SEARCH

Ctrl + /

SECTIONS

GuidedTrack Manual

Research Guide

Function & Keyword API

Changelog

Sitemap

Status

Function & Keyword API

Text variables

`text.clean`

Returns a copy of the text with spaces and any other "whitespaces" (tabs, carriage return, ...) removed from the beginning and end.

```
>> myText = "          ...that was good!     "
>> cleanText = myText.clean

```

`text.count(textToCount)`

Returns the number of times that `textToCount` appears in a given text.

```
>> myText = "the smell of cookies in the oven is the best thing
```

`text.decode(encodingScheme)`

Decodes a text variable using the specified encoding scheme.

There are two schemes currently supported: "URL" and "JSON".

The "URL" scheme returns a copy of the given text where the percent-encoding escape sequences of characters are replaced with their normal representations.

```
>> urlParams = "?city=Gen%C3%A8ve&planet=The%20Earth"
>> decodedPar = urlParams.decode("URL")
-- "?city=Genève&planet=The Earth"

```

You can also use the "URL" scheme to insert double quotes or newlines into a text variable.

```
>> textWithQuotes = "Hello, %22world%22!".decode("URL")
-- Hello, "world"!

>> textWithNewlines = "Hello,%0Aworld!".decode("URL")
-- Hello,
-- world!

```

The "JSON" scheme converts a JSON-formatted text into a GuidedTrack value. This can be useful when you receive JSON from a service like ChatGPT and want to use it in GuidedTrack.

```
-- jsonText = '{"name":"John Doe","age":25}'
>> association = jsonText.decode("JSON")
-- { "name" -> "John Doe", "age" -> 25 }

```

See: `.encode`.

`text.encode(encodingScheme)`

Encodes a text variable using the specified encoding scheme.

There are two schemes currently supported: "URL" and "JSON". Currently the only scheme supported for text variables is "URL", which replaces non-ASCII characters with their percent encoding. This can be useful when providing parameters to a URL in a *goto keyword or a *service path.

```
>> urlParams = "?city=Genève&planet=The Earth"
>> encodedPar = urlParams.encode("URL")
-- "?city=Gen%C3%A8ve&planet=The%20Earth"

```

See: `.decode`.

`text.find(textToFind)`

Returns the first position in which the text `textToFind` appears in a given text.

```
>> myText = "Did you ever visit Paris?"
>> parisIndex = myText.find("Paris")

```

`text.lowercase`

Returns a copy of the text converted to lowercase.

```
>> myText = "Text with Capital Letters"
>> lowercaseText = myText.lowercase

```

`text.size`

Returns the number of characters in a text.

```
>> myText = "I love going to the movies"
>> textLenght = myText.size

```

`text.split(delimiter)`

Splits a text into chunks at every `delimiter` and returns the chunks in a collection.

```
>> fruitsILike = "pears and bananas and oranges and mangoes"
>> collectionOfFruits = fruitsILike.split(" and ")
-- collectionOfFruits = ["pears", "bananas", "oranges", "mangoes

```

`text.uppercase`

Returns a copy of the text converted to uppercase.

```
>> myText = "Beauty is Truth"
>> uppercaseText = myText.uppercase

```

Numeric variables

`number.round`

Rounds a number to the nearest integer. Optionally, a number of decimal places can be specified.

```
>> pi = 3.14159265359
>> piInteger = pi.round
>> piToFourDecimals = pi.round(4)

```

`number.[duration]`

Returns a duration of the relevant size. Valid time units are: seconds, minutes, hours, days, weeks, months, and years.

```
>> oneYear = 365.days
*wait: 5.seconds
>> threeAndAHalfMinutes = (3.5).minutes

```

Collections

`collection.add(elementToAdd)`

Adds `elementToAdd` to the end of a collection.

```
>> peopleIKnow = ["Peter", "Lisa"]
>> peopleIKnow.add("Roger")

```

`collection.combine(collectionToAdd)`

Adds the elements of `collectionToAdd` to the end of a given collection.

```
>> numbers = [1, 2, 3]
>> nextNumbers = [4, 5, 6]
>> numbers.combine(nextNumbers)
-- Now `numbers` = [1, 2, 3, 4, 5, 6]

```

By the way, the `.combine` method is useful when you need to make a copy or duplicate of a collection. For example, here's a case where we might want to shuffle a collection but keep a copy of the original, un-shuffled collection:

```
-- define the original collection
>> x = [1, 2, 3, 4, 5]
```

`collection.count(valueToCount)`

Returns the number of times that the `valueToCount` appears in the collection.

```
>> tripsLastYear = ["Amsterdam", "Madrid", "London", "Amsterdam"
>> timesInLondon = tripsLastYear.count("London")

```

`collection.erase(valueToErase)`

Removes all elements of a collection that match a given value.

```
>> tripsThisYear = ["Paris", "London", "Rome", "London", "Madric
>> tripsThisYear.erase("London")

```

See `.remove` for a comparison of the two methods.

`collection.find(valueToFind)`

Returns the first position in which the `valueToFind` appears in the collection.

`collection.insert(elementToAdd, position)`

Inserts `elementToAdd` at a given position of a collection.

```
>> peopleIKnow = ["Peter", "Lisa"]
>> peopleIKnow.insert("Roger", 2)

```

`collection.max`

Returns the maximum value in a collection of numbers.

```
>> dailyPushupsLastWeek = [110, 132, 123, 149, 162, 148, 135]
>> maximumDailyPushups = dailyPushupsLastWeek.max

```

`collection.mean`

Returns the mean value of a collection of numbers.

```
>> dailyPushupsLastWeek = [110, 132, 123, 149, 162, 148, 135]
>> averageDailyPushups = dailyPushupsLastWeek.mean

```

`collection.median`

Returns the median value of a collection of numbers.

```
>> dailyPushupsLastWeek = [110, 132, 123, 149, 162, 148, 135]
>> medianDailyPushups = dailyPushupsLastWeek.median

```

collection.min

Returns the minimum value in a collection of numbers.

```
>> dailyPushupsLastWeek = [110, 132, 123, 149, 162, 148, 135]
>> minimumDailyPushups = dailyPushupsLastWeek.min

```

---

collection.remove(position)

Removes the element at a given position of a collection.

```
>> friends = ["Liz", "Ana", "Sam"]
>> friends.remove(2)

```

Note that `.remove` differs from `.erase`. The `.remove` method allows you to remove a single value by specifying a position, whereas the `.erase` method allows you to remove all instances of a specified value.

Note also that these two methods perform analogous actions in associations; i.e., `.remove` removes a single key-value pair from an association by specifying a position, and `.erase` removes all relevant key-value pairs by specifying a value.

---

collection.shuffle

Shuffles the elements in a collection.

```
>> myCollection = [1, 2, 3, 4, 5]
>> myCollection.shuffle

```

---

collection.size

Returns the number of elements in a collection.

```
>> tripsLastYear = ["Amsterdam", "Madrid", "London", "Amsterdam"
>> totalTrips = tripsLastYear.size

```

`collection.sort(direction)`

Sorts a collection where `direction` is either "increasing" or "decreasing".

```
>> friends = ["Liz", "Ana", "Sam"]
>> friends.sort("increasing")

```

`collection.unique`

Returns a copy of the collection without duplicate values (i.e., returns the set of values in the collection).

```
>> tripsLastYear = ["Amsterdam", "Madrid", "London", "Amsterdam"
>> citiesVisited = tripsLastYear.unique

```

Associations

`association.encode(encodingScheme)`

Encodes a variable using the specified encoding scheme.

Currently, the only supported scheme for associations is "JSON". When applied, converts an association into a JSON-formatted text.

```
>> association = { "name" -> "John Doe", "age" -> 25 }
>> jsonText = association.encode("JSON")
-- '{"name":"John Doe","age":25}'

```

See: `.decode`.

`association.erase(valueToErase)`

Removes all elements in an association whose value match a given one.

```
>> favFoods = {"Pete" -> "Cake", "Lucas" -> "Soup", "Beth" -> "P
>> favFoods.erase("Cake")

```

See `.remove` for a comparison of the two methods.

`association.keys`

Returns a collection of the keys of an association.

```
>> phoneBook = {"Lucas" -> 223142, "Mara" -> 772632, "Sue" -> 88
>> peopleInPhoneBook = phoneBook.keys

```

`association.remove(keyToRemove)`

Removes a key and its associated value from an association.

```
>> phoneBook = {"Lucas" -> 223142, "Mara" -> 772632, "Sue" -> 88
>> phoneBook.remove("Lucas")

```

Note that `.remove` differs from `.erase`. The `.remove` method allows you to remove a single key-value pair by specifying a key, whereas the `.erase` method allows you to remove all relevant key-value pairs by specifying a value.

All of the keys of an association are unique; i.e., there can't be two

keys of the same name in an association. But there *can* be multiple copies of the same value in an association. So, if you wanted to remove a particular key from an association, then there would only one key-value pair to remove, and you'd use the `.remove` method to remove it; but if you wanted to remove a particular value from an association, then there could potentially be a lot of key-value pairs to remove, and you'd use the `.erase` method to remove them.

Note also that these two methods perform analogous actions in collections; i.e., `.remove` removes a single item from a collection by specifying a position, and `.erase` removes all instances of a specified value from a collection.

Datetime and duration variables

`calendar::date`

Returns a datetime. Optionally, an association argument can be accepted. If no argument is given, then the current date is returned.

```
>> todaysDate = calendar::date
>> otherDate = calendar::date({"year" -> 2021, "month" -> 12, "d
>> wednesdayThisWeek = calendar::date({"weekday" -> "Wednesday"}

```

`calendar::now`

Returns the current datetime.

```
>> rightNow = calendar::now

```

`calendar::time`

Returns a datetime. Optionally, an association argument can be

accepted. If no argument is given, then the current time is returned.

```
>> currentTime = calendar::time
>> otherTime = calendar::time({"hour" -> 13, "minute" -> 42})

```

`duration.to(timeUnit)`

Converts a time duration variable into a different time unit. Valid time units are: seconds, minutes, hours, days, weeks, months, and years

```
>> timeElapsed = calendar::date - calendar::date({ "month" -> 12
>> daysElapsed = timeElapsed.to("days")

```

Any variable type

`any.text`

Returns the value of the variable converted to text.

```
>> today = calendar::date
>> textDate = today.text

```

`any.type`

Returns the variable's type (e.g., number, collection, association, etc.).

```
>> variable = 43.22
>> typeOfVariable = variable.type

```

Manual data management

You can set a specific column's value in your program's CSV file using `data::store`:

```
>> data::store(someColumnName, someValue)

```

For example:

```
*question: What's your email address?
    *save: email

>> data::store("Email address", email)

```

See this page for more information about how using `data::store` differs from simply assigning a value to a variable.

Keywords

*audio: audioUrl

Embeds an audio file into the page.

Sub-keywords:

- `*start: yesOrNo` = Determines whether or not the audio can auto-play; defaults to "no". This sub-keyword is optional.
- `*hide: yesOrNo` = Determines whether or not the audio player controls are visible; defaults to "yes". This sub-keyword is optional.

```
Here's episode 021 of the /Clearer Thinking/ podcast:

*audio: https://dts.podtrac.com/redirect.mp3/1227185674.rsc.cdn7
    *start: yes
    *hide: no

```

There's one important caveat to setting `*start: yes`, and it's this: some browsers block the auto-playing of audio and video. And

browsers aren't very consistent (at least not at the time of this writing) about how they let users enable or disable auto-playing on a global or individual page basis. One possible work-around is this: If you ask users play an audio file *manually*, then all subsequent audio files can be auto-played. That's because the browser assumes that their manual playing of the first audio file counts as permission to hear more audio files from the same web page. Here's how that might look in practice:

```
*label: start

At the end of your meditation session, we'll play a chime sounc
    that the volume of your speakers or headphones is at the ri

/NOTE: It's important for you to play this audio file at least
    auto-play future audio files, like the chime sound we play

*audio: https://1227185674.rsc.cdn77.org/aud/demo/bell.mp3
    *start: no
    *hide: no

*question: Did the chime sound play correctly?
    Yes
    No
        *goto: start

Now, meditate on something for 15 seconds. When the time is fir
    on the previous page.

*wait: 15.seconds

*clear

Meditation time is over now!

*audio: https://1227185674.rsc.cdn77.org/aud/demo/bell.mp3
    *start: yes
    *hide: yes

*button: Restart

*goto: start

```

*button: textToDisplay

Puts a button with specific text in a page.

Note that buttons are typically used to mark the end of a page, usually after a body of text. For example:

```
------------
-- PAGE 1 --
------------

Hello!

Here is some text. I'm so glad we're here, aren't you? There's

Okay, now I'm done.

*button: Next

------------
-- PAGE 2 --
------------

Here's some stuff on the next page.

*button: Yes, you're right!

```

In the above example, everything above `*button: Next` would be displayed on a single page, and the "Next" button would appear at the bottom. When users click the "Next" button, the program will continue onward to the "Here's some stuff..." page and place a "Yes, you're right" button at the bottom.

---

*chart: chartName

Displays a chart

Sub-keywords:

- `*data: myDataCollection` = Defines the data to be displayed;

must be a collection of collections (i.e., a two-dimensional collection). For bar charts, the collection must include pairs of labels and values, like: `*data: [["chocolate", 123], ["vanilla", 234]]`. For line and scatter charts, the collection must include pairs of x-values and y-values, like: `*data: [[23, 45], [67, 89]]`. This sub-keyword is required.

- `*type: chartType` = Specifies the type of chart. Options include "bar", "line", and "scatter". This sub-keyword is required.
- `*xaxis` = Defines the range of values to be displayed on the x-axis. Must include `*min` and `*max` sub-keywords. This sub-keyword is optional.
- `*yaxis` = Defines the range of values to be displayed on the y-axis. Must include `*min` and `*max` sub-keywords. This sub-keyword is optional.
- `*trendline` = Computes and draws a trend-line (i.e., the line of best fit) in scatter charts. This sub-keyword is optional.

```
*chart: My daily push-up count
    *type: scatter
    *data: [[0, 4], [1, 7], [2, 13], [3, 10], [4, 14], [5, 16],
    *trendline

```

---

*clear

Clears text that was kept on the page by `*maintain`.

```
*maintain: Please answer honestly!

*question: Have you ever stolen anything?
    Yes
    No

*question: Have you ever told a lie?
    Yes
    No

*clear

```

*component

Displays a bordered content box.

Sub-keywords:

- `*classes: class1, class2, class3, ...` = Defines a list of class names to be applied to the component. Possible class names include Bootstrap class names (like "alert-warning") or class names defined elsewhere (e.g., in `<style>` tags, in CSS files, etc.) This sub-keyword is optional.
- `*click` = Defines code (indented beneath) that will run if the user clicks on the component. This sub-keyword is optional.
- `*with` = Used in combination with `*click`, creates a local variable to generate different click handlers. Useful when a page has multiple components that are dynamically rendered. This sub-keyword is optional.

```
*component
    *classes: alert-info
    Hey! This is a blue info box!
    *click
        >> someVariable = someValue
        *goto: someLabel

```

Here is an example of the syntax when using the `*with` keyword in combination with `*click`. Follow this link for additional info on its usage.

```
*label: people
*for: person in people
    *component
        *header: {person["name"]}
        *with: person
        *click
            Name: {it["name"]}
            Age: {it["age"]}
            *button: Back
```

*database

Requests user info from the GuidedTrack database.

Sub-keywords:

- `*what: typeOfData` = Specifies the kind of data you're requesting. Currently, "email" is the only valid option. This sub-keyword is required.
- `*success` = Defines code to run if the request succeeds. Any data returned from the request will exist in a variable called `it` and will only be available in this block. This sub-keyword is required.
- `*error` = Defines code to run if the request fails. Error information will exist in a variable called `it` (with properties "reason" and "message") and will only be available in this block. This sub-keyword is required.

```
*database: request
    *what: email
    *success
        >> emailAddress = it["email"]
    *error
        >> errorReason = it["reason"]
        >> errorMessage = it["message"]

```

*email

Sends an email immediately or at a specified time.

Sub-keywords:

- `*subject: subjectLine` = Defines the subject of the email. This sub-keyword is required.
- `*body` = Defines the body of the email. This sub-keyword is

required.

- `*to: emailAddress` = Defines an address to which the email should be sent. If this keyword is omitted, then the email will be sent to the logged-in user or merely ignored if no user is logged in. This sub-keyword is optional.
- `*when: datetime` = Defines when the email should be sent. For recurring emails, it defines when the *first* email should be sent. Note that the specified datetime *must* be in the future (i.e., it can't be in the past, of course, but it also can't be `calendar::now`). This sub-keyword is optional.
- `*every: duration` = Defines how often recurring emails should be sent. Note that emails cannot be sent more frequently than once per day. This sub-keyword is optional.
- `*until: datetime` = Defines when recurring emails should stop. This sub-keyword is optional.
- `*identifier: someName` = Defines a name by which a group of recurring emails should be known, which is useful for cancelling upcoming recurring emails with `*cancel: someName`. This sub-keyword is optional.
- `*cancel: someName` = Cancels a scheduled email with a given identifier. This sub-keyword is optional.

```
>> wantsToSubscribe = "no"
>> wantsToUnsubscribe = "no"

*question: Would you like to modify your subscription?
    Yes, I want to subscribe to daily reminders to brush my tee
        >> wantsToSubscribe = "yes"
    Yes, I want to unsubscribe from the daily reminders to brus
        >> wantsToUnsubscribe = "yes"
    No, I want to leave.

*if: wantsToSubscribe = "yes"
    *login
        *required: yes

    *email
        *identifier: toothBrushingEmailSubscription
        *when: calendar::now + 3.seconds
        *every: 1.days
        *until: calendar::now + 1.year
```

Note that the unsubscription block in the example above is a little redundant because users can actually unsubscribe at any time by clicking the unsubscribe link that's automatically included in the email itself. But we included it to show how you, the developer, might cancel recurring emails on behalf of a user.

`*events` 🔗

Defines named events that can be executed using `*trigger` or jQuery's `trigger` function.

Sub-keywords:

- `*startup` = Defines an event to be run any time the program is loaded. This sub-keyword is optional.

Data sent to events is temporarily stored in a variable called `it` that is only available inside the event block.

`*goto` keywords used inside an event block must always include a `*reset` indented beneath. There's a subtle point here about where the program execution goes when an event is triggered. Take, for

example, the functionality of a standard `*goto` / `*label` combo: when calling `*goto`, the program moves to the relevant label and continues (in order) from there. So, if we call `*goto: myLabel`, and if `*label: myLabel` appears on line 113, then execution would move to line 113 and continue in order from there. We mention this example because events are sort of like `*goto` in the way that they move the program execution. If we call `*trigger: someEvent`, and if "someEvent" is defined on line 2, then execution would move to line 2 and continue in order from there (though skipping over other events). When the `*events` block is finally exited, execution still continues in order — meaning that the program basically starts over (because it moves to the next line of code *after* the `*events` block)! This is usually an undesired behavior, so it's very common to use `*goto` inside of an event block to prevent the program from starting over.

To illustrate this point, consider this example:

```
1  *events
2      myCoolEvent
3          >> x = 5
4
5  *trigger: myCoolEvent
6  Here's the value of x: {x}
7  *button: Okay
8
```

This program is actually badly designed and would cause the browser to freeze and/or crash (much like an out-of-control `*while` loop) because the `*trigger: myCoolEvent` on line 5 causes execution to move up to line 2 (where "myCoolEvent" is defined); and then, when "myCoolEvent" has finished running, execution continues from line 4; and since there's nothing on line 4, it progresses to line 5, where it once again triggers "myCoolEvent"; and so on forever.

So, a common pattern to avoid this behavior involves two steps:

1. Trigger an event and then wait indefinitely.
2. In the event definition, specify a label to which the program execution should move once the event block has finished running.

Here's how the above example would look after this pattern has been implemented:

```
*events
    myCoolEvent
        >> x = 5
        *goto: afterMyCoolEventLabel
            *reset

*trigger: myCoolEvent
*wait

*label: afterMyCoolEventLabel
Here's the value of x: {x}
*button: Okay

```

In embedded programs, events can be triggered with jQuery, like this:

```
const dataToSend = { name: "John" }
$(window).trigger("myCoolEvent", dataToSend)

```

In the above example, the data defined in `dataToSend` would be available in the "myCoolEvent" block as the `it` variable. For example:

```
*events
    myCoolEvent
        >> name = it["name"]

```

---

*startup

For most events, you'll use a custom name, like `myCoolEvent`, as in the examples above. However, there's one special event that has a fixed name: `*startup`. This event is run every time the program is loaded, which happens (1) when a user runs the program for the first time and (2) any time they refresh the page. For example:

```
*events
    *startup
        You should see this text every time the program loads!
```

Once again, there's a subtle point to notice here. As we mentioned in the first part of this section, triggering custom events causes the program execution to move to the line where the event is defined and then continue in order from there, moving to the very next line below the `*events` block once the event code has finished executing. Surprisingly, though, `*startup` does *not* exhibit this behavior. For example, if your program has a `*startup` event defined, and if a user makes it half-way through your program and then refreshes the page, then the `*startup` event will be run before anything else — but then the program execution will move to where the user last left off, *not* to the next line after the `*events` block!

`*experiment: experimentName` 🔗

Defines an experiment by name and assigns a user (permanently) to an experiment group.

`*experiment` is similar to `*randomize` but differs in two ways:

1. An experiment guarantees that an equal number of users will encounter each block of code.
2. An `*everytime` keyword *cannot* be applied to experiments because it's expected that a user will be permanently assigned to a particular experiment group.

Optionally, a name can be applied to each group using `*group: groupName`. We highly recommend using this option since it makes it easier to analyze individual groups later.

```
1  *experiment: Sleep Experiment
2      *group: Control
3          Before bedtime each night, do nothing at all.
```

---

*for: indexOrKey, value in variable 🔗

Loops through the elements of a collection, association, or string (text) variable.

`indexOrKey` can be ommited in all three cases, as you can see in the example below where we just retrieve the values of the elements in `favRestaurants`:

```
>> favRestaurants = ["Cracker Barrel", "Olive Garden", "Panera B

*for: restaurant in favRestaurants
    *question: What is your favorite food at {restaurant}?

```

Iterate over a collection:

```
>> names = ["Alice", "Bob", "Charlie"]

*for: index, name in names
    {name}, you're #{index}.

```

Iterate over an association:

```
Here are some important female scientists in history:

>> femaleScientists = {"Physics" -> ["Marie Curie", "Maria Goep
    "Biology" -> ["Rachel Carson", "Rita Levi-Montalcini", "Bar
    Elion", "Rosalind Franklin"], "Medicine" -> ["Elizabeth Bla
    ["Vera Rubin"]}

*for: field, scientists in femaleScientists
    *{field}*

    *for: scientist in scientists
        {scientist}
```

Iterate over a text variable:

```
*page
    >> acronym = "CIA"

    In the acronym "{acronym}"...

    *for: character in acronym
        *question: The "{character}" stands for:

```

---

*goto: labelName

Instructs the program to return to a specific label.

Sub-keywords:

- `*reset` = Resets the navigation stack in places where navigation potentially becomes especially convoluted. Also, note that a `*reset` keyword is required in `*goto` statements that appear under `*events`. This sub-keyword is optional.

```
*events
    someEvent
        >> result = it["result"]
        *goto: someLabel
            *reset

```

---

*group

Defines a block of code to run, usually used in the context of `*randomize` or `*experiment`. Optionally, a name can be applied to the group using: `*group: groupName`.

```
The following items will be displayed in a random order:
```

See the `*randomize` and `*experiment` keywords for examples of using `*group`.

---

*html

Inserts arbitrary HTML code into the page.

Virtually any HTML can be injected into the page *except* `<script>` tags. The HTML content to be injected must be indented beneath `*html`.

```
*html
    <table align="center">
        <thead>
            <tr><th>Name</th><th>Age</th></tr>
        </thead>
        <tbody>
            <tr><td>Sue</td><td>29</td></tr>
        </tbody>
    </table>

```

Because GuidedTrack uses Bootstrap CSS (with its own styles applied on top), it's possible to apply Bootstrap classes to elements in the `*html` block. It's also possible to style your own HTML using `<style>` tags. For example:

```
*html
    <style>
```

---

*if: condition

Runs a block of code if `condition` is a true statement.

```
*if: age < 18
    You're not eligible to take this survey!
    *quit

```

---

*image: imageUrl

Inserts an image into the page.

Sub-keywords:

- `*caption: textToDisplay` = Defines a description to be displayed beneath the image. This sub-keyword is optional.
- `*description: altText` = Defines a text to display if the image fails to load. This sub-keyword is optional.

```
Here's a picture of a cat:

*image: https://images.unsplash.com/photo-1511044568932-338cba0a
```

*label: labelName

Declares a place in the code to which the program can return at any time.

```
*label: startLabel

*question: Try to guess the number I'm thinking of:
    7
        Nope! Try again!
        *goto: startLabel
    13
        Nope! Try again!
        *goto: startLabel
    42
        Correct!

```

*list

Inserts a list into the page. By default, lists are unordered. Use `*list: ordered` for an ordered list. Use `*list: expandable` for a list of collapsible / expandable boxes.

Here's an unordered list:

```
*list
    Lions
    Tigers
    Bears

```

Here's an ordered list:

```
*list: ordered
    Lions
```

Here's a list of expandable boxes:

```
*list: expandable
    Lions
        Lions are scary!
    Tigers
        Tigers are stripey!
    Bears
        Bears are cool.

```

The above example is rendered like this (though the boxes will be collapsed by default):

`*login` 🔗

Asks the user to log in using a GuidedTrack account. If they don't have an account already, they'll be prompted to create one.

Sub-keywords:

- `*required: yesOrNo` = Indicates whether or not users will be required to log in. This sub-keyword is optional.

```
*login
    *required: yes

```

`*maintain: someText`

Keeps text in a gray box at the top of the page across multiple pages (until a `*clear`).

```
*maintain: Please answer honestly!

*question: Have you ever stolen anything?
    Yes
    No

*question: Have you ever told a lie?
    Yes
    No

*clear

```

`*navigation`

Creates a navigation bar.

Sub-keywords:

- `*name: navBarName` = Assigns the navbar a name to ensure that existing runs are not broken when changes are made to it. It is **highly recommended** to always name navbars. This sub-keyword is optional.

Items to be listed in the navbar must be indented beneath

`*navigation`. Under each navbar item, indent code that should be executed when a user clicks on that navbar item.

Optionally, each navbar item can have a Font Awesome `*icon: fontawesome-icon-class`.

```
*navigation
    *name: myToolNavBar
    Home
        *icon: fa-home
        *goto: homeLabel
    Quiz
        *icon: fa-question-circle
        *goto: quizLabel
    Settings
        *icon: fa-cog
        *goto: settingsLabel

```

---

*page

Creates a page of content.

Everything indented under `*page` will all be included together on a single "page" of content. Note, though, that there are some subtleties around what's allowed beneath a `*page`. For example, a common way of asking questions is to write code that should execute depending on the answer a users gives. For example:

```
*question: What's your favorite kind of ice cream?
    chocolate
        >> isCool = "yes"
    vanilla
        >> isCool = "no"
    strawberry
        >> isCool = "unsure"

```

A `*page` gives the ability to include multiple questions on a single page. But it's *not* possible to write questions like the one above in a `*page` because a person must click a "Next" button in order to leave

the page, and the above question includes code designed to run when a particular answer is clicked. To be successfully included in a `*page` with other content, the above question can be refactored using a pattern like this:

```
*page
    -- Pretend that there's other stuff on this page, please.
    -- Then:

    *question: What's your favorite kind of ice cream?
        *save: favoriteIceCream
        chocolate
        vanilla
        strawberry

*if: favoriteIceCream = "chocolate"
    >> isCool = "yes"

*if: favoriteIceCream = "vanilla"
    >> isCool = "no"

*if: favoriteIceCream = "strawber"
    >> isCool = "unsure"

```

---

*points: numberOfPoints 🔗

Gives or takes away points from users' scores. Optionally, a tag can be included as a way of grouping related points, like: `*points: numberOfPoints tagName`.

```
*question: What's 2 + 2?
    3
        *points: -1 mathPoints
    4
        *points: 1 mathPoints
    5
        *points: -1 mathPoints
    I don't know
        *points: 0 mathPoints

*points: 100 forNoReasonPoints
```

---

*program: programName

Seamlessly includes another GT program in the current program.

We tend to refer to these included programs as "subprograms" because they're included inside of an "outer" or "parent" or "main" program.

The `*program` keyword causes a specified subprogram to run immediately, resuming execution of the main program as soon as the subprogram finishes. Use this keyword to run code that you want to run temporarily (and that will necessarily complete) before you continue on, but *not* when you want to permanently switch to something else.

Note that `*program` is similar to `*switch`, but there's at least one significant difference between the two: `*switch` causes another specified subprogram to run immediately and *permanently pauses the current program* (unless another `*switch` is used in the future to resume it). Use `*switch` to switch to running some other portion of code that you may or may not ever come back from, or when you want to pause progress with one thing as you switch to do another thing and want to force the target subprogram to make an explicit choice about where to switch to next (instead of merely completing and returning automatically to a parent program).

```
1  -- https://www.guidedtrack.com/programs/11336
2  *program: 5 Factors of Happiness
3
```

---

*progress: percent

Displays a progress bar filled `percent`%.

```
*progress: 50%
*progress: {someVariable}%

```

---

*purchase

Allows you to process in-app purchases in your program.

Sub-keywords:

- `*status` = Used for checking if a user has a subscription and its status. This sub-keyword is required if `*frequency` and `*management` are *not* used.
- `*frequency` = Used to generate a subscription. This sub-keyword is required if `*status` and `*management` are *not* used.
- `*management` = Used to open a new window or tab where the user can manage their subscription. This sub-keyword is required if `*status` and `*frequency` are *not* used.
- `*success` = Defines code to run if the request succeeds. Any data returned from the request will exist in a variable called `it` and will only be available in this block. This sub-keyword is required if `*status` or `*frequency` are used.
- `*error` = Defines code to run if the request fails. Error information will exist in a variable called `it` (with properties "reason" and "message") and will only be available in this block. This sub-keyword is required if `*status` or `*frequency` are used.

See the In-app purchases section of the manual for instructions on how to configure your program to process subscriptions.

You must use exactly one of these sub-keywords: `*status`, `*frequency`, or `*management`. If the `*status` or `*frequency` keywords are used, then `*success` and `*error` keywords must also be used.

Sample code to check if a user has a subscription:

```
>> purchaseStatusError = 0

```

Sample code for subscribing a new user:

```
*purchase: trialPlan
    *frequency: recurring
    *success
        >> userSubscribed = 1
    *error
        >> userSubscribed = 0

```

Sample code to allow users to manage their subscriptions:

```
*purchase
    *management

```

This code will attempt to open a new tab or window for the user to manage their subscription. Some browsers may keep your program from automatically opening a new page. This happens because most modern browsers block events (like opening new tabs or auto-playing videos) that are not initiated by a user via a mouse or keyboard event. So, to help ensure that the browser allows a new window or tab to open, we recommend placing the above code in a clickable `*component`, like this:

```
*component
    Click here to manage your subscription.
    *click
        *purchase
            *management

```

`*question: questionText` 🔗

Asks a question.

Sub-keywords:

- `*type: typeOfQuestion` = Determines the type of question. Valid values are "calendar", "checkbox", "choice", "number", "paragraph", "ranking", "slider", and "text". This sub-keyword is optional.
- `*shuffle` = Randomizes the order of answer options in a multiple-choice or checkbox question. This sub-keyword is optional.
- `*save: variableName` = saves the response to a variable This sub-keyword is optional.
- `*tip: textToDisplay` = Inserts smaller text under the question. Usually, this is used for providing hints or extra help to users. This sub-keyword is optional.
- `*confirm` = Requires users to click a "Next" button after selecting their answers (as opposed to automatically progressing as soon as they select an option) in multiple-choice questions. This sub-keyword is optional.
- `*searchable` = Displays a text box and, when the user starts typing in an answer to the question, suggests possible matches from a predefined set of valid answers. This sub-keyword is optional.
- `*throwaway` = Prevents the question and responses from being stored in the CSV. This sub-keyword is optional.
- `*countdown: duration` = Requires that an answer be given in a certain amount of time (e.g., `*countdown: 1.minute + 30.seconds`). This sub-keyword is optional.
- `*tags: tagList` = Enables grouping related questions. Multiple tags can be separated by commas. All responses to questions with a given tag can be summarized with `*summary: tagName`. This sub-keyword is optional.
- `*answers: myAnswerList` = Injects multiple-choice answer options using a collection variable (e.g.,

`*answers: myAnswerOptions`). Note that this collection can be two-dimensional (i.e., a collection of collections) as a way of defining both a text to display and a value to be recorded (e.g., [['Agree', 1], ['Neither agree nor disagree', 0], ['Disagree', -1]]). This sub-keyword is optional.

- `*blank` = Allows users to continue without answering the question. This sub-keyword is optional.
- `*multiple` = Allows users to enter multiple answers to text questions. This sub-keyword is optional.
- `*default: myDefaultAnswers` = Pre-enters or pre-selects default answers. The variable `myDefaultAnswers` should match the type of question. For number questions, it should be a number; for text questions, it should be a text value; for multiple-choice or checkbox questions, it should be a collection of values; etc. This sub-keyword is optional.
- `*before: textToDisplay` = Puts text to the left of the answer box in text or number questions (e.g., `*before: $` puts a dollar sign to the left of the text box). This sub-keyword is optional.
- `*after: textToDisplay` = Puts text to the right of the answer box in text or number questions (e.g., `*after: donuts per month` puts "donuts per month" to the right of the text box). This sub-keyword is optional.
- `*min: minValue` = Sets a minimum value for a continuous range of values in slider questions. This sub-keyword is optional.
- `*max: maxValue` = Sets a maximum value for a continuous range of values in slider questions. This sub-keyword is optional.
- `*time: yesOrNo` = Determines whether or not users can specify a time in calendar questions. This sub-keyword is optional.
- `*date: yesOrNo` = Determines whether or not users can specify a date in calendar questions. This sub-keyword is optional.
- `*placeholder: placeholderText` = Inserts placeholder text in the field of a text or paragraph question. The text is not selectable and will disappear when the user starts typing in the field. This sub-keyword is optional.
- `*other` = Allows users to enter other answers to multiple-choice and checkbox questions. This sub-keyword is optional.

Normally, `*question` will create a single page, and the only thing on that page will be the question. However, it's possible to put multiple questions on a page by indenting them under a `*page` keyword.

In multiple-choice questions, note that answer options listed manually (i.e., *not* using `*answers`) may optionally include a Font Awesome `*icon: fontawesome-icon-class` or an `*image: imageUrl`.

If no sub-keywords are used at all, then the question will default to a text question.

---

*quit

Ends the entire program (including all subprograms) immediately.

```
*if: age < 18
    You're not eligible to take this survey!
    *quit

```

---

*randomize

Randomly selects blocks of code to run. A number of items to select and randomize can be specified with `*randomize: someNumber` or `*randomize: all`.

Sub-keywords:

- `*everytime` = Re-randomizes the selection every time a user passes the `*randomize`. By default, users will receive the exact same items each time they pass the same `*randomize` *unless* an `*everytime` keyword is indented beneath `*randomize`. This sub-keyword is optional.
- `*name: someName` = Defines a name for the randomized selection. This sub-keyword is optional.

By default, only a single block is randomly chosen from among the

blocks indented beneath `*randomize`. However, a number of items to be randomly selected can optionally be specified, like: `*randomize: 7`. Or, if all items should be selected (in a random order), use: `*randomize: all`.

Also note that while it's *likely* that blocks will be selected a roughly equal number of times, it's not *guaranteed*. If you need a guarantee that each block will be seen by an equal number of users, we recommend using `*experiment`.

```
*label: startLabel
*Here are random three idioms related to cats:*

*randomize: 3
    *name: catIdioms
    *everytime
    Cat got your tongue?
    Like a cat on a hot tin roof
    Curiosity killed the cat
    Like herding cats
    Look what the cat dragged in.
    It's raining cats and dogs.
    The cat's pajamas
    Let the cat out of the bag
    When the cat's away, the mice will play.
    Fat cat

*button: Get three more!
*goto: startLabel

```

In the above example, each "block" of code is actually just a single line of text. But sometimes it's useful to randomize entire blocks of code. In such cases, we recommend using `*group`, like this:

```
*randomize
    *group
        You're in the Pandas group!
        >> group = "pandas"

    *group
        You're in the Dolphins group!
        >> group = "dolphins"
```

```

```

`*repeat: numberOfTimes` 🔗

Repeats a block of code `numberOfTimes` times.

```
*repeat: 3
    For he's a jolly good fellow,

Which nobody can deny!

```

`*return` 🔗

Ends the current subprogram and returns to the main / parent / outer program.

```
*if: age < 18
    You're not eligible to take this part of the survey! However
    *return

```

`*service: serviceName` 🔗

Makes an HTTP request.

Sub-keywords:

- `*path: /some/path` = Defines the path to be appended to the URL defined in the service settings (e.g., if the service URL is https://example.com and the path is /hello/world, then the service request will be sent to https://example.com/hello/world). This sub-keyword is required.
- `*method: methodType` = Defines which HTTP method to use.

Valid options are GET, POST, PUT, and DELETE. This sub-keyword is required.

- `*send: associationToSend` = Sends an association along in the request. This sub-keyword is optional.
- `*success` = Defines code to run if the request succeeds. Any data returned from the request will exist in a variable called `it` and will only be available in this block. This sub-keyword is required.
- `*error` = Defines code to run if the request fails. Any error information will exist in a variable called `it` and will only be available in this block. This sub-keyword is required.

Once a service has been defined in the "Services" settings for a program, that service can be invoked by name via the `*service` keyword.

Unlike `*trigger`, `*service` calls are *synchronous*, meaning that the program will wait for the request's success or failure before continuing to subsequent parts of the program.

This example shows how to use the Free Dictionary API in a service call:

```
1  *label: startLabel
2
3  *question: Word:
4      *save: word
5      *type: text
6
7  >> response = "No results."
8  >> error = ""
9
10 *service: Free Dictionary API
11     *path: /{word}
12     *method: GET
13     *success
14         >> response = it[1]["meanings"][1]["definitions"][1]["c
15     *error
16         >> error = it
17
18 *if: error.size > 0
19     *ERROR* = {error}
20
```

*set: variableName

Sets a variable's value to `true`.

```
*question: Have you ever cheated on a school assignment?
    Yes
        *set: isACheater
    No

*if: isACheater
    Cheater!

```

*settings

Applies certain settings to the program.

Sub-keywords:

- `*back: yesOrNo` = Gives users the ability to navigate backward through a program by clicking a "back" arrow. This sub-keyword is optional.
- `*menu: yesOrNo` = Hides the run menu from the top left of the screen. This sub-keyword is optional.

```
*settings
    *back: yes
    *menu: no

```

*share

Inserts a button that enables users to share the program on Facebook.

```
*share

```

---

*summary

Summarizes user responses. Optionally, a tag can be specified with `*summary: tagName`.

```
-- Summarize all responses:
*summary

-- Summarize only responses to personality questions:
*summary: personality

```

---

*switch: programName

Switches to another program.

Sub-keywords:

- `*reset` = Indicates that the target program should be restarted from the beginning (instead of being resumed from its last state). This sub-keyword is optional.

See the `*program` section for more about the differences between `*switch` and `*program`.

```
*question: Which lesson would you like to view (from the beginr
    Lesson 1
        *switch: Lesson 1
            *reset
    Lesson 2
        *switch: Lesson 2
            *reset
    Lesson 3
        *switch: Lesson 3
```

`*trigger: eventName` 🔗

Triggers an event by name.

Sub-keywords:

- `*send: associationToSend` = Defines data to be sent to the event listener. This sub-keyword is optional.

Events can either be defined in `*events` at the beginning of the program or in JavaScript code in embedded GT programs. Triggered events run asynchronously, meaning that calling `*trigger` will initiate the relevant event, but the program *will not* wait for the event to finish its execution; instead, the program will continue right on to the next line of code after triggering the event. It's important to keep this in mind if subsequent parts of your program rely on the result of an event's execution.

Here's an example of triggering an event that has been defined within the program itself (under `*events`):

```
1  *events
2      incrementMyNumber
3          >> myNumber = myNumber + 1
4          *goto: startLabel
5              *reset
6
7  >> myNumber = 0
8
9  *label: startLabel
10 Currently, your number is: {myNumber}.
11 *button: Increment it, please!
12 *trigger: incrementMyNumber
13 *wait
14
```

(That final `*wait` is included because otherwise the program would

end! Since triggered events run asynchronously, the program would keep going after `*trigger: incrementMyNumber`...but since there's nothing after it, the program would end.)

And here's an example of triggering an event in an embedded program where the relevant event is defined in the embedded page itself.

```
*trigger: showAnAlert
    *send: {"message" -> "Danger, Will Robinson!"}

```

The "showAnAlert" event would be defined somewhere in the JavaScript code in the embedding page. Read more about defining JS events here.

---

*video: youtubeUrl

Embeds a YouTube video into the page.

```
Here's a video of Vi Hart talking about the dragon curve fractal
*video: https://www.youtube.com/watch?v=EdyociU35u8

```

---

*wait

Causes the program to wait forever (`*wait`), for a certain amount of time (`*wait: duration`), or until data has been sent to the GT server(s) (`*wait: data`).

```
Take 30 seconds now to ponder the meaning of life.

*wait: 30.seconds

*question: Would you like to know what /we/ think is the meanir
    Yes
        *wait: data
        *goto: https://en.wikipedia.org/wiki/42_(number)#The_Hi
    No

```

`*while: condition`

Runs a block of code as long as `condition` is a true statement.

Be aware that failing to make the condition eventually become false can cause the browser to freeze up and/or crash! For example, the condition in the following `*while` loop will always be true, so the loop will never exit, so the browser will freeze up and/or crash.

```
*while: 0 < 1
    All work and no play makes Jack a dull boy.

```

The `guidedtrack` library within custom services

The `guidedtrack` library contains useful functions which you can use within custom service routes. It should be automatically imported into every route you create with this line at the top:

```
import guidedtrack from "guidedtrack-db";

```

In case it isn't, add the line above near the top to get used to the functionalities documented below.

The `table` namespace (`Table` object)

The `guidedtrack.table` namespace allows you to work with a specific table in your custom database.

To open a connection to a table, call `guidedtrack.table(tableName)`, where `tableName` is a string representing the name of the table you'd like to work with:

```
const table = guidedtrack.table("participants");

```

The above will return a `Table` object with methods you can use to do useful things:

`Table.insert(data)`

Inserts the data provided into the table. If the data includes an `_id` column/field, the operation will try to use that ID to insert. If not, it will randomly generate an ID for the new record, which will appear in the `_id` column after the insert has been executed and will be a part of the result returned.

Arguments

- `data`: the data to insert as a JSON object. The object keys are the columns to write to, and the values of the object should contain the data to write into the relevant column.

Return value

An `Operation` instance.

Example

```
1  import guidedtrack from "guidedtrack-db";
2
3  export const handler = async (event) => {
4      var data_sent = JSON.parse(event.body);
5
6      return await guidedtrack.table("participants").insert(data_s
7  };
8
```

`Table.find(id)`

Loads the record with the given ID from the table.

Arguments

- `id`: the ID of the record in the table; this is usually randomly

generated when the record is inserted into the table.

Return value

An `Operation` instance.

Example

```
import guidedtrack from "guidedtrack-db";

export const handler = async (event) => {
    var id = event.queryStringParameters.id;

    return await guidedtrack.table("participants").find(id).resp
```

`Table.search(selector)`

Loads all records in the table that match the given selector.

Arguments

- `selector`: a JSON object describing the criteria documents need to meet; must be a valid Mango query selector.

Return value

An `Operation` instance.

Example

```
import guidedtrack from "guidedtrack-db";

export const handler = async (event) => {
    var query = JSON.parse(event.body);

    return await guidedtrack.table("participants").search(query)
};
```

The `Operation` object

Represents a pending database operation, which can be executed and converted into an appropriate response via the `.response()`, `.values()`, and `.value()` methods.

`Operation` instances also have other functions, which can be used to configure the operation while it's still pending and before it is executed.

Functions to execute the operation

`Operation.response`

Returns a JSON object representing the result of the database operation wrapped in a valid AWS Lambda response object:

```
{
  statusCode: 200,
  body: JSON.stringify(OPERATION_RESULT)
}

```

Above, `OPERATION_RESULT` is the result returned by the database after the specified operation is executed.

`Operation.values`

Returns an array of documents returned by the database in response to the operation.

This is typically the method you want to call to get at the result of a `Table.search` operation.

`Operation.value`

Returns the document returned by the database in response to the operation. In case the database responds with multiple documents, `value()` will just be the first document in the result.

This is typically the method you want to call to get the result of a `Table.find` operation.

Functions to configure a pending operation

Currently, these only apply to search operations. Other table operations are simpler and therefore cannot be configured.

`Operation.columns(...columns)`

This only applies to search operations.

Limits the columns the result will contain to the ones in the argument list:

For example, the code below will get just the last name and age of all Johns in the "participants" table:

```
const result = guidedtrack.table("participants")
  .search({"first_name": "John"})
  .columns("last_name", "age")
  .values();

```

`Operation.skip(n)`

This only applies to search operations.

Skips the first `n` records that match the criteria and returns the rest.

This might be useful in situations where you want to load or show records in batches, such as when you want to paginate data. Suppose you're showing all entries in the "participants" table split up into pages of 100 participants each. To get the participants that would be on the second page, you would need to avoid showing records from the first page like so:

```
const participantsExcludingFirstPage = guidedtrack.table("partic
  .search()
  .skip(100)
  .values()

```

`Operation.limit(n)`

This only applies to search operations.

Instruct the search operation not to return more than `n` records. To continue the example above, we could combine `.skip` with `limit` to get the records that we'd need to display on the second page:

```
const participantsExcludingFirstPage = guidedtrack.table("partic
  .search()
  .skip(100)
  .limit(100)
  .values();

```

The above will load 100 records, starting at record #101, which is precisely the data for the second page.

`Operation.sort(...fields)` 🔗

This only applies to search operations.

Specifies the order in which we'd like the search operation to return the results. For example, if we want to get all Johns in the "participants" table and order them by age (oldest first), we'd do this:

```
const result = guidedtrack.table("participants")
  .search({"first_name": "John")
  .sort({"age": "desc"})
  .values();

```

Each `field` argument in `fields` must be a JSON object with the name of the field to sort by as a key and the sort direction as the value, and you can sort by multiple fields at the same time. Here's how we'd get all Johns in the "participants" table and order them by age (oldest first) and last name:

```
const result = guidedtrack.table("participants")
  .search({"first_name": "John")
  .sort({"age": "desc"}, {"last_name": "asc"})
  .values();

```