YAGPDB

Functions

Functions are underappreciated. In general, not just in templates. // Rob Pike

> Every function having both cases possible for an argument - ID/name, then this name is handled case insensitive, for example `getRole "yagpdb"` and `getRole "yAgPdB"` would have same responses even if server has both of these roles, so using IDs is better.

Channel

> The ratelimit for editing a channel is 2 requests per 10 minutes per channel.

Function	Description
`editChannelName` channel "newName"	Function edits channel's name. `channel` can be either ID, "name" or even `nil` if triggered in that channel name change is intended to happen. `"newName"` has to be of type *string*. For exampl > `{{editChannelName nil (print "YAG " (randInt 1000))}}`
`editChannelTopic` channel "newTopic"	Function edits channel's topic/description. `channel` can be either ID, "name" or `nil` if triggered in that channel where name change is intended to happen. `"newTopic"` has to be of type *string*. For example > `{{editChannelTopic nil "YAG is cool"}}`
`getChannel` channel	Function returns full channel object of given `channel` argument which can be either its ID, name or `nil` for triggering channel, and is of typ **templates.CtxChannel*. For example > `{{(getChannel nil).Name}}` returns the name of the channel command was triggered in.
`getChannelOrThread` channel	Returns type **templates.CtxChannel* corresponding to Channel object.
	Returns the count of pinned messages in given

`getPinCount` channel	channel which can be either its ID, name or `nil` for triggering channel. Can be called 2 times for regular and 4 for premium servers
`getThread` channel	Returns type **templates.CtxChannel* corresponding to Channel object.

Database

Function	Description
`dbBottomEntries` pattern amount nSkip	Returns `amount (max 100)` top entries of key determined by the `pattern` from the database, sorted by the value in a ascending order.
`dbCount` (userID|pattern|query)	Returns the count of all database entries which ar not expired. Optional arguments: if `userID` is given, counts entries for that userID; if `pattern` only those keys are counted that match the given pattern; and if `query` is provided, it should be ar sdict with the following keys: • `userID` - only counts entries with that userI defaults to counting entries with any userID. • `pattern` - only counts dbEntry keys with names matching the pattern given, defaults to counting entries with any name.
`dbDel` userID key	Deletes the specified key for the specified value from the database.
`dbDelByID` userID ID	Deletes database entry by its ID.
`dbDelMultiple` query amount skip	Deletes `amount (max 100)` entries from the database matching the criteria provided. `query` should be an *sdict* with the following options: • `userID` - only deletes entries with the dbEntry field .UserID provided, defaults to deleting entries with any ID. • `pattern` - only deletes entry keys with a name matching the pattern given. • `reverse` - if true, starts deleting entries with the lowest values first; otherwise starts deletin entries with the highest values first. Default is `false`. Returns the number of rows that got deleted or an

	error.
`dbGet` userID key	Retrieves a value from the database for the specified user, this returns DBEntry object.
`dbGetPattern` userID pattern amount nSkip	Retrieves up to `amount (max 100)` entries fror the database in ascending order.
`dbGetPatternReverse` userID pattern amount nSkip	Retrieves `amount (max 100)` entries from the database in descending order.
`dbIncr` userID key incrBy	Increments the value for specified key for the specified user, if there was no value then it will be set to `incrBy`. Also returns the entry's current, increased value.
`dbRank` query userID key	Returns the rank of the entry specified by the user ID and key provided in the set of entries matching the criteria provided. `query` specifies the set of entries that should be considered, and should be sdict with the following options: • `userID` - only includes entries with that use ID, defaults to including entries with any user ID • `pattern` - only includes database's `key` entries with names matching the pattern giver defaults to counting entries with any name • `reverse` - if true, entries with lower value have higher rank; otherwise entries with highe value have higher rank. Default is `false`.
`dbSet` userID key value	Sets the value for the specified `key` for the specific `userID` to the specified `value`. `userID` can be any number of type *int64*. Values are stored either as of type *float64* (for numbers, oct or hex) or as varying type in bytes (fo *slices*, *maps*, *strings* etc) depending on input argument.
`dbSetExpire` userID key value ttl	Same as `dbSet` but with an expiration `ttl` which is an *int* and represents seconds.
`dbTopEntries` pattern amount nSkip	Returns `amount (max 100)` top entries of key determined by the `pattern` from the database, sorted by the value in a descending order.

Patterns are basic PostgreSQL patterns, not Regexp: An underscore `(_)` matches any single character; a

percent sign `(%)` matches any sequence of zero or more characters.

> **Note about saving numbers into database:** As stated above, database stores numbers as type *float64*. If you save a large number into database like an *int64* (which IDs are), the value will be truncated. To avoid this behavior, you can stringify the number before saving and convert it back to its original type when retrieving it. Example: `{{$v := .User.ID}} {{dbSet 0 "userid" (str $v)}} {{$fromDB := toInt (dbGet 0 "user_id").Value}}`
>
> `dict` key values are also retrieved as *int64,* so to use them for indexing one has to e.g. `index $x (toInt64 0)`

ExecCC

> `execCC` calls are limited to 1 / CC for non-premium users and 10 / CC for premium users.

Function	Description
`cancelScheduledUniqueCC` ccID key	Cancels a previously scheduled custom comman execution using `scheduleUniqueCC`.
`execCC` ccID channel delay data	Function that executes another custom command specified by `ccID`. With delay 0 the max recursi depth is 2 (using `.StackDepth` shows the current depth). `execCC` is rate-limited strictly at max 10 delayed custom commands executed per channel per minute, if you go over that it will be simply thrown away. Argument `channel` can be `nil`, channel's ID or name. The `delay` argume is execution delay of another CC is in seconds. The `data` argument is content that you pass to the other executed custom command. To retrieve that `data` you use `.ExecData`. This example is important > execCC example also next snippet which shows you same thing run using the same custom command > Snippets.
`scheduleUniqueCC` ccID channel delay key data	Same as `execCC` except there can only be 1 scheduled cc execution per server per key, if key already exists then it is overwritten with the new data and delay (as above, in seconds). An example would be a mute command that schedules the unmute action sometime in the future. However, let's say you use the unmute command again on the same user, you would war

	to override the last scheduled unmute to the new

ExecCC section's snippets:

- To demonstrate execCC and .ExecData using the same CC.

```
{{ $yag := "YAGPDB rules! " }}
{{ $ctr := 0 }} {{ $yourCCID := .CCID }}
{{ if .ExecData }}
    {{ $ctr = add .ExecData.number 1 }}
    {{ $yag = print $yag $ctr }} {{ .ExecData.YAGPDB }}
{{ else }}
    So, someone rules.
    {{ $ctr = add $ctr 1 }} {{ $yag = print $yag 1 }}
{{ end }}
{{ if lt $ctr 5 }}
    {{ execCC $yourCCID nil 3 (sdict "YAGPDB" $yag "number" $ctr) }}
{{ else }} FUN'S OVER! {{ end }}
```

Math

> Boolean logic (and, not, or) and comparison operators (eq, gt, lt, etc.) are covered in conditional branching.

Function	Description
`add` x y z ...	Returns x + y + z + ..., detects first number's type - is it *int* or *float* and based on that adds. (use `toFloat` on the first argument to force floating point math.) `{{add 5 4 3 2 -1}}` sums all these numbers and returns `13`.
`bitwiseAnd` x y	The output of bitwise AND is 1 if the correspondin bits of two operands is 1. If either bit of an operanc is 0, the result of corresponding bit is evaluated to 0. Example: `{{bitwiseAnd 12 25}}` returns `8`, that in binary 00001100 AND 00011001 is 00001000.
`bitwiseAndNot` x y	This function is called bit clear because of AND NOT. For example in the expression z = x AND NOT y, each bit of z is 0 if the corresponding bit of is 1; otherwise it equals to the corresponding bit o x. `{{bitwiseAndNot 7 12}}` returns `3`, tha is 0111 AND NOT 1100 is 11.

`bitwiseNot` x	The bitwise NOT operator inverts the bits of the argument. Example: `{{bitwiseNot 7}}` returns `-8`. that in binary 0111 to 1000
`bitwiseOr` x y z...	The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. Example: `{{bitwiseOr 12 25}}` returns `29`, that in binary 00001100 OR 00011001 is 00011101.
`bitwiseXor` x y	The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. Example: `{{bitwiseXor 12 25}}` returns `21`, that in binary 00001100 OR 00011001 is 00010101.
`bitwiseLeftShift` x y	Left shift operator shifts all bits towards left by a certain number of specified bits. The bit positions that have been vacated by the left shift operator ar filled with 0. Example: `{{range seq 0 3}} {{bitwiseLeftShift 212 .}} {{end}}` returns `212 424 848`
`bitwiseRightShift` x y	Right shift operator shifts all bits towards right by certain number of specified bits. Example: `{{range seq 0 3}} {{bitwiseRightShift 212 .}} {{end}}` returns `212 106 53`.
`cbrt` x	Returns the cube root of given argument in type *float64* e.g. `{{cbrt 64}}` returns `4`.
`div` x y z ...	Division, like `add` or `mult`, detects first number type first. `{{div 11 3}}` returns `3` whereas `{{div 11.1 3}}` returns `3.6999999999999997`
`fdiv` x y z ...	Meant specifically for floating point numbers division.
`log` x base	Log is a logarithm function using (log base of x). Arguments can be any type of numbers, as long a they follow logarithm logic. Return value is of type *float64*. If base argument is not given It is using natural logarithm (base e - The Euler's constant) a default. `{{ log "123" 2 }}` will return `6.94251450533924`.
`mathConst` "arg"	Function returns all constants available in golang' math package as *float64*. `"arg"` has to be a case-insensitive *string* from math constants list. F

	example `{{mathConst "sqrtphi"}}` would return `1.272019649514069`.
`max` x y	Returns the larger of x or y as type *float64*.
`min` x y	Returns the smaller of x or y as type *float64*.
`mod` x y	Mod (modulo) returns the floating-point remainder of x/y. `mod 17 3` returns `2` of type *float64*.
`mult` x y z ...	Multiplication, like `add` or `div`, detects first number's type. `{{mult 3.14 2}}` returns `6.28`
`pow` x y	Pow returns x**y, the base-x exponential of y whic have to be both numbers. Type is returned as *float64*. `{{ pow 2 3 }}` returns `8`.
`randInt` (stop, or start stop)	Returns a random integer between 0 and stop, or start - stop if two args are provided. Result will be `start <= random number < stop`. Example in section's Snippets.

Math section's snippets:

- `{{$d := randInt 10}}` Stores random *int* into variable `$d` (a random number from 0-9).
- To demonstrate rounding float to 2 decimal places.
  `{{div (round (mult 12.3456 100)) 100}}` returns 12.35
  `{{div (roundFloor (mult  12.3456 100)) 100}}` returns 12.34

Member

Function	Description
`getTargetPermissionsIn` memberID channelID	Returns target's permissions in the given channel
	Edits triggering user's nickname, argument has to

`editNickname` "newNick"	be of type *string*. YAGPDB's highest role has to be above the highest role of the member.
`hasPermissions` arg	Returns true/false on whether triggering user has the permission bit *int64* that is also set in .Permissions.
`getMember` mention/userID	Function returns Member object having above methods. `{{(getMember .User.ID).JoinedAt}}` is the same as `{{.Member.JoinedAt}}`
`onlineCount`	Returns the count of online users/members on current server.
`targetHasPermissions` memberID arg	Returns true/false on whether targeted member ha the permission bit *int64*.

Mentions

Function	Description
`mentionEeveryone`	Mentions `@everyone`.
`mentionHere`	Mentions `@here`.
`mentionRoleID` roleID	Mentions the role found with the provided ID.
`mentionRoleName` "rolename"	Mentions the first role found with the provided name (case-insensitive).

There is also .Mention method available for channel, role, user structs/objects.

Mentions section's snippets:

- `<@{{.User.ID}}>` Outputs a mention to the user that called the command and is the same as `{{.User.Mention}}`
- `<@###########>` Mentions the user that has the ID ###### (See How to get IDs to get ID).
- `<#&&&&&&&&&&&>` Mentions the channel that has ID &&&&&& (See How to get IDs to get ID).
- `<@&##########>` Mentions the role with ID ######## (listroles command gives roleIDs). This is usable for example with `{{sendMessageNoEscape nil "Welcome to role <@&11111111...>"}}`. Mentioning that role has to be enabled server- side in Discord.

Message

Function	Description

`addMessageReactions` channel messageID emojis...	Same as `addReactions` or `addResponseReactions`, but can be used on any messages using its ID. `channel` can be either `nil`, channel's ID or its name. Example in section's Snippets.
`addReactions` "" "" ...	Adds each emoji as a reaction to the message tha triggered the command (recognizes Unicode emojis and `emojiName:emojiID`).
`addResponseReactions` "" "" ...	Adds each emoji as a reaction to the response message (recognizes Unicode emojis and `emojiName:emojiID`).
`complexMessage` "content" args "embed" args "file" args "filename" args	`complexMessage` creates a *so-called* bundle c different message fields for `sendMessage...` functions to send them out all together. Its arguments need to be preceded by predefined key `"content"` for regular text, `"embed"` for embed arguments created by `cembed` or `sdict` `"file"` for printing out content as a file with default name `attachment_YYYY-MM-DD_HH-MM-SS.txt` (max size 100 000 characters ca 100kB). `"filename"` lets you define custom file name if `"file"` is used with max length of 64 characters, extension name remains `txt`. Example in this section's Snippets.
`complexMessageEdit` "content" args "embed" args	Special case for `editMessage` function - either `complexMessage` is involved or works even with regular message. Has two parameters `"content"` and `"embed"` to edit regular text part or embed part. If `"embed"` is set to `nil`, it deletes whole embed. Example in this section's Snippets.
`deleteAllMessageReactions` channel messageID (emojis...)	Deletes all reactions pointed message has. `channel` can be ID, "name" or `nil`. `emojis` argument is optional and works like it's described for the function `deleteMessageReaction`.
`deleteMessage` channel messageID (delay)	Deletes message with given `messageID` from `channel`. Channel can be either `nil`, channel ID or its name. `(delay)` is optional and like following two delete functions, it defaults to 10 seconds, max being 1 day or 86400 seconds. Example in section's Snippets.
	Deletes reaction(s) from a message. `channel`

`deleteMessageReaction` channel messageID userID emojis...	can be ID, "name" or `nil`. `emojis` argument can be up to 10 emojis, synta is `emojiName` for Unicode/Discord's default emojis and `emojiName:emojiID` for custom emotes. Example: `{{deleteMessageReaction nil (index .Args 1) .User.ID " " " "}}` will delete current user's reactions with thumbsUp/Down emotes from current running channel's message which ID is given to commanc as first argument `(index .Args 1)`. Also usable with Reaction trigger
`deleteResponse` (delay)	Deletes the response after a certain time from optional `delay` argument (max 86400 seconds = 1 day). Defaults to 10 seconds.
`deleteTrigger` (delay)	Deletes the trigger after a certain time from optiona `delay` argument (max 86400 seconds = 1 day). Defaults to 10 seconds.
`editMessage` channel messageID newMessageContent	Edits the message in channel, channel can be either `nil`, channel's ID or "name". Light examp in section's Snippets.
`editMessageNoEscape` channel messageID newMessageContent	Edits the message in channel and has same logic in escaping characters as `sendMessageNoEscape`.
`getMessage` channel messageID	Returns a Message object. `channel` can be either its ID, name or nil for triggering channel.
`pinMessage` channel messageID	Pins a message by its ID in given channel. `channel` can be either its ID, name or nil for triggering channel. Can be called 5 times.
`sendDM` "message here"	Sends the user a direct message, only one DM ca be sent per custom command (accepts embed objects). YAG will only DM triggering user.
`sendMessage` channel message	Sends `message (string or embed)` in `channel`, channel can be either `nil`, the channel ID or the channel's "name".
`sendMessageNoEscape` channel message	Sends `message (string or embed)` in `channel`, channel can be either `nil`, the channel ID or the channel "name". Doesn't escap mentions (e.g. role mentions or @here/@everyone).
`sendMessageNoEscapeRetID` channel	Same as `sendMessageNoEscape`, but also

message	returns messageID to assigned variable for later use.
`sendMessageRetID` channel message	Same as `sendMessage`, but also returns messageID to assigned variable for later use. Example in section's Snippets.

Message section's snippets:

- Sends message to current channel `nil` and gets messageID to variable `$x`. Also adds reactions to this message. After 5 seconds, deletes that message. > `{{$x := sendMessageRetID nil "Hello there!"}} {{addMessageReactions nil $x " " " "}} {{deleteMessage nil $x 5}}`
- To demonstrate `sleep` and slightly also `editMessage` functions. > `{{$x := sendMessageRetID nil "Hello"}} {{sleep 3}} {{editMessage nil $x "There"}} {{sleep 3}} {{sendMessage nil "We all know, that"}} {{sleep 3}} YAGPDB rules!`
- To demonstrate usage of `complexMessage` with `sendMessage`. `{{sendMessage nil (complexMessage "content" "Who rules?" "embed" (cembed "description" "YAGPDB of course!" "color" 0x89aa00) "file" "Here we print something nice - you all are doing awesome!")}}`
- To demonstrate usage of `complexMessageEdit` with `editMessage`. `{{$mID := sendMessageRetID nil (complexMessage "content" "You know what is..." "embed" (cembed "title" "FUN!?" "color" 0xaa8900))}} {{sleep 3}} {{editMessage nil $mID (complexMessageEdit "embed" (cembed "title" "YAGPDB!" "color" 0x89aa00) "content" "Yes, it's always working with...")}}{{sleep 3}}{{editMessage nil $mID (complexMessageEdit "embed" nil "content" "Embed deleted, goodbye YAG!")}}{{deleteMessage nil $mID 3}}`

Miscellaneous

> ⓘ `if`, `range`, `try-catch`, `while`, `with` actions are all covered here.

Function	Description
`adjective`	Returns a random adjective.
`cembed` "list of embed values"	Function to generate embed inside custom command. Mo in-depth here.
`createTicket` author topic	Creates a new ticket with the author and topic provided. Covered in its own section here.

`cslice`, `sdict`	These functions are covered in their own section here.
`dict` key1 value1 key2 value2 ...	Creates an unordered collection of key-value pairs, a dictionary so to say. The number of parameters to form key value pairs must be even. Example here. Keys and values can be of any type. Key is not restricted to *string* only as in case with `sdict`. `dict` also has helper methods .Del, .Get, .HasKey and .Set and they function the same way as `sdict` ones discussed here.
`exec` "command" "args" "args" "args" ...	Executes a YAGPDB command (e.g. roll, kick etc) in a custom command. Exec can be run max 5 times per CC. If real command returns an embed - `exec` will return raw data of type embed, so you can use embed fields for better formatting - e.g. `{{$resp := exec "whois"}} {{$resp.Title}} Joined at > {{(index $resp.Fields 4).Value}}` will return the title (username#discriminator) and "Joined at" field's value from `whois` command. **NB!** This will not work for commands with paginated embed returns, like `un/nn` commands! exec syntax is `exec "command" arguments` - this means you format it the same way as you would type the command regularly, just without the prefix, e.g. if you want to clear 2 messages and avoiding the pinned message > `{{exec "clear 2 -nopin"}}`, where `"command"` part is whole `"clear 2 -nopin"`. If you change that number inside CC somewhere then you have to use `arguments` part of exec formatting > `{{$x := 2}} {{exec "clear" $x "-nopin"}}` Here `"clear"` is the `"command"` and it is followed by `arguments`, one variable `$x` and one string `"-nopin"`. Last example is the same as `{{exec (joinStr " " "clear" $x "-nopin")}}` (also notice the space in `joinStr` separator).
`execAdmin` "command" "args" "args" "args" ...	Functions same way as `exec` but effectively runs the command as the bot user (YAGPDB). This has essentially the same effect as if a user with the same permissions and roles as YAGPDB ran the command: for example, if YAGPDB had ban members permission but the user which ran the command did not, `{{exec "ban" 12345}}` would error due to insufficient permissions but `{{execAdmin "ban" 12345}}` would succeed.
`execTemplate` "template" data	Executes the associated template, optionally with data. A more detailed treatment of this function can be found in the

	Associated Templates section
`hasPrefix` string prefix	`hasPrefix` tests whether the given `string` begins with `prefix` and returns *bool*. Example > `{{hasPrefix "YAGPDB" "YAG"}}` returns `true`.
`hasSuffix` string suffix	hasSuffix tests whether the given `string` ends with `suffix` and returns *bool*. Example > `{{hasSuffix "YAGPDB" "YAG"}}` returns `false`.
`humanizeThousands` arg	This function places comma to separate groups of thousands of a number. `arg` can be *int* or *string*, has to b a whole number, e.g. `{{humanizeThousands "1234567890"}}` will return `1,234,567,890`.
`in` list value	Returns *bool* true/false whether case-sensitive value is in list/*slice*. `{{ in (cslice "YAGPDB" "is cool") "yagpdb" }}` returns `false`.
`index` arg ...keys	Returns the result by indexing its first argument with following arguments. Each indexed item must be a *map*, *slice* or *array.* Indexed *string* returns value in *uint8.* Example: `{{index .Args 1}}` returns first argument after trigger which is always at position 0. More than one positional keys can be used, in pseudo-code: `index X 0 1` is equivalent to calling `index (index X 0) 1`
`inFold` list value	Same as `in`, but is case-insensitive. `{{inFold (cslice "YAGPDB" "is cool") "yagpdb"}}` returns `true`.
`kindOf` value (flag)	This function helps to determine what kind of data type we are dealing with. `flag` part is a *bool* and if set as **true** (**false** is optional) returns the value where given `value` points to. Example: `{{kindOf cembed false}}` `{{kindOf cembed true}}` will return `ptr` and `struct`.
`len` arg	Returns the integer length of its argument. arg can be an *array*, *slice*, *map*, or *string.* `{{ len (cslice 1 2 3) }}` returns `3`.
`noun`	Returns a random noun.
	Checks the arguments for a specific type. Has methods

`parseArgs` required_args error_message `...carg`	`.Get` and `.IsSet`. `carg "type" "name"` is required by `parseArgs` an it defines the type of argument for `parseArgs`. More in depth here and an example in Custom Command Examples.
`sendTemplate` channel templateName data	Function sends a formulated template to another channel and returns sent response's messageID. Channel is like always either name, number or nil; and returns messageID. Example: `{{define "logsTemplate"}}This tex will output on different channel, you car also use functions like {{currentTime}}. {{.TemplateArgs}} would be additional dat sent out. {{end}}` Now we call that "logs" in the same custom command. `{{sendTemplate "logs" "logsTemplate" "YA rules!"}}.` Template definitions are discussed here.

Role functions

Function	Description
`addRoleID` roleID	Adds the role with the given ID to the user that triggered the command (use the `listroles` command for a list of roles).
`addRoleName` roleName	Adds the role with given name to the user that triggered the command (use the `listroles` command for a list of roles).
`getRole` role	Returns a role object of type **discordgo.Role.* `role` can be either role's ID or role's name.

`giveRoleID` userID roleID	Gives a role by ID to the target.
`giveRoleName` userID "roleName"	Gives a role by name to the target.
`hasRoleID` roleID	Returns true if the triggering user has the role with the specified ID (use the listroles command for a l of roles).
`hasRoleName` "rolename"	Returns true if the triggering user has the role with the specified name (case-insensitive).
`removeRoleID` roleID (delay)	Removes the role with the given ID from the user that triggered the command (use the listroles command for a list of roles). `Delay` is optional argument in seconds.
`removeRoleName` roleName (delay)	Removes the role with given name from the user that triggered the command (use the listroles command for a list of roles). `Delay` is optional argument in seconds.
`roleAbove` role1 role2	`roleAbove` compares two role objects e.g. `getRole` return and gives `true/false` value `role1` positioned higher than `role2` or not.
`setRoles` userID roles	Overwrites the roles of the given user using the slice of roles provided, which should be a slice of role IDs. IDs can be ints or strings. Example: `{{setRoles .User.ID cslice}}` would clear the roles of the triggering user.
`takeRoleID` userID roleID (delay)	Takes away a role by ID from the target. `Delay` i optional argument in seconds.
`takeRoleName` userID "roleName" (delay)	Takes away a role by name from the target. `Delay` is optional argument in seconds.
`targetHasRoleID` userID roleID	Returns true if the given/targeted user has the role with the specified ID (use the listroles command fo a list of roles). Example in section's Snippets.
`targetHasRoleName` userID "roleName"	Returns true if the given/targeted user has the role with the specified name (case-insensitive).

String manipulation

> ⓘ All regexp functions are limited to 10 different pattern calls per CC.

Function	Description
`joinStr` "separator" "str1" (arg1)(arg2) "str2" ...	Joins several strings into one, separated by the fir argument `"separator"`, example: `{{joinStr "" "1" "2" "3"}}` returns `123`. Also if functions have *string, []string* or easily convertible return, they can be used inside `joinStr` e.g. `{{joinStr "" "Let's calculate " (add (mult 13 3) 1 2) ", was returned at " (currentTime.Format "15:04") "."}}`
`lower` "string"	Converts the string to lowercase.
`print`, `printf`, `println`	These are GO template package's predefined functions and are aliases for fmt.Sprint, fmt.Sprintf and fmt.Sprintln. Formatting is also discussed her `printf` cheat sheet here. `printf` is usable for example to determine the type of the value > `{{printf "%T" currentTime}}` outputs `currentTime` functions output value type of `time.Time`. In many cases, `printf` is a great alternative to `joinStr` for concatenate strings.
`reFind` "regex" "string"	Compares "string" to regex pattern and returns firs match. `{{reFind "AG" "YAGPDB is cool!"}}` returns `AG` (regex pattern is case sensitive).
`reFindAll` "regex" "string" (count)	Adds all regex matches from the "string" to a *slice* Example in section's Snippets. Optional `count` determines how many matches are made. **Example:** `{{reFindAll "a*" "abaabaccadaaae" 4}}` would return `[a aa a ].`
`reFindAllSubmatches` "regex" "string" (count)	Returns whole-pattern matches and also the sub-matches within those matches as *slices* inside a *slice*. `{{reFindAllSubmatches "(?i)y([a-z]+)g" "youngish YAGPDB"}}` returns `[[young oun] [YAG A]]` (regex pattern here is case insensitive). Optional `count` works the same way as for `reFindAll`. So example above with `count` set to 1 would return `[[young oun]]`.
	`reQuoteMeta` returns a string that escapes all

`reQuoteMeta` "string"	regular expression metacharacters inside the argument text; the returned string is a regular expression matching the literal text. Example in package documentation.
`reReplace` "regex" "string1" "string2"	Replaces "string1" contents with "string2" at rege match point. `{{reReplace "I am" "I am cool!" "YAGPDB is"}}` returns `YAGPDB i cool!` (regex pattern here is case sensitive). Inside "string2" dollar-sign, $ with numeric name like $1 or ${1} are interpreted as referrals to the submatches in "regex" pattern, so for instance $1 represents the text of the first submatch. To insert literal $ in the output, use $$.
`reSplit` "regex" "string" (count)	`reSplit` slices `string` into substrings separated by the `regex` expression and returns *slice* of the substrings between those expression matches. The optional `count` determines the number of substrings to return. If `count` is negative number the function returns all substring if 0 then none. If `count` is bigger than 0 it returns at most n substrings, the last substring being the unsplit remainder. **Example:** `{{ $x := reSplit "a" "yagpd has a lot of fame" 5}}` `{{$x}} {{index $x 3}}` would return `[y gpdb h s lot of f me]` and `lot of f.`
`slice` arg integer (integer2)	Function's first argument must be of type *string* or *slice*. Outputs the `arg` after cutting/slicing off integer (numeric) value of symbols (actually starting the string's index from integer through integer2) - e.g. `{{slice "Fox runs" 2}}` outputs `x runs` When using also integer2 - e.g. `{{slice "Fox runs" 1 7}}`, it outputs `ox run`. For slicing whole arguments, let's say words, see example in section's Snippets. This `slice` function is not the same as basic dynamically-sized *slice* data type discussed in thi reference doc. Also it's custom, not having 3-indices as the default one from text/template package.
	Splits given `"string"` to substrings separated by `"sepr"` arg and returns new *slice* of the

`split` "string" "sepr"	substrings between given separator e.g. `{{spli "YAG, is cool!" ","}}` returns `[YAG  is cool!]` *slice* where `YAG` is at `index` position and `is cool!` at `index` position 1. Example also in section's Snippets.
`title` "string"	Returns the string with the first letter of each word capitalized.
`trimSpace` "string"	Returns the string with all leading and trailing whi space removed.
`upper` "string"	Converts the string to uppercase.
`urlescape` "string"	Escapes the *string* so it can be safely placed insic a URL path segment - e.g. "Hello, YAGPDB!" becomes "Hello%2C%20YAGPDB%21" There's also predefined template package functio `urlquery` which is covered here.

> Special information we can include in the string is *escape sequences*. Escape sequences are two (or more) characters, the first of which is a backslash `\`, which gives the remaining characters special meaning - let's call them metacharacters. The most common escape sequence you will encounter is `\n`, which means "newline".

> With regular expression patterns - when using quotes you have to "double-escape" metacharacters starting with backslash. You can use backquotes/ticks to simplify this: `{{reFind "\\d+" (toString 42)}}` versus `{{reFind `\d+` (toString 42)}}`

String manipulation section's snippets:

- `{{$args:= (joinStr " " (slice .CmdArgs 1))}}` Saves all the arguments except the first one to a variable `$args`.
- To demonstrate usage of `split` function. > `{{$x := "Hello, World, YAGPDB, here!"}} {{range $k, $v := (split $x ", ")}}Word {{$k}}: __{{$v}}__ {{end}}`
- To demonstrate usage of `reFindAll`. > `Before regex: {{$msg := "1 YAGPDB and over 100000 servers conquered."}} {{$re2 := reFindAll "[0-9]+" $msg}} {{$msg}}` `After regex matches: {{println "Only" (index $re2 0) "YAGPDB and already" (index $re2 1) "servers captured."}}`

Time

Function	Description
`currentTime`	Gets the current time, value is of type *time.Time* which can be used in a custom embed. More info here.
`formatTime` Time ("layout arg")	Outputs given time in RFC822 formatting, first argument `Time` shows it needs to be of type *time.Time*, also with extra layout if second argument is given - e.g. `{{formatTime currentUserCreated "3:04PM"}}` would output `11:22AM` if that would have been when user was created. Layout argument is covered here.
`humanizeDurationHours`	Returns given integer (whole number) or *time.Duration* argument in nanoseconds in humar readable format - as how long it would take to get towards given time - e.g. `{{humanizeDurationHours 9000000000000000000}}` returns `285 years 20 weeks 6 days and 16 hours`. More in Snippets.
`humanizeDurationMinutes`	Same as `humanizeDurationHours`, this time duration is returned in minutes - e.g. `{{humanizeDurationMinutes 3500000000000}}` would return `58 minutes`
`humanizeDurationSeconds`	Same as both humanize functions above, this time duration is returned in seconds - e.g. `{{humanizeDurationSeconds 3500000000000}}` would return `58 minutes and 20 seconds`.
`humanizeTimeSinceDays`	Returns time passed since given argument of type *time.Time* in human readable format - e.g. `{{humanizeTimeSinceDays currentUserCreated}}.`
`loadLocation` "location"	Retruns value of type **time.Location* which can be used further in other golang's time functions, for example `{{currentTime.In (loadLocation "Asia/Kathmandu")}}` would return current time in Nepal. `location` is of type *string* and has to be in ZONEINFO syntax.
	Returns type *time.Time* object in UTC using giver

`newDate` year month day hour minute second (timezone)	syntax (all required arguments need to be of type *int*), for example > `{{humanizeDurationHours ((newDate 2059 1 2 12 34 56).Sub currentTime)}}` will give you how much time till year 2059 January 2nd 12:34:56. More Snippets. `timezone` is an optional argument of type *strin* which uses golang's LoadLocation function and ZONEINFO syntax. For example: `{{newDate 2020 4 20 12 34 56 "Atlantic/Reykjavik"}}` would return that time in GMT+0.
`snowflakeToTime` snowflake	Converts given snowflake to type *time.Time* e.g. using bot's ID `{{snowflakeToTime .BotUser.ID}}` returns `2016-07-17 15:17:19 +0000 UTC` for YAGPDB.
`weekNumber` time	Returns the week number as *int* of given argumen `time` of type *time.Time*. `{{weekNumber currentTime}}` would return the week numbe of current time.

> ⓘ Discord Timestamp Styles referenced here can be done using `print` function e.g. `{{print "<t:" currentTime.Unix ":F>"}}` for "Long Date/Time" formatting.

Time section's snippets:

- To demonstrate `humanizeDurationHours` and also how to parse a timestamp, output will be like `whois` command shows user's *join server age*. `{{humanizeDurationHours (currentTime.Sub .Member.JoinedAt.Parse)}}`
- To demonstrate `newDate` to get Epoch times. `{{$unixEpoch := newDate 1970 1 1 0 0 0}} in seconds > {{$unixEpoch.Unix}}` `{{$discordEpoch := newDate 2015 1 1 0 0 0}} in seconds > {{$discordEpoch.Unix}}`

Type conversion

Function	Description
	Traverses given `value` through MarshalJSON (more here) and returns it as type *string*. For example `{{json .TimeHour}}` outputs type *string*; before this

`json` value	`.TimeHour` was of type *time.Duration*. Basically it's goc to use if multistep type conversion is needed `(toString (toInt value) )` and certain parts of `cembed` neec
`structToSdict` struct	Function converts exported field-value pairs of a *struct* to a *sdict*. For example it is useful for editing embeds, rather than having to reconstruct the embed field by field manually. Exampe: `{{$x := cembed "title" "Something rules!" "color" 0x89aa00}} {{$x.Title}} {{$x = structToSdict $x}} {{ $x.Set "Title" "No, YAGPDB rules!!!" -}} {{$x.Title}} {{$x}}` will return No, YAGPDB rules and whole sdict-mapped *cembed*.
`toByte` "arg"	Function converts input to a slice of bytes - meaning *[]uint* `{{toByte "YAG€"}}` would output `[89 65 71 226 130 172]`. `toString` is capable of converting that slice back to *string*.
`toDuration`	Converts the argument, number or string to type *time.Duration* - more duration related methods here. Number represents nanoseconds. String can be with time modifier (second, minute, hour, day etc) `s, m, h, d, w, mo, y` ,without a modifier string will be converted to minutes. Usage: `(toDuration x)`. Example in section Snippets.
`toFloat`	Converts argument (*int* or *string* type of a number) to type *float64*. Usage: `(toFloat x)`. Function will return 0, if type can't be converted to *float64*.
`toInt`	Converts something into an integer of type *int*. Usage: `(toInt x)`. Function will return 0, if type can't be converted to *int.*
`toInt64`	Converts something into an *int64*. Usage: `(toInt64 x)` Function will return 0, if type can't be converted to *int64.*
`toRune` "arg"	Function converts input to a slice of runes - meaning *[]int3* `{{toRune "YAG€"}}` would output `[89 65 71 8364]`. These two functions - the one above, are good fo further analysis of Unicode strings. `toString` is capabl of converting that slice back to *string*.
`toString`	Has alias `str`. Converts some other types into a *string*. Usage: `(toString x)`.

Type conversion section's snippets:

- To demonstrate `toDuration`, outputs 12 hours from current time in UTC. `{{(currentTime.Add (toDuration (mult 12 .TimeHour))).Format "15:04"}}` is the same as `{{(currentTime.Add (toDuration "12h")).Format "15:04"}}` or `{{(currentTime.Add (toDuration 43200000000000)).Format "15:04"}}`

> **Tip:** You can convert a Unicode code point back to its string equivalent using `printf "%c"`. For example, `printf "%c" 99` would result in the string `c` as `99` is the Unicode code point for `c`. `printf` is briefly covered later on in the next section, further documentation can be found here. Cheat sheet here.

User

Function	Description
`currentUserAgeHuman`	The account age of the current user in more huma readable format.
`currentUserAgeMinutes`	The account age of the current user in minutes.
`currentUserCreated`	Returns value of type *time.Time* and shows when the current user was created.
`pastNicknames` userID offset	Same as `pastUsernames`.
`pastUsernames` userID offset	Returns a *slice* of type *[]*logs.CCNameChange* having fields .Name and .Time of previous 15 usernames and skips `offset` number in that lis `{{range pastUsernames .User.ID 0}}` `{{.Name}} - {{.Time.Format "Jan _2 2006"}}` `{{end}}`
`userArg` mention/userID	Function that can be used to retrieve .User object from a mention or userID. `{{(userArg .User.ID).Mention}}` mentions triggering user. Explained more in this section's snippets. Previous limit of 5 to this functions is no longer there.

User section's snippets:

`{{(userArg .Guild.OwnerID).String}}` this template's action-structure returns Guild/Server owner's username and discriminator as of type *string*. First, `userArg` function is given `.Guild.OwnerID` as argument (what it does, explained in templates section). The parentheses surrounding them make `userArg` function return `.User` as .User object which is handled further by `.String` method (ref. `.User.String`), giving a result like > `YAGPDB#8760`.