible\_text", "xdy + z")**

*Posts a hyperlink in chat that when clicked rolls the supplied dice expression. The expression can be any you would send in chat via a '/roll' command.*

"visible\_text" can be anything, such as "Click to Roll Damage". This is the hyperlink that you will see in chat.

"xdy + z" can be any /roll command, such as "2d6 + 4".

- link\_roll("Roll Damage", "(2d8 - 4) \* 3");
- link\_roll("Ambush! Click for Number of Goblins", "10d10");
- link\_roll("Roll Damage", "1d6 +" + self.STR);

---

**link\_table("visible\_text", "table\_name")**

*Posts a hyperlink in chat that when clicked rolls on the table specified by "table\_name".*

"visible\_text" can be anything, such as "Roll Dungeon Loot Table". This is the hyperlink that you will see in chat.

"table\_name" is the name of any table from the compendium.

- link\_table("Roll For Loot", "dungeon loot table");
- link\_table("Random Name: Male", "male names");

---

**max(a, b)**

*Returns the larger of two values.*

- max(3, 8) => 8

---

**min(a, b)**

*Returns the smaller of two values.*

- min(3, 8) => 3

## `pick_token()`

Allows for the creation of [Token Variables](#). Running a macro with `pick_token()` in it will interrupt the macro to allow you to choose any token on a map for you to manipulate.

In the following examples, the words in blue are [Token Variables](#), allowing you to run [Dot Functions](#) on them.

*Note: Using `pick_token()` in any way other than the way outlined below will most likely lead to unwanted macro behavior.*

- `target_token = pick_token();`
  - `variable123 = pick_token();`
- 

## `.play_music("name_of_track")`

Must immediately follow a [Token Variable](#).

Plays the music track, with the sound emanating from the token's position. Used for creating on-map jukeboxes, radios, or ambient noise you want to appear to come from a specific location on the battle map.

- `self.play_music("Dragonborn.wav");`
  - `play_music("Dragonborn.wav");`
- 

## `.play_sound("name_of_track")`

Must immediately follow a [Token Variable](#).

Plays the sound file with the sound emanating from the token's position. Used for short sound files like battlecries when joining combat, gunshot sounds when attacking, screams when hurt, etc.

- `self.play_sound("gunshot_3.wav");`
- `play_sound("gunshot_3.wav");`

## `.remove_from_initiative()`

Must immediately follow a [Token Variable](#).

Removes the token from the initiative list.

- `self.remove_from_initiative();`
- `remove_from_initiative();`

---

## `roll(quantity, sides, "roll_option", arg1, ... argn)`

Rolls dice. Allows for the manipulation of dice rolls in several common ways.

`quantity` and `sides` should be any integer greater than 0. Quantity is how many dice you are rolling, and sides are how many sides they have. For rolling 3d6, 3 would be the quantity, and 6 would be the sides.

"roll\_option" can be any one of the following:

### `"explode"`

Any dice that roll their highest value are first added to the total, then rerolled.

`arg1` (optional) is the maximum number of rerolls allowed.

### `"keep_highest"`

Keeps `arg1` highest rolls out of the quantity rolled.

### `"keep_middle"`

Keeps `arg1` median rolls out of the quantity rolled.

### `"keep_lowest"`

Keeps `arg1` lowest rolls out of the quantity rolled.

### `"count"`

Counts occurrences of `arg1` through `argn` over all the rolled dice.

### `"count_over"`

Counts number of rolls over `arg1`.

### `"count_under"`

Counts number of rolls under `arg1`.

### `"reroll"`

Rerolls any rolls listed in `arg1` through `argn`. Gives up if the task is impossible or extremely unlikely.

- |                                                |               |      |
|------------------------------------------------|---------------|------|
| • <code>roll(4, 6);</code>                     | => 3, 2, 5, 1 | (11) |
| • <code>roll(1, 4, "explode");</code>          | => 4, 4, 1    | (9)  |
| • <code>roll(2, 20, "keep_highest", 1);</code> | => 17, 8      | (17) |
| • <code>roll(2, 20, "keep_lowest", 1);</code>  | => 17, 8      | (8)  |
| • <code>roll(4, 4, "count", 4);</code>         | => 1, 1, 2, 3 | (0)  |
| • <code>roll(4, 4, "count_over", 2);</code>    | => 1, 1, 2, 3 | (1)  |
| • <code>roll(4, 4, "count_under", 2);</code>   | => 1, 1, 2, 3 | (2)  |

## `roll_table("table_name", times)`

Rolls on the specified table in the Campaign Compendium. Optionally you can provide a number as the second argument which allows you to roll multiple times.

"table\_name" should be in quotes, and should be the name of a table in the compendium.

times is optional, and should be an integer greater than 0.

- `roll_table("male names") => Tom`
  - `roll_table("male names", 3) => Tom, Jerry, Spike`
- 

## `round(x)`

Returns x rounded to the nearest integer.

- `round(3.9) => 4`
  - `round(3.2) => 3`
- 

## `.set_aura("a/b/c", size, "color")`

Must immediately follow a [Token Variable](#).

"a/b/c" should be one of a, b, or c, in quotes.

size should be any number greater than zero.

"color" should be one of the following:

"red", "orange", "yellow", "green", "blue", "indigo", or "violet"

- `self.set_aura("a", 3, "yellow");`
- `self.set_aura("b", 4.5, "orange");`
- `set_aura("a", 3, "red");`

**.set\_status("status\_name", true/false)**

Must immediately follow a [Token Variable](#).

This is used to set the visibility of different status icons on a token. Token statuses can be toggled manually via the hotbar.

"status\_name" should be replaced with one of the following statuses:

|            |    |                        |
|------------|----|------------------------|
| "dead"     | -> | gravestone over token  |
| "blind"    | -> | crossed out eye        |
| "deaf"     | -> | crossed out ear        |
| "mute"     | -> | crossed out microphone |
| "poisoned" | -> | skull and crossbones   |
| "red"      | -> | red dot                |
| "orange"   | -> | orange dot             |
| "yellow"   | -> | yellow dot             |
| "green"    | -> | green dot              |
| "blue"     | -> | blue dot               |
| "indigo"   | -> | indigo dot             |
| "violet"   | -> | violet dot             |

- `self.set_status("blind", true);`    => turns on blind status icon
- `self.set_status("red", false);`    => turns off red status icon
- `set_status("deaf", false);`

---

**.set\_value("path/to/value" , new\_value)**

Must immediately follow a [Token Variable](#).

This can be used to change (almost) any value on a token's Character Sheet.

"path/to/value" must be surrounded by quotes. A list of valid paths can be found on the Macros' "Help" tab, you can copy a path to your clipboard by clicking an item in the list.

- `self.set_value("bio/name", "Fred Flintstone");`
- `set_value("bio/name", "Fred Flintstone");`

## show(\*)

*Intelligently\* prints things to chat.*

Anything inside the parentheses will appear in chat verbatim, with some exceptions. Any variables will be evaluated, and all strings and values will be concatenated, with any plus signs '+' being ignored.

If the only argument is a variable storing a roll() function result, it will appear in chat formatted in a box, and the roll tooltip will show the roll() command, and individual die rolls.

```
show("Hello, my name is ", self.get_value("bio/name") + "!");
```

```
Mr. Wallaby: Gothic runs Greeting:  
Hello, my name is Gothic!
```

```
result = roll(3,6);  
show("I rolled " + result + "!");
```

```
Mr. Wallaby: Gothic runs roll:  
I rolled 12!
```

```
result = roll(4, 4, "explode");  
show(result);
```

```
Mr. Wallaby: Gothic runs roll:
```



Hover:

|               |               |
|---------------|---------------|
| 4d4, explode: | 1, 3, 2, 4, 1 |
| 4d4, explode  | 11            |

*\*It's not that smart.*

---

## .stop\_sound()

Must immediately follow a [Token Variable](#).

Stops any currently playing sounds/music on a token.

- `self.stop_sound();`
- `stop_sound();`

# Macro Script Local Variables

Local script variables can be any combination of the following characters: (A-Z, a-z, '\_', '-', 0-9)

*Variable names should start with at least one letter (a-Z). See the examples below for valid variable names.*

A local variable is any word that precedes a single equal sign '='. You can assign different types of things to local variables including:

- Numbers/ result of arithmetic.
- Strings ("text wrapped in double quotes", meant to be displayed or evaluated as written)
- boolean values (true, false) as a result of comparisons (<, >, <=, >=, ==, !=)
- The result of certain macro functions [roll(), pick\_token(), .get\_input(), .get\_value()]

In the example script below, words highlighted in **green** are local variable declarations (you can tell because they have a single equal sign '=' after them). **Green** words are currently having a value assigned to them. Words in **salmon** are local variables that are being evaluated, or in other words, the value that the variable holds is being accessed or revealed.

```
target = pick_token();
to_hit = roll(1, 20);
success = to_hit >= target.get_value("hud/AC/value");
if success {
    show("Hit!");
    damage = roll(2, 8);
    damage2 = roll(1, 4);
    show("Sword deals " + damage + " slashing,");
    show("and " + damage2 + " fire damage!");
}
else {
    show("Miss");
}
```

**target** is an example of a [Token Variable](#).

# Token Variables

Token Variables are [Local Variables](#) that hold a reference to a token. They are made by assigning the result of [pick\\_token\(\)](#) to an available local script variable.

Token Variables can be used to run a [Dot Function](#). Common uses include:

- Getting an enemy token's AC
- Dealing damage to or healing a specific token

'[self](#)' is a special token variable that cannot be overwritten. It *always* represents the token that owns the running macro.

## If-Else Blocks

If-Else blocks can be used to execute macro functions if certain conditions are met.

If-Else blocks look like this:

```
if condition {...}
else {...}
```

... can be replaced with any number of valid macro script statements, including if-else blocks. Lines inside of curly brackets '{}' should behave all other macro script rules, such as being ended with semicolons ';'.

*condition* should be replaced either with a local variable holding a boolean (*true* or *false*) value, or a comparison statement using one of the following [comparison](#) operators:

| Comparison | If block will be evaluated if: |
|------------|--------------------------------|
|------------|--------------------------------|

|                             |                                                           |
|-----------------------------|-----------------------------------------------------------|
| <code>a &lt; b</code>       | <code>a</code> is less than <code>b</code>                |
| <code>a &gt; b</code>       | <code>a</code> is greater than <code>b</code>             |
| <code>a &lt;= b</code>      | <code>a</code> is less than or equal to <code>b</code>    |
| <code>a &gt;= b</code>      | <code>a</code> is greater than or equal to <code>b</code> |
| <code>a == b</code>         | <code>a</code> is equal to <code>b</code>                 |
| <code>a != b</code>         | <code>a</code> is not equal to <code>b</code>             |
| <code>local_variable</code> | <code>local_variable</code> is <i>true</i>                |

If the comparison (or the local variable) is not *true*, the else block will be executed if one is provided.

# BBcode / Rich Text

The following table shows supported tags in compendium entry 'description' fields, as well as in macro show() output.

*Note: Bbcode tags must be closed in a 'last in, first out' order. This means the last set of tags you open should be closed first. See the last two examples below.*

| Input                                           | Output                      |
|-------------------------------------------------|-----------------------------|
| <code>[b]Test![/b]</code>                       | <b>Test!</b>                |
| <code>[u]Test![/u]</code>                       | <u>Test!</u>                |
| <code>[i][u]Test![/u][/i]</code>                | <u><i>Test!</i></u>         |
| <code>[i][u]Test![/i][/u]</code>                | <i>Test!</i> <u>[/i]</u>    |
| <code>[color=#ff0000][u]test[/color][/u]</code> | <u>test</u> <u>[/color]</u> |

| Command       | Tag                                                  | Description                                                               |
|---------------|------------------------------------------------------|---------------------------------------------------------------------------|
| bold          | <code>[b]{text}[/b]</code>                           | Makes {text} bold.                                                        |
| italics       | <code>[i]{text}[/i]</code>                           | Makes {text} italics.                                                     |
| underline     | <code>[u]{text}[/u]</code>                           | Makes {text} underline.                                                   |
| strikethrough | <code>[s]{text}[/s]</code>                           | Makes {text} strikethrough.                                               |
| code          | <code>[code]{text}[/code]</code>                     | Makes {text} use the code font (which is typically monospace).            |
| center        | <code>[center]{text}[/center]</code>                 | Makes {text} horizontally centered.                                       |
| right         | <code>[right]{text}[/right]</code>                   | Makes {text} horizontally right-aligned.                                  |
| fill          | <code>[fill]{text}[/fill]</code>                     | Makes {text} fill the RichTextLabel's width.                              |
| indent        | <code>[indent]{text}[/indent]</code>                 | Increase the indentation level of {text}.                                 |
| color         | <code>[color=&lt;code/name&gt;]{text}[/color]</code> | Change {text} color; use name or # format, such as <code>#ff00ff</code> . |
| table         | <code>[table=&lt;number&gt;]{cells}[/table]</code>   | Creates a table with <number> of columns.                                 |
| cell          | <code>[cell]{text}[/cell]</code>                     | Adds cells with the {text} to the table.                                  |