bTagScript

JonSnowbd, PhenoM4n4n, Leg3ndary

Jul 14, 2022

USER GUIDES

Note: Check out User Guides if first!

CHAPTER

ONE

BLOCKS

1.1 All Block

class bTagScript.block.**AllBlock**

Bases: VerbRequiredBlock

The all block checks that all of the passed expressions are true. Multiple expressions can be passed to the parameter by splitting them with |.

The payload is a required message that must be split by |. If the expression evaluates true, then the message before the | is returned, else the message after is returned.

Usage: {all(<expression|expression|...>):<message>}

Aliases: and

Payload: message

Parameter: expression

Examples:

```
{all({args}>=100|{args}<=1000):You picked {args}.|You must provide a number between␣
↪100 and 1000.}
# if {args} is 52
You must provide a number between 100 and 1000.

# if {args} is 282
You picked 282.
```

1.2 Any Block

class bTagScript.block.**AnyBlock**

Bases: VerbRequiredBlock

The any block checks that any of the passed expressions are true. Multiple expressions can be passed to the parameter by splitting them with |.

The payload is a required message that must be split by |. If the expression evaluates true, then the message before the | is returned, else the message after is returned.

Usage: {any(<expression|expression|...>):<message>}

Aliases: or

Payload: `message`

Parameter: `expression`

Examples:

```
{any(hi=={args}|hello=={args}|heyy=={args}):Hello {user}!|How rude.}
If {args} is hi
Hello _Leg3ndary#0001!

If {args} is what's up!
How rude.
```

1.3 Assignment Block

1.4 Blacklist Block

class bTagScript.block.**BlacklistBlock**

Bases: `VerbRequiredBlock`

The blacklist block will attempt to convert the given parameter into a channel or role, using name or ID. If the user running the tag is in the targeted channel or has the targeted role, the tag will stop processing and it will send the response if one is given. Multiple role or channel requirements can be given, and should be split by a ",".

Usage: `{blacklist(<role,channel>):[response]}`

Payload: `response`

Parameter: `role, channel`

Examples:

```
{blacklist(Muted)}
{blacklist(#support):This tag is not allowed in #support.}
{blacklist(Tag Blacklist, 668713062186090506):You are blacklisted from using tags.}
```

1.5 Break Block

class bTagScript.block.**BreakBlock**

Bases: *Block*

The break block will force the tag output to only be the payload of this block, if the passed expresssion evaluates true. If no message is provided to the payload, the tag output will be empty.

This differs from the `StopBlock` as the stop block stops all tagscript processing and returns its message while the break block continues to process blocks. If command blocks exist after the break block, they will still execute.

Usage: `{break(<expression>):[message]}`

Aliases: `short, shortcircuit`

Payload: `message`

Parameter: `expression`

Examples:

```
{break(==){args}):You did not provide any input.}
```

1.6 Command Block

class bTagScript.block.**CommandBlock**(*limit: int = 3*)

Bases: VerbRequiredBlock

Run a command as if the tag invoker had ran it. Only 3 command blocks can be used in a tag.

Usage: {command:<command>}

Aliases: c, com, command

Payload: command

Parameter: None

Examples:

```
{c:ping}
# Invokes ping command

{c:ban {target(id)} Chatflood/spam}
# Invokes ban command on the pinged user with the reason as "Chatflood/spam"
```

1.7 Cooldown Block

class bTagScript.block.**CooldownBlock**

Bases: VerbRequiredBlock

The cooldown block implements cooldowns when running a tag. The parameter requires 2 values to be passed: rate and per integers. The rate is the number of times the tag can be used every per seconds.

The payload requires a key value, which is the key used to store the cooldown. A key should be any string that is unique. If a channel's ID is passed as a key, the tag's cooldown will be enforced on that channel. Running the tag in a separate channel would have a different cooldown with the same rate and per values.

The payload also has an optional message value, which is the message to be sent when the cooldown is exceeded. If no message is passed, the default message will be sent instead. The cooldown message supports 2 blocks: key and retry_after.

Usage: {cooldown(<rate>|<per>):<key>|[message]}

Payload: key, message

Parameter: rate, per

Examples:

```
{cooldown(1|10):{author(id)}}
the tag author used the tag more than once in 10 seconds
The bucket for 741074175875088424 has reached its cooldown. Retry in 3.25 seconds."
```

(continues on next page)

(continued from previous page)

```
{cooldown(3|3):{channel(id)}|Slow down! This tag can only be used 3 times per 3␣
↪seconds per channel. Try again in **{retry_after}** seconds."}
the tag was used more than 3 times in 3 seconds in a channel
Slow down! This tag can only be used 3 times per 3 seconds per channel. Try again␣
↪in **0.74** seconds.
```

1.8 Embed Block

class bTagScript.block.**EmbedBlock**

Bases: *Block*

An embed block will send an embed in the tag response. There are two ways to use the embed block, either by using properly formatted embed JSON from an embed generator or manually inputting the accepted embed attributes.

JSON

Using JSON to create an embed offers complete embed customization. Multiple embed generators are available online to visualize and generate embed JSON.

Usage: `{embed(<json>)}`

Payload: `None`

Parameter: `json`

```
{embed({"title":"Hello!", "description":"This is a test embed."})}
{embed({
    "title":"Here's a random duck!",
    "image":{"url":"https://random-d.uk/api/randomimg"},
    "color":15194415
})}
```

Manual

The following embed attributes can be set manually:

- `title`
- `description`
- `color`
- `url`
- `thumbnail`
- `image`
- `field` - (See below)

Adding a field to an embed requires the payload to be split by `|`, into either 2 or 3 parts. The first part is the name of the field, the second is the text of the field, and the third optionally specifies whether the field should be inline.

Usage: `{embed(<attribute>):<value>}`

Payload: `value`

Parameter: `attribute`

```
{embed(color):#37b2cb}
{embed(title):Rules}
{embed(description):Follow these rules to ensure a good experience in our server!}
{embed(field):Rule 1|Respect everyone you speak to.|false}
```

Both methods can be combined to create an embed in a tag. The following tagscript uses JSON to create an embed with fields and later set the embed title.

:: tagscript:

```
{embed({{"fields":[{"name":"Field 1","value":"field description","inline":false}]})}
{embed(title):my embed title}
```

1.9 If Block

class bTagScript.block.**IfBlock**

Bases: VerbRequiredBlock

The if block returns a message based on the passed expression to the parameter. An expression is represented by two values compared with an operator.

The payload is a required message that must be split by `|`. If the expression evaluates true, then the message before the `|` is returned, else the message after is returned.

Expression Operators:

Operator	Check	Example	Description
==	equality	a==a	value 1 is equal to value 2
!=	inequality	a!=b	value 1 is not equal to value 2
>	greater than	5>3	value 1 is greater than value 2
<	less than	4<8	value 1 is less than value 2
>=	greater than or equality	10>=10	value 1 is greater than or equal to value 2
<=	less than or equality	5<=6	value 1 is less than or equal to value 2

Usage: `{if(<expression>):<message>]}`

Payload: `message`

Parameter: `expression`

Examples:

```
{if(63=={args}):You guessed it! The number I was thinking of was 63!|Too {if({args}
↪<63):low|high}, try again.}
If args is 63
You guessed it! The number I was thinking of was 63!

If args is 73
Too low, try again.

If args is 14
Too high, try again.
```

1.10 Loose Variable Block

class bTagScript.block.**LooseVariableGetterBlock**

Bases: *Block*

The loose variable block represents the adapters for any seeded or defined variables. This variable implementation is considered "loose" since it checks whether the variable is valid during `process()`, rather than `will_accept()`. You may also define variables here with {$<variable name>:<value>}

Usage: `{<variable_name>([parameter]):[payload]}`

Aliases: This block is valid for any inputted declaration.

Payload: Depends on the variable's underlying adapter.

Parameter: Depends on the variable's underlying adapter.

Examples:

```
{=(example):This is my variable.}
{example}
This is my variable.

{$variablename:This is another variable.}
{variablename}
This is another variable.
```

1.11 Math Block

class bTagScript.block.**MathBlock**

Bases: *Block*

A math block is a block that contains a math expression. Will write out everything later bleh

Usage: `{math:<expression>}`

Aliases: `math, m, +, calc`

Payload: `expression`

Parameter: None

Examples:

```
{m:2+3}
5.0

{math:7(2+3)}
42.0

{math:trunc(7(2+3))}
42
```

1.12 Override Block

class bTagScript.block.**OverrideBlock**

Bases: *Block*

Override a command's permission requirements. This can override mod, admin, or general user permission requirements when running commands with the *Command Block*. Passing no parameter will default to overriding all permissions.

In order to add a tag with the override block, the tag author must have `Manage Server` permissions.

This will not override bot owner commands or command checks.

Usage: `{override(["admin"|"mod"|"permissions"]):[command]}`

Aliases: `bypass`

Payload: `command`

Parameter: `"admin", "mod", "permissions"`

Examples:

```
{override}
overrides all commands and permissions

{override(admin)}
overrides commands that require the admin role

{bypass(permissions)}
{bypass(mod)}
overrides commands that require the mod role or have user permission requirements
```

1.13 Random Block

class bTagScript.block.**RandomBlock**

Bases: `VerbRequiredBlock`

Pick a random item from a list of strings, split by either ~ or `,`. An optional seed can be provided to the parameter to always choose the same item when using that seed. You can weight options differently by adding a weight and | before the item.

Usage: `{random([seed]):<list>}`

Aliases: `#, rand`

Payload: `list`

Parameter: `seed`

Examples:

```
{random:Carl,Harold,Josh} attempts to pick the lock!
Possible Outputs:
Josh attempts to pick the lock!
Carl attempts to pick the lock!
Harold attempts to pick the lock!
```

(continues on next page)

(continued from previous page)

```
{=(insults):You're so ugly that you went to the salon and it took 3 hours just to␣
↪get an estimate.~I'll never forget the first time we met, although I'll keep␣
↪trying.~You look like a before picture.}
{=(insult):{#:{insults}}}
{insult}
Assigns a random insult to the insult variable

{#:5|Cool,3|Lame}
5 to 3 chances of being cool vs lame
```

1.14 Range Block

class bTagScript.block.**RangeBlock**

Bases: VerbRequiredBlock

The range block picks a random number from a range of numbers seperated by -. The number range is inclusive, so it can pick the starting/ending number as well. Using the rangef block will pick a number to the tenth decimal place.

An optional seed can be provided to the parameter to always choose the same item when using that seed.

Usage: {range([seed]):<lowest-highest>}

Aliases: rangef

Payload: number

Parameter: seed

Examples:

```
Your lucky number is {range:10-30}!
Your lucky number is 14!
Your lucky number is 25!

{=(height):{rangef:5-7}}
I am guessing your height is {height}ft.
I am guessing your height is 5.3ft.
```

1.15 Redirect Block

class bTagScript.block.**RedirectBlock**

Bases: VerbRequiredBlock

Redirects the tag response to either the given channel, the author's DMs, or uses a reply based on what is passed to the parameter.

Usage: {redirect(<"dm"|"reply"|channel>)}

Payload: None

Parameter: "dm", "reply", "channel"

Examples:

```
{redirect(dm)}
{redirect(reply)}
{redirect(#general)}
{redirect(626861902521434160)}
```

1.16 Replace Block

class bTagScript.block.**ReplaceBlock**

Bases: VerbRequiredBlock

The replace block will replace specific characters in a string. The parameter should split by a `,`, containing the characters to find before the command and the replacements after.

Usage: `{replace(<original,new>):<message>}`

Aliases: sub

Payload: message

Parameter: original, new

```
{replace(o,i):welcome to the server}
welcime ti the server

{replace(1,6):{args}}
if {args} is 1637812
6637862

{replace(, ):Test}
T e s t
```

1.17 Require Block

class bTagScript.block.**RequireBlock**

Bases: VerbRequiredBlock

The require block will attempt to convert the given parameter into a channel role or member, using name or ID. If the user running the tag is not in the targeted channel or doesn't have the targeted role, the tag will stop processing and it will send the response if one is given. Multiple role or channel requirements can be given, and should be split by a ",".

Usage: `{require(<role, channel, member>):[response]}`

Aliases: whitelist

Payload: response

Parameter: role, channel, member

Examples:

```
{require(Moderator)}
{require(#general, #bot-cmds):This tag can only be run in #general and #bot-cmds.}
{require(757425366209134764, 668713062186090506, 737961895356792882):You aren't␣
↪allowed to use this tag.}
```

1.18 ShortCutRedirect Block

1.19 STRF Block

class bTagScript.block.**StrfBlock**

Bases: *Block*

The strf block converts and formats timestamps based on strftime formatting spec. Two types of timestamps are supported: ISO and epoch. If a timestamp isn't passed, the current UTC time is used.

Invoking this block with `Unix Specific Services` will return the current Unix timestamp.

Usage: `{strf([timestamp]):<format>}`

Aliases: `unix`

Payload: `format`

Parameter: `timestamp`

Example:

```
{strf:%Y-%m-%d}
2021-07-11

{strf({user(timestamp)}):%c}
Fri Jun 29 21:10:28 2018

{strf(1420070400):%A %d, %B %Y}
Thursday 01, January 2015

{strf(2019-10-09T01:45:00.805000):%H:%M %d-%B-%Y}
01:45 09-October-2019

{unix}
1629182008
```

1.20 Strict Variable Block

class bTagScript.block.**StrictVariableGetterBlock**

Bases: *Block*

The strict variable block represents the adapters for any seeded or defined variables. This variable implementation is considered "strict" since it checks whether the variable is valid during `will_accept()` and is only processed if the declaration refers to a valid variable.

Usage: `{<variable_name>([parameter]):[payload]}`

Aliases: This block is valid for any variable name in *Response.variables*.

Payload: Depends on the variable's underlying adapter.

Parameter: Depends on the variable's underlying adapter.

Examples:

```
{=(example):This is my variable.}
{example}
This is my variable.
```

1.21 URL Encode Block

class bTagScript.block.**URLEncodeBlock**

Bases: VerbRequiredBlock

This block will encode a given string into a properly formatted url with non-url compliant characters replaced. Using + as the parameter will replace spaces with + rather than %20.

Usage: {urlencode(["+"]):<string>}

Payload: string

Parameter: "+", None

Example:

```
{urlencode:covid-19 sucks}
covid-19%20sucks

{urlencode(+):im stuck at home writing docs}
im+stuck+at+home+writing+docs

You can use this to search up blocks
Eg if {args} is command block

<https://btagscript.readthedocs.io/en/latest/search.html?q={urlencode(+):{args}}&
↪check_keywords=yes&area=default>
<https://btagscript.readthedocs.io/en/latest/search.html?q=command+block&check_
↪keywords=yes&area=default>
```

1.22 URL Decode Block

class bTagScript.block.**URLDecodeBlock**

Bases: VerbRequiredBlock

This block will decode a given url into a string with non-url compliant characters replaced. Using + as the parameter will replace spaces with + rather than %20.

Usage: {urldecode(["+"]):<string>}

Payload: string

Parameter: "+", None

Examples:

```
{urldecode:covid-19%20sucks}
covid-19 sucks

{urldecode(+):im+stuck+at+home+writing+docs}
im stuck at home writing docs
```

This block is just the reverse of the urlencode block

1.23 Length Block

class bTagScript.block.**LengthBlock**

Bases: VerbRequiredBlock

The length block will check the length of the given String. If a parameter is passed in, the block will check the length based on what you passed in, w for word, s for spaces. If you provide an invalid parameter, the block will return -1.

Usage: {length(["w", "s"]):<text>}

Aliases: len

Payload: text

Parameter: "w", "s"

```
{length:TagScript}
9

{len(w):Tag Script}
2

{len(s):Hello World, Tag, Script}
3

{len(space):Hello World, Tag, Script}
-1
```

1.24 Count Block

class bTagScript.block.**CountBlock**

Bases: VerbRequiredBlock

The count block will count how much of text is in message. This is case sensitive and will include substrings, if you don't provide a parameter, it will count the spaces in the message.

Usage: {count([text]):<message>}

Aliases: None

Payload: message

Parameter: text

```
{count(Tag):TagScript}
1

{count(Tag):Tag Script TagScript}
2

{count(t):Hello World, Tag, Script}
1 as there's only one lowercase t in the entire string
```

1.25 Comment Block

class bTagScript.block.**CommentBlock**

Bases: *Block*

The comment block is literally just for comments, it will not be parsed, however it will be removed from your codes output.

Usage: `{comment([other]):[text]}`

Aliases: /, Comment, comment, //

Payload: `text`

Parameter: `other`

```
{//:Comment!}

{Comment(Something):Comment!}
```

1.26 OrdinalAbbreviation Block

class bTagScript.block.**OrdinalAbbreviationBlock**

Bases: *Block*

The ordinalabbreviation block returns the ordinal abbreviation of a number. If a parameter is provided, it must be, one of, c, comma, indicator, i Comma being adding commas every 3 digits, indicator, meaning the ordinal indicator. (The st of 1st, nd of 2nd, etc.)

The number may be positive or negative, if the payload is invalid, -1 is returned.

Usage: `{ord(["c", "comma", "i", "indicator"]):<number>}`

Aliases: `None`

Payload: `number`

Parameter: `"c", "comma", "i", "indicator"`

```
{ord:1000}
1,000th

{ord(c):1213123}
1,213,123
```

(continues on next page)

(continued from previous page)

```
{ord(i):2022}
2022nd
```

1.27 Debug Block

class bTagScript.block.**DebugBlock**

Bases: *Block*

The debug block allows you to debug your tagscript quickly and easily, it will save the output to the debug_var key in the response dict. Separate the variables you want to include or exclude with a comma or a tilde.

If no parameters are provided in addition to no payload, all variables will be included. If no parameters are provided and a payload is provided, it will assume you want to include those variables.

Usage: `{debug(["i", "include", "e", "exclude"]):<variables>}`

Aliases: `None`

Payload: `variables`

Parameter: `"i", "include", "e", "exclude"`

Note: {debug} is the same as {debug(exclude):}

{debug:somevar~anothervar} is the same as {debug(include):somevar~anothervar}

Examples:

Note: THIS SHOULD ALWAYS BE PLACED AT THE VERY BOTTOM, IT WILL NOT RETURN ANYTHING UNDER IT.

```
Assuming we have the following tagscript, we first set the var something, then set
parsed (using the dollar sign method), to Hello|World, (assume we actually wanted␣
↪just the Hello
but we forgot)

{=(something):Hello/World}
{$parsed:{something(1)}}
{if({parsed}==Hello):Hello|Bye}

Running this would provided the output Bye, using the debug block below:
{debug}
We'll get all the variables at their, "final state"
This will be provided in a dict, which you can further parse and output to your␣
↪liking.

EG, in YAML format:
something: Hello/World
parsed: Hello/World
```

(continues on next page)

(continued from previous page)

```
This allow's you to see that you forgot to parse with a delimiter which will lead␣
↪to easy fixing.
```

CHAPTER

TWO

GETTING STARTED

Please refer to existing TagScript implementations such as the following Tags cog until developer documentation is written.

CHAPTER

THREE

THE INTERPRETER

CHAPTER

FOUR

BLOCKS AND ADAPTERS

CHAPTER

FIVE

INTERPRETER MODULE

5.1 Interpreter

class bTagScript.interpreter.**Interpreter**(*blocks: Union[List[*Block*], Tuple[*Block*]]*)

Bases: object

The TagScript interpreter.

blocks

A list or tuple of blocks to be used for TagScript processing.

Type
UnionList[*Block*]

process(*message: str*, *seed_variables: Optional[Dict[str,* Adapter*]] = None*, **, charlimit: Optional[int] = None*, ***kwargs*) → *Response*

Processes a given TagScript string.

Parameters

- **message** (*str*) – A TagScript string to be processed.
- **seed_variables** (*Dict[str, Adapter]*) – A dictionary containing strings to adapters to provide context variables for processing.
- **charlimit** (*int*) – The maximum characters to process.
- **kwargs** (*Dict[str, Any]*) – Additional keyword arguments that may be used by blocks during processing.

Returns
A response object containing the processed body, actions and variables.

Return type
Response

Raises

- ***TagScriptError*** – A block intentionally raised an exception, most likely due to invalid user input.
- ***WorkloadExceededError*** – Signifies the interpreter reached the character limit, if one was provided.
- ***ProcessError*** – An unexpected error occurred while processing blocks.

5.1.1 AsyncInterpreter

class bTagScript.interpreter.**AsyncInterpreter**(*blocks: Union[List[Block], Tuple[Block]]*)

Bases: *Interpreter*

An asynchronous subclass of *Interpreter* that allows blocks to implement asynchronous methods. Synchronous blocks are still supported. This subclass has no additional attributes from the *Interpreter* class. See *Interpreter* for full documentation.

async process(*message: str*, *seed_variables: Optional[Dict[str, Adapter]] = None*, ***, *charlimit: Optional[int] = None*, ***kwargs*) → *Response*

Asynchronously process a given TagScript string. This method has no additional attributes from the *Interpreter* class. See `Interpreter.process` for full documentation.

5.2 Context

class bTagScript.interpreter.**Context**(*verb:* Verb, *res:* Response, *interpreter:* Interpreter, *og: str*)

Bases: `object`

An object containing data on the TagScript block processed by the interpreter. This class is passed to adapters and blocks during processing.

verb

The Verb object representing a TagScript block.

Type
Verb

original_message

The original message passed to the interpreter.

Type
str

interpreter

The interpreter processing the TagScript.

Type
Interpreter

5.3 Response

class bTagScript.interpreter.**Response**(***, *variables: Optional[Dict[str, Adapter]] = None*, *extras: Optional[Dict[str, Any]] = None*)

Bases: `object`

An object containing information on a completed TagScript process.

body

The cleaned message with all verbs interpreted.

Type
str

actions

A dictionary that blocks can access and modify to define post-processing actions.

Type
Dict[str, Any]

variables

A dictionary of variables that blocks such as the *LooseVariableGetterBlock* can access.

Type
Dict[str, *Adapter*]

extras

A dictionary of extra keyword arguments that blocks can use to define their own behavior.

Type
Dict[str, Any]

5.4 Node

class bTagScript.interpreter.**Node**(*coordinates: Tuple[int, int]*, *verb: Optional[*Verb*] = None*)

Bases: object

A low-level object representing a bracketed block.

coordinates

The start and end position of the bracketed text block.

Type
Tuple[int, int]

verb

The determined Verb for this node.

Type
Optional[*Verb*]

output

The *Block* processed output for this node.

5.4.1 build_node_tree

bTagScript.interpreter.**build_node_tree**(*message: str*) → List[*Node*]

Function that finds all possible nodes in a string.

Parameters
message (*str*) – The string to find nodes in.

Returns
A list of all possible text bracket blocks.

Return type
List[*Node*]

CHAPTER

SIX

INTERFACE

class bTagScript.interface.**Adapter**

Bases: object

The base class for TagScript adapters.

Implementations must subclass this to create adapters.

get_value(*ctx:* Context) → Optional[str]

Processes the adapter's actions for a given *Context*.

Subclasses must implement this.

Parameters
ctx (Context) – The context object containing the TagScript *Verb*.

Returns
The adapters's processed value.

Return type
Optional[str]

Raises
NotImplementedError – The subclass did not implement this required method.

class bTagScript.interface.**Block**

Bases: object

The base class for TagScript blocks.

Implementations must subclass this to create new blocks.

ACCEPTED_NAMES

The accepted names for this block. This ideally should be set as a class attribute.

Type
Tuple[str, ...]

classmethod will_accept(*ctx:* Context) → bool

Describes whether the block is valid for the given *Context*.

Parameters
ctx (Context) – The context object containing the TagScript *Verb*.

Returns
Whether the block should be processed for this *Context*.

Return type
bool

pre_process(*ctx:* Context) → Optional[str]

Any pre processing that needs to be done before the block is processed.

process(*ctx:* Context) → Optional[str]

Processes the block's actions for a given *Context*.

Subclasses must implement this.

Parameters
ctx (Context) – The context object containing the TagScript *Verb*.

Returns
The block's processed value.

Return type
Optional[str]

Raises
NotImplementedError – The subclass did not implement this required method.

post_process(*ctx:* Context) → Optional[str]

Any post processing that needs to be done after the block is processed.

bTagScript.interface.**verb_required_block**(*implicit: bool*, *, *parameter: bool = False*, *payload: bool = False*) → *Block*

Get a Block subclass that requires a verb to implicitly or explicitly have a parameter or payload passed.

Parameters

- **implicit** (*bool*) – Specifies whether the value is required to be passed implicitly or explicitly. {block()} would be allowed if implicit is False.
- **parameter** (*bool*) – Passing True will cause the block to require a parameter to be passed.
- **payload** (*bool*) – Passing True will cause the block to require the payload to be passed.

CHAPTER

SEVEN

VERB

class bTagScript.verb.**Verb**(*verb_string: Optional[str] = None*, *, *limit: int = 2000*)

Bases: object

Represents the passed TagScript block.

Parameters

- **verb_string** (*Optional[str]*) – The string to parse into a verb.
- **limit** (*int*) – The maximum number of characters to parse.

declaration

The text used to declare the block.

Type
Optional[str]

parameter

The text passed to the block parameter in the parentheses.

Type
Optional[str]

payload

The text passed to the block payload after the colon.

Type
Optional[str]

Example

Below is a visual representation of a block and its attributes:

```
.. tagscript::
```

Normally {declaration(parameter):payload}

set_payload() → None

Set the payload

open_parameter(*i: int*) → None

Open the parameter

close_parameter(*i: int*) → bool

Close the parameter

CHAPTER

EIGHT

BLOCK MODULE

class bTagScript.block.**BreakBlock**

Bases: *Block*

The break block will force the tag output to only be the payload of this block, if the passed expresssion evaluates true. If no message is provided to the payload, the tag output will be empty.

This differs from the `StopBlock` as the stop block stops all tagscript processing and returns its message while the break block continues to process blocks. If command blocks exist after the break block, they will still execute.

Usage: `{break(<expression>):[message]}`

Aliases: `short, shortcircuit`

Payload: `message`

Parameter: `expression`

Examples:

```
{break(=={args}):You did not provide any input.}
```

process(*ctx:* Context) → Optional[str]

Process the block and break the tag.

class bTagScript.block.**CommentBlock**

Bases: *Block*

The comment block is literally just for comments, it will not be parsed, however it will be removed from your codes output.

Usage: `{comment([other]):[text]}`

Aliases: /, Comment, comment, //

Payload: `text`

Parameter: `other`

```
{//:Comment!}

{Comment(Something):Comment!}
```

process(*ctx:* Context) → Optional[str]

Remove the block

class bTagScript.block.**AllBlock**

Bases: VerbRequiredBlock

The all block checks that all of the passed expressions are true. Multiple expressions can be passed to the parameter by splitting them with |.

The payload is a required message that must be split by |. If the expression evaluates true, then the message before the | is returned, else the message after is returned.

Usage: {all(<expression|expression|...>):<message>}

Aliases: and

Payload: message

Parameter: expression

Examples:

```
{all({args}>=100|{args}<=1000):You picked {args}.|You must provide a number between␣
↪100 and 1000.}
# if {args} is 52
You must provide a number between 100 and 1000.

# if {args} is 282
You picked 282.
```

process(*ctx:* Context) → Optional[str]

Process all the expressions

class bTagScript.block.**AnyBlock**

Bases: VerbRequiredBlock

The any block checks that any of the passed expressions are true. Multiple expressions can be passed to the parameter by splitting them with |.

The payload is a required message that must be split by |. If the expression evaluates true, then the message before the | is returned, else the message after is returned.

Usage: {any(<expression|expression|...>):<message>}

Aliases: or

Payload: message

Parameter: expression

Examples:

```
{any(hi=={args}|hello=={args}|heyy=={args}):Hello {user}!|How rude.}
If {args} is hi
Hello _Leg3ndary#0001!

If {args} is what's up!
How rude.
```

process(*ctx:* Context) → Optional[str]

Process the any block

class bTagScript.block.**IfBlock**

Bases: VerbRequiredBlock

The if block returns a message based on the passed expression to the parameter. An expression is represented by two values compared with an operator.

The payload is a required message that must be split by |. If the expression evaluates true, then the message before the | is returned, else the message after is returned.

Expression Operators:

Operator	Check	Example	Description
==	equality	a==a	value 1 is equal to value 2
!=	inequality	a!=b	value 1 is not equal to value 2
>	greater than	5>3	value 1 is greater than value 2
<	less than	4<8	value 1 is less than value 2
>=	greater than or equality	10>=10	value 1 is greater than or equal to value 2
<=	less than or equality	5<=6	value 1 is less than or equal to value 2

Usage: {if(<expression>):<message>]}

Payload: message

Parameter: expression

Examples:

```
{if(63=={args}):You guessed it! The number I was thinking of was 63!|Too {if({args}
↪<63):low|high}, try again.}
If args is 63
You guessed it! The number I was thinking of was 63!

If args is 73
Too low, try again.

If args is 14
Too high, try again.
```

process(*ctx:* Context) → Optional[str]

Process the if block

class bTagScript.block.**CountBlock**

Bases: VerbRequiredBlock

The count block will count how much of text is in message. This is case sensitive and will include substrings, if you don't provide a parameter, it will count the spaces in the message.

Usage: {count([text]):<message>}

Aliases: None

Payload: message

Parameter: text

```
{count(Tag):TagScript}
1
```

(continues on next page)

(continued from previous page)

```
{count(Tag):Tag Script TagScript}
2

{count(t):Hello World, Tag, Script}
1 as there's only one lowercase t in the entire string
```

process(*ctx:* Context) → Optional[str]

Check the count of a string

class bTagScript.block.**LengthBlock**

Bases: VerbRequiredBlock

The length block will check the length of the given String. If a parameter is passed in, the block will check the length based on what you passed in, w for word, s for spaces. If you provide an invalid parameter, the block will return -1.

Usage: `{length(["w", "s"]):<text>}`

Aliases: `len`

Payload: `text`

Parameter: `"w", "s"`

```
{length:TagScript}
9

{len(w):Tag Script}
2

{len(s):Hello World, Tag, Script}
3

{len(space):Hello World, Tag, Script}
-1
```

process(*ctx:* Context) → Optional[str]

Check the length of a string

class bTagScript.block.**BlacklistBlock**

Bases: VerbRequiredBlock

The blacklist block will attempt to convert the given parameter into a channel or role, using name or ID. If the user running the tag is in the targeted channel or has the targeted role, the tag will stop processing and it will send the response if one is given. Multiple role or channel requirements can be given, and should be split by a ",".

Usage: `{blacklist(<role,channel>):[response]}`

Payload: `response`

Parameter: `role, channel`

Examples:

```
{blacklist(Muted)}
{blacklist(#support):This tag is not allowed in #support.}
{blacklist(Tag Blacklist, 668713062186090506):You are blacklisted from using tags.}
```

process(*ctx:* Context) → Optional[str]

Process the blacklists

class bTagScript.block.**CommandBlock**(*limit: int = 3*)

Bases: VerbRequiredBlock

Run a command as if the tag invoker had ran it. Only 3 command blocks can be used in a tag.

Usage: {command:<command>}

Aliases: c, com, command

Payload: command

Parameter: None

Examples:

```
{c:ping}
# Invokes ping command

{c:ban {target(id)} Chatflood/spam}
# Invokes ban command on the pinged user with the reason as "Chatflood/spam"
```

process(*ctx:* Context) → Optional[str]

Process the block and update response.actions

class bTagScript.block.**CooldownBlock**

Bases: VerbRequiredBlock

The cooldown block implements cooldowns when running a tag. The parameter requires 2 values to be passed: rate and per integers. The rate is the number of times the tag can be used every per seconds.

The payload requires a key value, which is the key used to store the cooldown. A key should be any string that is unique. If a channel's ID is passed as a key, the tag's cooldown will be enforced on that channel. Running the tag in a separate channel would have a different cooldown with the same rate and per values.

The payload also has an optional message value, which is the message to be sent when the cooldown is exceeded. If no message is passed, the default message will be sent instead. The cooldown message supports 2 blocks: key and retry_after.

Usage: {cooldown(<rate>|<per>):<key>|[message]}

Payload: key, message

Parameter: rate, per

Examples:

```
{cooldown(1|10):{author(id)}}
the tag author used the tag more than once in 10 seconds
The bucket for 741074175875088424 has reached its cooldown. Retry in 3.25 seconds."

{cooldown(3|3):{channel(id)}|Slow down! This tag can only be used 3 times per 3␣
↪seconds per channel. Try again in **{retry_after}** seconds."}
the tag was used more than 3 times in 3 seconds in a channel
Slow down! This tag can only be used 3 times per 3 seconds per channel. Try again␣
↪in **0.74** seconds.
```

classmethod create_cooldown(*key: Any*, *rate: int*, *per: int*) → CooldownMapping

Create a new cooldown for the given key.

process(*ctx:* Context) → Optional[str]

Process the cooldown block.

class bTagScript.block.**DeleteBlock**

Bases: *Block*

The delete block will delete the message if the condition provided in the parameter is met, or if just the block is added, the message will be deleted. Only one delete block will be processed, the rest, removed, but ignored.

Note: This block will only set the actions "delete" key to True/False. You must set the behaviour manually.

Usage: `{delete(<expression>)}`

Aliases: `del`

Payload: `None`

Parameter: `expression`

```
{delete}
{del(true==true)}
```

process(*ctx:* Context) → Optional[str]

Process the delete

class bTagScript.block.**EmbedBlock**

Bases: *Block*

An embed block will send an embed in the tag response. There are two ways to use the embed block, either by using properly formatted embed JSON from an embed generator or manually inputting the accepted embed attributes.

JSON

Using JSON to create an embed offers complete embed customization. Multiple embed generators are available online to visualize and generate embed JSON.

Usage: `{embed(<json>)}`

Payload: `None`

Parameter: `json`

```
{embed({"title":"Hello!", "description":"This is a test embed."})}
{embed({
    "title":"Here's a random duck!",
    "image":{"url":"https://random-d.uk/api/randomimg"},
    "color":15194415
})}
```

Manual

The following embed attributes can be set manually:

- `title`
- `description`

- `color`
- `url`
- `thumbnail`
- `image`
- `field` - (See below)

Adding a field to an embed requires the payload to be split by `|`, into either 2 or 3 parts. The first part is the name of the field, the second is the text of the field, and the third optionally specifies whether the field should be inline.

Usage: `{embed(<attribute>):<value>}`

Payload: `value`

Parameter: `attribute`

```
{embed(color):#37b2cb}
{embed(title):Rules}
{embed(description):Follow these rules to ensure a good experience in our server!}
{embed(field):Rule 1|Respect everyone you speak to.|false}
```

Both methods can be combined to create an embed in a tag. The following tagscript uses JSON to create an embed with fields and later set the embed title.

:: tagscript:

```
{embed({{"fields":[{"name":"Field 1","value":"field description","inline":false}]})}
{embed(title):my embed title}
```

static get_embed(*ctx:* Context) → Embed

Gets the embed object

static value_to_color(*value: Optional[Union[int, str]]*) → Colour

Converts a value to a discord.Colour object

text_to_embed(*text: str*) → Embed

Converts json to an embed

classmethod update_embed(*embed: Embed*, *attribute: str*, *value: str*) → Embed

Update the embed with all attributes

static return_error(*error: Exception*) → str

Return an error message

static return_embed(*ctx:* Context, *embed: Embed*) → str

Returns the embed

process(*ctx:* Context) → Optional[str]

Process the block

class bTagScript.block.**OverrideBlock**

Bases: *Block*

Override a command's permission requirements. This can override mod, admin, or general user permission requirements when running commands with the *Command Block*. Passing no parameter will default to overriding all permissions.

In order to add a tag with the override block, the tag author must have `Manage Server` permissions.

This will not override bot owner commands or command checks.

Usage: `{override(["admin"|"mod"|"permissions"]):[command]}`

Aliases: `bypass`

Payload: `command`

Parameter: `"admin", "mod", "permissions"`

Examples:

```
{override}
overrides all commands and permissions

{override(admin)}
overrides commands that require the admin role

{bypass(permissions)}
{bypass(mod)}
overrides commands that require the mod role or have user permission requirements
```

process(*ctx:* Context) → Optional[str]

Process the block and update response.actions with correct overrides

class bTagScript.block.**ReactBlock**(*limit: int = 5*)

Bases: `VerbRequiredBlock`

The react block will set the actions "react" key to a list of reactions.

Note: You must set the behaviour manually.

Usage: `{react:<emojis>}`

Aliases: `None`

Payload: `emojis`

Parameter: `None`

```
{react:}
{react:,:)}
{react:~:)~:D}
```

process(*ctx:* Context) → Optional[str]

Process the reactions

class bTagScript.block.**RedirectBlock**

Bases: `VerbRequiredBlock`

Redirects the tag response to either the given channel, the author's DMs, or uses a reply based on what is passed to the parameter.

Usage: `{redirect(<"dm"|"reply"|channel>)}`

Payload: `None`

Parameter: `"dm", "reply", "channel"`

Examples:

```
{redirect(dm)}
{redirect(reply)}
{redirect(#general)}
{redirect(626861902521434160)}
```

process(*ctx:* Context) → Optional[str]

Process the redirect block and params

class bTagScript.block.**RequireBlock**

Bases: VerbRequiredBlock

The require block will attempt to convert the given parameter into a channel role or member, using name or ID. If the user running the tag is not in the targeted channel or doesn't have the targeted role, the tag will stop processing and it will send the response if one is given. Multiple role or channel requirements can be given, and should be split by a ",".

Usage: {require(<role, channel, member>):[response]}

Aliases: whitelist

Payload: response

Parameter: role, channel, member

Examples:

```
{require(Moderator)}
{require(#general, #bot-cmds):This tag can only be run in #general and #bot-cmds.}
{require(757425366209134764, 668713062186090506, 737961895356792882):You aren't␣
↪allowed to use this tag.}
```

process(*ctx:* Context) → Optional[str]

Process the requirements

class bTagScript.block.**MathBlock**

Bases: *Block*

A math block is a block that contains a math expression. Will write out everything later bleh

Usage: {math:<expression>}

Aliases: math, m, +, calc

Payload: expression

Parameter: None

Examples:

```
{m:2+3}
5.0

{math:7(2+3)}
42.0

{math:trunc(7(2+3))}
42
```

process(*ctx:* Context) → Optional[str]

Try and process the block into a float

class bTagScript.block.**OrdinalAbbreviationBlock**

Bases: *Block*

The ordinalabbreviation block returns the ordinal abbreviation of a number. If a parameter is provided, it must be, one of, c, comma, indicator, i Comma being adding commas every 3 digits, indicator, meaning the ordinal indicator. (The st of 1st, nd of 2nd, etc.)

The number may be positive or negative, if the payload is invalid, -1 is returned.

Usage: `{ord(["c", "comma", "i", "indicator"]):<number>}`

Aliases: `None`

Payload: `number`

Parameter: `"c", "comma", "i", "indicator"`

```
{ord:1000}
1,000th

{ord(c):1213123}
1,213,123

{ord(i):2022}
2022nd
```

process(*ctx:* Context) → str

Process the ordinal abbreviation block

class bTagScript.block.**RandomBlock**

Bases: `VerbRequiredBlock`

Pick a random item from a list of strings, split by either ~ or `,`. An optional seed can be provided to the parameter to always choose the same item when using that seed. You can weight options differently by adding a weight and | before the item.

Usage: `{random([seed]):<list>}`

Aliases: `#, rand`

Payload: `list`

Parameter: `seed`

Examples:

```
{random:Carl,Harold,Josh} attempts to pick the lock!
Possible Outputs:
Josh attempts to pick the lock!
Carl attempts to pick the lock!
Harold attempts to pick the lock!

{=(insults):You're so ugly that you went to the salon and it took 3 hours just to␣
↪get an estimate.~I'll never forget the first time we met, although I'll keep␣
↪trying.~You look like a before picture.}
{=(insult):{#:{insults}}}
{insult}
```

(continues on next page)

(continued from previous page)

```
Assigns a random insult to the insult variable

{#:5|Cool,3|Lame}
5 to 3 chances of being cool vs lame
```

process(*ctx:* Context) → Optional[str]

Process the randomness woo

class bTagScript.block.**RangeBlock**

Bases: VerbRequiredBlock

The range block picks a random number from a range of numbers seperated by `-`. The number range is inclusive, so it can pick the starting/ending number as well. Using the rangef block will pick a number to the tenth decimal place.

An optional seed can be provided to the parameter to always choose the same item when using that seed.

Usage: `{range([seed]):<lowest-highest>}`

Aliases: `rangef`

Payload: `number`

Parameter: `seed`

Examples:

```
Your lucky number is {range:10-30}!
Your lucky number is 14!
Your lucky number is 25!

{=(height):{rangef:5-7}}
I am guessing your height is {height}ft.
I am guessing your height is 5.3ft.
```

process(*ctx:* Context) → Optional[str]

Process the range block

class bTagScript.block.**PythonBlock**

Bases: VerbRequiredBlock

The in block serves three different purposes depending on the alias that is used.

The `in` alias checks if the parameter is anywhere in the payload.

`contain` strictly checks if the parameter is the payload, split by whitespace.

`index` finds the location of the parameter in the payload, split by whitespace. If the parameter string is not found in the payload, it returns 1.

index is used to return the value of the string form the given list of

Usage: `{in(<string>):<payload>}`

Aliases: `index, contains`

Payload: `payload`

Parameter: `string`

Examples:

```
{in(apple pie):banana pie apple pie and other pie}
true
{in(mute):How does it feel to be muted?}
true
{in(a):How does it feel to be muted?}
false

{contains(mute):How does it feel to be muted?}
false
{contains(muted?):How does it feel to be muted?}
false

{index(food):I love to eat food. everyone does.}
4
{index(pie):I love to eat food. everyone does.}
-1
```

will_accept(*ctx:* Context) → bool

Check if we can accept

process(*ctx:* Context) → Optional[str]

Process the block

class bTagScript.block.**ReplaceBlock**

Bases: VerbRequiredBlock

The replace block will replace specific characters in a string. The parameter should split by a `,`, containing the characters to find before the command and the replacements after.

Usage: `{replace(<original,new>):<message>}`

Aliases: sub

Payload: message

Parameter: original, new

```
{replace(o,i):welcome to the server}
welcime ti the server

{replace(1,6):{args}}
if {args} is 1637812
6637862

{replace(, ):Test}
T e s t
```

process(*ctx:* Context) → Optional[str]

Replace the characters in the payload

class bTagScript.block.**StopBlock**

Bases: VerbRequiredBlock

The stop block stops tag processing if the given parameter is true. If a message is passed to the payload it will return that message.

Usage: `{stop(<bool>):[string]}`

Aliases: `halt, error`

Payload: `string`

Parameter: `bool`

Example:

```
{stop(==){args}):You must provide arguments for this tag.}
enforces providing arguments for a tag
```

process(*ctx:* Context) → Optional[str]

Process the stop block

class bTagScript.block.**StrfBlock**

Bases: *Block*

The strf block converts and formats timestamps based on strftime formatting spec. Two types of timestamps are supported: ISO and epoch. If a timestamp isn't passed, the current UTC time is used.

Invoking this block with `Unix Specific Services` will return the current Unix timestamp.

Usage: `{strf([timestamp]):<format>}`

Aliases: `unix`

Payload: `format`

Parameter: `timestamp`

Example:

```
{strf:%Y-%m-%d}
2021-07-11

{strf({user(timestamp)}):%c}
Fri Jun 29 21:10:28 2018

{strf(1420070400):%A %d, %B %Y}
Thursday 01, January 2015

{strf(2019-10-09T01:45:00.805000):%H:%M %d-%B-%Y}
01:45 09-October-2019

{unix}
1629182008
```

process(*ctx:* Context) → Optional[str]

Process the strf block

class bTagScript.block.**URLDecodeBlock**

Bases: `VerbRequiredBlock`

This block will decode a given url into a string with non-url compliant characters replaced. Using + as the parameter will replace spaces with + rather than `%20`.

Usage: `{urldecode(["+"]):<string>}`

Payload: string

Parameter: "+", None

Examples:

```
{urldecode:covid-19%20sucks}
covid-19 sucks

{urldecode(+):im+stuck+at+home+writing+docs}
im stuck at home writing docs
```

This block is just the reverse of the urlencode block

process(*ctx:* Context) → Optional[str]

Process the block

class bTagScript.block.**URLEncodeBlock**

Bases: VerbRequiredBlock

This block will encode a given string into a properly formatted url with non-url compliant characters replaced. Using + as the parameter will replace spaces with + rather than %20.

Usage: {urlencode(["+"]):<string>}

Payload: string

Parameter: "+", None

Example:

```
{urlencode:covid-19 sucks}
covid-19%20sucks

{urlencode(+):im stuck at home writing docs}
im+stuck+at+home+writing+docs

You can use this to search up blocks
Eg if {args} is command block

<https://btagscript.readthedocs.io/en/latest/search.html?q={urlencode(+):{args}}&
↪check_keywords=yes&area=default>
<https://btagscript.readthedocs.io/en/latest/search.html?q=command+block&check_
↪keywords=yes&area=default>
```

process(*ctx:* Context) → str

Processes the block's actions for a given *Context*.

Subclasses must implement this.

Parameters
ctx (Context) – The context object containing the TagScript *Verb*.

Returns
The block's processed value.

Return type
Optional[str]

Raises
NotImplementedError – The subclass did not implement this required method.

class bTagScript.block.**DebugBlock**

Bases: *Block*

The debug block allows you to debug your tagscript quickly and easily, it will save the output to the debug_var key in the response dict. Separate the variables you want to include or exclude with a comma or a tilde.

If no parameters are provided in addition to no payload, all variables will be included. If no parameters are provided and a payload is provided, it will assume you want to include those variables.

Usage: `{debug(["i", "include", "e", "exclude"]):<variables>}`

Aliases: `None`

Payload: `variables`

Parameter: `"i", "include", "e", "exclude"`

Note: {debug} is the same as {debug(exclude):}

{debug:somevar~anothervar} is the same as {debug(include):somevar~anothervar}

Examples:

Note: THIS SHOULD ALWAYS BE PLACED AT THE VERY BOTTOM, IT WILL NOT RETURN ANYTHING UNDER IT.

```
Assuming we have the following tagscript, we first set the var something, then set
parsed (using the dollar sign method), to Hello|World, (assume we actually wanted
↪just the Hello
but we forgot)

{=(something):Hello/World}
{$parsed:{something(1)}}
{if({parsed}==Hello):Hello|Bye}

Running this would provided the output Bye, using the debug block below:
{debug}
We'll get all the variables at their, "final state"
This will be provided in a dict, which you can further parse and output to your
↪liking.

EG, in YAML format:
something: Hello/World
parsed: Hello/World

This allow's you to see that you forgot to parse with a delimiter which will lead
↪to easy fixing.
```

process(*ctx:* Context) → Optional[str]

Debug the tagscript!

class bTagScript.block.**VarBlock**

Bases: `VerbRequiredBlock`

Variables are useful for choosing a value and referencing it later in a tag. Variables can be referenced using brackets as any other block. Note that if the variable's name is being "used" by any other block the variable will be ignored.

Usage: `{=(<name>):<value>}`

Aliases: `let, var, =`

Payload: `value`

Parameter: `name`

Examples:

```
{=(prefix):!}
The prefix here is `{prefix}`.
The prefix here is `!`.

{let(day):Monday}
{if({day}==Wednesday):It's Wednesday my dudes!|The day is {day}.}
The day is Monday.

Variables can also be created like so
{$<name>:<value>}
{$day:Monday} == {=(day):Monday}
```

process(*ctx:* Context) → Optional[str]

Process the block and assign the variable.

class bTagScript.block.**LooseVariableGetterBlock**

Bases: *Block*

The loose variable block represents the adapters for any seeded or defined variables. This variable implementation is considered "loose" since it checks whether the variable is valid during `process()`, rather than `will_accept()`. You may also define variables here with {$<variable name>:<value>}

Usage: `{<variable_name>([parameter]):[payload]}`

Aliases: This block is valid for any inputted declaration.

Payload: Depends on the variable's underlying adapter.

Parameter: Depends on the variable's underlying adapter.

Examples:

```
{=(example):This is my variable.}
{example}
This is my variable.

{$variablename:This is another variable.}
{variablename}
This is another variable.
```

will_accept(*ctx:* Context) → bool

This block will accept any declaration.

process(*ctx:* Context) → Optional[str]

This block will check whether the variable is valid.

class bTagScript.block.**StrictVariableGetterBlock**

Bases: *Block*

The strict variable block represents the adapters for any seeded or defined variables. This variable implementation is considered "strict" since it checks whether the variable is valid during `will_accept()` and is only processed if the declaration refers to a valid variable.

Usage: `{<variable_name>([parameter]):[payload]}`

Aliases: This block is valid for any variable name in *Response.variables*.

Payload: Depends on the variable's underlying adapter.

Parameter: Depends on the variable's underlying adapter.

Examples:

```
{=(example):This is my variable.}
{example}
This is my variable.
```

will_accept(*ctx:* Context) → bool

Check if the declaration is in the response variables

process(*ctx:* Context) → Optional[str]

Process the strict variable block

CHAPTER

NINE

ADAPTER MODULE

class bTagScript.adapter.**SafeObjectAdapter**(*base*)

Bases: *Adapter*

For objects

get_value(*ctx:* Verb) → str

Get the value safely

class bTagScript.adapter.**StringAdapter**(*string: str*, *, *escape: bool = False*)

Bases: *Adapter*

String adapter, allows blocks to be parsed, used basically only for variables

get_value(*ctx:* Verb) → str

Get the value given the verb

handle_ctx(*ctx:* Verb) → str

Transform any parsing data the block may have

return_value(*string: str*) → str

Return the value, escaped

class bTagScript.adapter.**IntAdapter**(*integer: int*)

Bases: *Adapter*

IntAdapter, so far no use for this?

get_value(*ctx:* Verb) → str

Get the value of the int into string, not sure why this even exists

class bTagScript.adapter.**FunctionAdapter**(*function_pointer: Callable[[], str]*)

Bases: *Adapter*

Function adapter...

Would be cool to have functions in tagscript

get_value(*ctx:* Verb) → str

Run the function and get the value

class bTagScript.adapter.**AttributeAdapter**(*base: Union[TextChannel, Member, Guild]*)

Bases: *Adapter*

Base attribute adapter for discord.py objects

update_attributes() → None

Update attributes for the block

update_methods() → None

Update methods for the block

get_value(*ctx:* Verb) → str

Get the value for the adapter

class bTagScript.adapter.**MemberAdapter**(*base: Union[TextChannel, Member, Guild]*)

Bases: *AttributeAdapter*

The {author} block with no parameters returns the tag invoker's full username and discriminator, but passing the attributes listed below to the block payload will return that attribute instead.

Aliases: user

Usage: {author([attribute])

Payload: None

Parameter: attribute, None

id

The author's Discord ID.

name

The author's username.

nick

The author's nickname, if they have one, else their username.

avatar

A link to the author's avatar, which can be used in embeds.

discriminator

The author's discriminator.

created_at

The author's account creation date.

timestamp

The author's account creation date as a UTC timestamp.

joined_at

The date the author joined the server.

mention

A formatted text that pings the author.

bot

Whether or not the author is a bot.

color

The author's top role's color as a hex code.

top_role

The author's top role.

boost

If the user boosted, this will be the the UTC timestamp of when they did, if not, this will be empty.

timed_out

If the user is timed out, this will be the the UTC timestamp of when they'll be "untimed-out", if not timed out, this will be empty.

banner

The users banner url

roleids

A list of the author's role IDs, split by spaces.

update_attributes() → None

Update the adapter with all it's needed attributes

class bTagScript.adapter.**ChannelAdapter**(*base: Union[TextChannel, Member, Guild]*)

Bases: *AttributeAdapter*

The {channel} block with no parameters returns the channel's full name but passing the attributes listed below to the block payload will return that attribute instead.

Usage: {channel([attribute])

Payload: None

Parameter: attribute, None

id

The channel's ID.

name

The channel's name.

created_at

The channel's creation date.

timestamp

The channel's creation date as a UTC timestamp.

nsfw

Whether the channel is nsfw.

mention

A formatted text that pings the channel.

topic

The channel's topic.

slowmode

The channel's slowmode in seconds, 0 if disabled

update_attributes() → None

Update block attributes

class bTagScript.adapter.**GuildAdapter**(*base: Union[TextChannel, Member, Guild]*)

Bases: *AttributeAdapter*

The {server} block with no parameters returns the server's name but passing the attributes listed below to the block payload will return that attribute instead.

Aliases: `guild`

Usage: `{server([attribute])`

Payload: None

Parameter: attribute, None

`id`
: The server's ID.

`name`
: The server's name.

`icon`
: A link to the server's icon, which can be used in embeds.

`created_at`
: The server's creation date.

`timestamp`
: The server's creation date as a UTC timestamp.

`member_count`
: The server's member count.

`bots`
: The number of bots in the server.

`humans`
: The number of humans in the server.

`description`
: The server's description if one is set, or "No description".

`random`
: A random member from the server.

`update_attributes()` → None
: Update block attributes

`update_methods()` → None
: Update methods for the block

`random_member()` → None
: Return a random member

CHAPTER

TEN

EXCEPTIONS

exception bTagScript.exceptions.**TagScriptError**

Bases: Exception

Base class for all module errors.

exception bTagScript.exceptions.**WorkloadExceededError**

Bases: *TagScriptError*

Raised when the interpreter goes over its passed character limit.

exception bTagScript.exceptions.**ProcessError**(*error: Exception*, *response:* Response, *interpreter:* Interpreter)

Bases: *TagScriptError*

Raised when an exception occurs during interpreter processing.

original

The original exception that occurred during processing.

Type
Exception

response

The incomplete response that was being processed when the exception occurred.

Type
Response

interpreter

The interpreter used for processing.

Type
Interpreter

exception bTagScript.exceptions.**EmbedParseError**

Bases: *TagScriptError*

Raised if an exception occurs while attempting to parse an embed.

exception bTagScript.exceptions.**BadColourArgument**(*argument: str*)

Bases: *EmbedParseError*

Raised when the passed input fails to convert to discord.Colour.

argument

The invalid input.

Type
str

exception bTagScript.exceptions.**StopError**(*message: str*)

Bases: *TagScriptError*

Raised by the StopBlock to stop processing.

message

The stop error message.

Type
str

exception bTagScript.exceptions.**CooldownExceeded**(*message: str*, *cooldown: Cooldown*, *key: str*, *retry_after: float*)

Bases: *StopError*

Raised by the cooldown block when a cooldown is exceeded.

message

The cooldown error message.

Type
str

cooldown

The cooldown bucket with information on the cooldown.

Type
discord.ext.commands.Cooldown

key

The cooldown key that reached its cooldown.

Type
str

retry_after

The seconds left til the cooldown ends.

Type
float

exception bTagScript.exceptions.**BlocknameDuplicateError**(*blockname: str*)

Bases: *TagScriptError*

Raised when a block's name is duplicated when passed to the interpreter

blockname

The blockname that was duplicated

Type
str

Welcome to bTagScript's official documentation.

Note: This documentation is still being regularily updated! Best to check back later if you can't find something you need.

PYTHON MODULE INDEX

b

INDEX

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W