An Introduction To GuidedTrack v0.29

# **TABLE OF CONTENTS**

[TABLE OF CONTENTS](#h.v8dk4jeylk5x)

[HELPFUL RESOURCES](#h.2t69cyp51orq)

[Sample GuidedTrack Programs](#h.8bq7m0bumapo)

[GuidedTrack Toolbar](#h.dardcsgayz53)

[GuidedTrack Blog](#h.2y6pjqq9nna8)

[INTRODUCTION](#h.4g15lpsldhkq)

[What is GuidedTrack?](#h.vk4618aa9f10)

[Why use GuidedTrack?](#h.r61j75wsc9h4)

[What Does a GuidedTrack Program File Look Like?](#h.owvvrrqgyaat)

[How are GuidedTrack Programs Deployed?](#h.giy2ev2rizxa)

[USING THE GUIDEDTRACK WEBSITE](#h.d5310r7761rs)

[Creating an Account](#h.cf7vmt7kw5j7)

[Creating a Program](#h.hrbip5hwyck6)

[Writing Your Program in a Separate Text Editor](#h.datxzimgsm2i)

[Writing or Adding Your Program to GuidedTrack](#h.so1b292v6nqd)

[Previewing Versus Running a Program](#h.xpn8xq5z20p7)

[Sharing Your Program’s “Run” Link With Others](#h.jvnlgpg9jluk)

[Sharing the “Preview” Version of Your Program](#h.umv6y0k6aa7c)

[Adding Collaborators and Restricting Access](#h.2fzea2w7rorr)

[Looking at the Data](#h.gy6ztt8dmohg)

[What the Data Page Shows](#h.j2ibpyoqardj)

[Understanding and Looking at the Questions Tab](#h.p7y3f23iinra)

[Generating and Understanding a CSV of Your Data](#h.hn1l60bibnst)

[BASIC ELEMENTS OF A GUIDEDTRACK PROGRAM](#h.iavbjwaq33pt)

[Text Lines](#h.a0v1mid0wc6y)

[Bold, Italic and Underlined Text](#h.3sw5xby2irmm)

[Keyword Lines](#h.wmx986x02q56)

[Keyword Configurations](#h.2bvw8v3iy4qc)

[Comment Lines](#h.eoep4fxojzc7)

[Indentation](#h.mxcyfoj2cped)

[Adding Blank Lines](#h.ca60h21nnpxe)

[ADVANCED OPTIONS](#h.3kqxg77p8cu4)

[Customizing the Look of Your Program (for CSS Smarties)](#h.xdjyf948df6r)

[Sharing a “Run” Link with Custom-Loaded Variables](#h.59exr989reqe)

[Adding a single text variable](#h.btq3tpwfk8r)

[Adding a single numerical variable](#h.hjgd61v9pbd4)

[Adding a single collection](#h.klifzdi52oma)

[Adding multiple variables](#h.er796ecq6loy)

[ASKING QUESTIONS](#h.2w217a80et01)

[Standard Multiple Choice Questions](#h.uo18l1mh0wmx)

[Have Users \*confirm Their Selection](#h.p8hr9u6w7j1c)

[Standard Text Box Questions](#h.45durwkejc8v)

[Paragraph Text Box](#h.wnwstu1qf21)

[Number Only Responses](#h.x7jjgdt57hi4)

[Allowing Blank Responses](#h.sacl9ucxxs93)

[Adding Text \*before and \*after](#h.s0o4mdmf2479)

[Allowing Users to List \*multiple Answers to a Single Question](#h.xttok7owraje)

[Standard Checkbox Questions](#h.25jmb8iz3van)

[Standard Slider Questions](#h.l0fg2iufa5ls)

[Multiple Choice Where What Happens Depends on the Answer](#h.kelicmd5pfhy)

[Text Boxes Where What Happens Depends on the Answer](#h.lf0r1d595kqp)

[Question Add-Ons](#h.tz69diiikuem)

[Adding an \*icon](#h.rbn29jup8a2e)

[Displaying answer options from a collection of answers](#h.x7jh8eab34xs)

[Giving a Tip](#h.ib4u9vl4htfz)

[Providing Default Answers to Questions](#h.ktkg3t5dhrqf)

[Save a User’s Answer](#h.2r0dzihnrymi)

[Shuffle Answer Options](#h.9tcq8ymt6zgq)

[Summarizing Entered Data](#h.ft5cpuwz6kg8)

[ADDITIONAL KEYWORDS](#h.5t6yovik0229)

[Charts](#h.qsyt2y3733c2)

[Basic Bar Chart and Charting 101](#h.8n6syuppvfsc)

[Basic Bar Chart](#h.omfiz12i3p2w)

[Using Simple Variables with \*data](#h.fnzaycj2gbvm)

[Using a Variable of a Collection with \*data](#h.tvthuhrl66jf)

[Using a Variable of a Collection of Collections with \*data](#h.2b7k762gu1v8)

[Setting the \*min and \*max Values of Your \*xaxis and \*yaxis](#h.qm31bv9gnz68)

[Adding a Pretty Chart \*color](#h.c5lahantnykl)

[Multiple colors and multiple \*data keywords](#h.leyqb2ptbf5e)

[Add \*ticks - I.e., Customize the Look and Position of the Intervals on the Axes](#h.bsiz7mlbuqno)

[Multiple Chart Types on One Chart - and Multiple Axes on One Chart](#h.87kkp46d6h5c)

[Scatter Plots](#h.h12ul1rpu9fc)

[Using the \*rollover Keyword with Scatter Charts](#h.xpz7za7q8nu)

[Adding a \*trendline to Scatter Charts](#h.t081e66kuuze)

[Line Graphs](#h.493vi273iqe1)

[Clearing the Screen](#h.ixiznz3z534e)

[Emailing Users](#h.1mw275vkdtql)

[Experiments](#h.5kw3jlihw299)

[Going to Different Parts of the Program or to Other Websites](#h.6fjpk0um0568)

[Going to Different Parts of the Program](#h.axq80nlapjkd)

[Going Immediately to Other Websites](#h.h0tpsn9kjy3g)

[Headers](#h.liwlzvb3zox0)

[If This… Then That](#h.4ieiokufb9r9)

[Links](#h.9s0t6qmjn3yg)

[Logging In](#h.8xvm8ca5k2pr)

[Adjust the settings to make the login screen appear in the beginning](#h.7u2m0pg4bazr)

[Add a login screen in the middle or end of the program](#h.axk3bs156zq2)

[Lists](#h.l5m6iv9fhc0f)

[Bullets](#h.ypth1eiy5n4o)

[Expandable](#h.m4n4zixini54)

[Numbered Lists](#h.ortb815ppid)

[Maintaining Text on the Screen](#h.jua2wrmq7oqc)

[Media](#h.nkjqs5vuiejt)

[Audio](#h.p06avlbgcb81)

[Images](#h.apieg2qehxne)

[Videos](#h.b7go696a3ir3)

[Page Breaks](#h.f2ctivxug144)

[Questions](#h.z7ap9ulc52ee)

[Buttons](#h.aqu3jxtqzmxf)

[The Clear Keyword](#h.nbma0651oo6m)

[Page Navigation](#h.nzxb4qoikbi)

[Points](#h.ksh6vnvmus8d)

[Progress Bars](#h.4t8twnjzrqwa)

[Quitting a User’s Run](#h.wgj3oeitvzvq)

[Randomizing What is Displayed](#h.hy1j1pd6pyqp)

[Randomize All](#h.tdi20v4fq8jf)

[Randomizing Every Time](#h.hd9abyu9nakk)

[Embedded Expressions in \*randomize](#h.rcts3p6x2f4k)

[Randomizing a Group of Commands](#h.av3xilwimqe1)

[Make Viewing Data Easier by Naming Randomizations](#h.6njp7tbwphsk)

[Repeating](#h.mleik2rfn7y2)

[Repeating While a Given Condition is Being Met](#h.lzyn9nnurcvk)

[Share Buttons](#h.24dc5cauru4q)

[Using Collections](#h.6evhh8hfcft)

[Getting and using a specific item in a collection](#h.a9fervx00f1)

[Get the size of a collection](#h.wzwtsaey94q7)

[Get the mean value of a collection of numbers](#h.tybxq3fr93jr)

[Add something to the end of a collection](#h.h04p8i46lore)

[Insert something in a specific point in the collection](#h.16rm4ormz3yq)

[Sort a collection](#h.tgfgonpjtq27)

[Randomize a collection](#h.3v9xzc9kvfl4)

[Combine two or more collections](#h.d793mf79d6so)

[Using Multiple GuidedTrack Programs](#h.oqt8u92epsxw)

[Using Variables](#h.zcqh86rktvec)

[Waiting a Specified Amount of Time](#h.48j24ledv7h4)

[Special Settings](#h.ji9syz4s0i3j)

[Back Buttons](#h.tupsdph8mhot)

# **HELPFUL RESOURCES**

## Sample GuidedTrack Programs

We’ve created some sample GuidedTrack programs to help you learn common keywords for common GuidedTrack uses. Click any of the programs below to be taken to a separate document featuring the GuidedTrack language of the program. Each contains sample keywords, detailed explanations of how to use each, and links to the live version of the program.

[Sample GuidedTrack Survey](https://docs.google.com/document/d/1l4-j-KEIJw6Twan7sE2IKY8A59oc6XKR2A9P19HhqK4/pub)

[Sample GuidedTrack Quiz](https://docs.google.com/document/d/1smgGrSqnmMB61zZ-ynZIxFR1jzvPbXdGjKbTZ53KSRc/pub)

[Sample GuidedTrack Training](https://docs.google.com/document/d/1l-KwfTp0BnyE6gSwLgl2sKjRTi21GuqMvhg95fUl62U/pub)

[Sample GuidedTrack Experiment](https://docs.google.com/document/d/1nz6XDCsg8td7FbT-qstTd3fWQUxUTSw8alEBGYwt0JI/pub)

[Sample GuidedTrack Tool](https://docs.google.com/document/d/1yUJ2zjIvkcP8tuMs8PfMQwOlWeMrGcvnsMP7Jt9CWTM/pub)

## GuidedTrack Toolbar

Would you prefer to read a shorter description of GuidedTrack keywords? Our GuidedTrack Toolbar demonstrates how to use each keyword. It uses short, quick examples and contains very little explanatory text. It’s a great tool for those who understand coding basics and want a quick explanation of the GuidedTrack language. To access it, simply navigate to the editing screen of the program you’re working on. You’ll see the toolbar to the left of the editing window.

## GuidedTrack Blog

Stay up to date on brand new features by subscribing to our GuidedTrack newsletter and blog. [Click here for more info](http://blog.guidedtrack.com).

# **INTRODUCTION**

## What is GuidedTrack?

GuidedTrack (abbreviated “GT”) is a programming language for writing dynamic, interactive applications that can be run on the web (and eventually also on Mac, PC, iPhones, and Android phones, with or without an internet connection.) GT allows you to write apps that consist of text, multimedia (such as images, video and audio) and questions presented to the user. GT also allows you to vary what happens in the app based on the user’s response to questions. GT makes it easy for both seasoned programmers and complete beginners to create:

* interactive training and educational materials
* polls and surveys
* quizzes
* choose**-**your**-**own**-**adventure stories
* self-help and mood monitoring applications

There are many types of programs that *cannot* be built using GT, such as**:**

* address books
* word processors
* image editors
* web browsers
* chess programs

What’s the difference between these two lists of applications? The first list can usually be created with just a series of text, images, video, audio and questions, with the answers to questions determining what happens next. The second list of applications doesn’t fit this paradigm. For instance, in an image editor program**,** you need to be able to click to choose the paintbrush you want to use, and then click and drag to indicate where to put digital paint. Merely asking the user questions (as GT programs do) is not enough to determine what the user wants (e.g. where exactly they wish to put paint on an image).

## Why use GuidedTrack?

If the type of application you are building can be written in the GT language then you’re in luck! That’s because it is *much* easier to make GT programs than to build programs from scratch using programming languages like C++, Java, Python or Javascript. Learning GT is quick and easy, even for people who have never programmed before. Total novices can make simple programs with GT in a few minutes, and can learn to make more complex programs in a few hours. Therefore, GT allows for very rapid development of dynamic**,** interactive programs. Furthermore, GT programs just need to be written once, and then will (in time) be deployable on many platforms independently of one another (web, phones, Mac and PC). Very few programming languages have this ability. To summarize, GT allows non-programmers to quickly create dynamic applications and to easily deploy them on a wide variety of platforms. And making GT apps is fun!

## What Does a GuidedTrack Program File Look Like?

GT programs are text files. You can edit these files in any standard text editor (like TextEdit for Mac, or Notepad for Windows). However, some text editors are nicer for writing GT programs than others. We recommend using [Sublime Text](http://www.sublimetext.com/) which can be downloaded for free, and is available for Mac, Windows and Linux. You can also create, edit, and save your GT programs on the GuidedTrack.com website.

A GT program (which as we’ve said, is just a text file) contains text and commands that represent the interactive experience the user will have when running the program. When a GT program is run, the first line of the file is run first, followed by the second line, and so on (though there are some commands that can cause the lines to be run out of order). Sometimes GT programs consist of multiple text files rather than just one; however one file will still be the “main” file and will be run first when a user goes to that app. See [Using Multiple GuidedTrack Programs](#h.oqt8u92epsxw) for more on this.

## How are GuidedTrack Programs Deployed?

GuidedTrack programs are either created in a separate text editor and added to the GuidedTrack website, or they are created directly in GuidedTrack.

Once your program is on the GuidedTrack website, you’ll have access to a unique link that you can give to other people. This link will allow others to run your program from the GuidedTrack website.

You can also give out a special link to those you’d like to preview your program, and you can give special access to other people with a GuidedTrack account who can edit your program or view your program (including the GuidedTrack language).

In the future, you’ll also be given the option to automatically convert your program into iPhone and Android apps.

# **USING THE GUIDEDTRACK WEBSITE**

## Creating an Account

Simply go to <http://www.guidedtrack.com/users/sign_up> to sign up for an account. A valid email address is required.

## Creating a Program

You can create a separate text file that contains your GuidedTrack language and upload it into GuidedTrack, or write a program directly in the GuidedTrack website.

### *Writing Your Program in a Separate Text Editor*

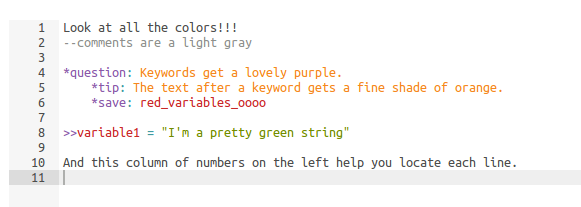
If you choose to write your program in a separate text editor first, we recommend one like [Sublime Text](http://www.sublimetext.com/). Sublime Text gives you a lot of options when writing your program, and it is easy to copy and paste your Sublime Text document directly into a GuidedTrack program.

Copying and pasting from other programs, such as Microsoft Word, sometimes cause formatting errors and you may find that your indentations have been altered. If you do wish to use Microsoft Word for writing your programs we’ve created some [recommendations](https://docs.google.com/document/d/1MDAQdv3J3Ydl4vP8tXbX3QozcHlYWxtBwuRgoBUwRU4/pub).

At this point, copying and pasting your program is the only option for adding it to your programs list. However, in the future we will offer upload options.

### *Writing or Adding Your Program to GuidedTrack*

From your programs page, click on the tab in the blue navigation bar called “New program.” Enter a name for your program, as well as a short description, if you so desire. In the large box, copy and paste your program from a separate text editor or begin to type it out if you are writing it in the GuidedTrack website. You’ll notice that as you edit your program, certain keywords will begin to change color.



The GuidedTrack text editor is actually very powerful stuff. There are a number of special shortcuts you can use to become an editing ninja. For example, you can use Ctrl+f (Command+f on a Mac) to bring up a box that will help you find text and Ctrl+h (Command+Option+f on a Mac) to replace text. [Click here](https://github.com/ajaxorg/ace/wiki/Default-Keyboard-Shortcuts) to learn more ways to be an editing whiz.

Once you are finished, click either “Save” or “Preview” to save your program and see how it looks live.

**Previewing Your Program as You are Editing It**

As you’re making changes to your program, you might want to see how those changes look on the screen. To do that, locate and click the “Preview” button at the top of your edit screen. You’ll then see the program as an actual user would see it.

At the top right corner of the preview screen you’ll also notice a little icon. Clicking this icon reveals several additional options you can click:

* Preview again
* Run
* Edit
* Share
* Data

To return back to editing, simply click “Edit.”

All these preview runs will show up on your data page as previews. They will be in a lighter font color and will have a small icon on the far right that says “preview” instead of “data.” When you download a CSV of your data or look at the answers users gave to your questions, you’ll ONLY see the real data, not data from your preview runs. If you’ve made a mistake, you can easily toggle any run from “preview” to “data” and vice versa. This helps cleanly separate data generated during previewing from real data produced by people using your program.

### *Previewing Versus Running a Program*

As an editor, you can “preview” your program as we described above, or you can “run” your program. Both options will allow you to see your program in its live form, as an actual user would see it. The only difference is how your answers to questions in the program and path through the program is recorded in the data. In a preview run, your data will be stored on the data page as a preview, and will not appear in downloaded copies of your data. In a regular run, your data will be stored on the data page as data, and *will* appear in downloaded copies of your data.

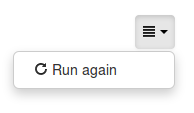
## Sharing Your Program’s “Run” Link With Others

When your program is ready for others to use it, click “Save” from the edit page, then visit “Share” from the blue navigation bar.

You’ll see a page with different ways to share your program. You can email or send the program link directly, place the link on your website, or embed your program directly into a website.

When users take your program through any of the ways to share it, their responses and path through the program will be recorded as data in the data page.

Users who run your program will also have a small icon on the top right corner of their screen. Clicking on this icon will show them the option to “Run again.” If they click this button they can start your program again from the beginning.



As an editor running your own program, you will see this icon as well, plus additional options. You will also see the following:

* Run again
* Preview
* Edit
* Share
* Data

**Advanced Topic: Custom set certain variables from the share link with URL parameters.**

You can share a link that has custom-loaded variables, giving you the flexibility to give out slightly different versions of your program to different groups of people (e.g. if you have a quiz and want some people to see their quiz results right away, but want to hide the results for other, specific people). To learn more about this advanced topic, [click here](#h.59exr989reqe).

### *Sharing the “Preview” Version of Your Program*

If you’d like to share your program with others, but would like their data saved as “preview” data, not run data, then you have three options:

1. You can give them access to your program by adding them as a collaborator. They will then be able to view information about the program and can edit the program with your permission. They will also have an option to “preview” the program, which will allow them to try the program without their data being recorded. For more information about giving access, see the section on [adding collaborators](#h.2fzea2w7rorr).
2. You can give users the “run” link, then change their data on the data page to reflect a “preview,” not actual data. For more information on this, see the section about the [data page](#h.gy6ztt8dmohg).
3. You can give them the “run” link, after first changing the word “run” to “preview” at the end of the link.

## *Adding Collaborators and Restricting Access*

Adding collaborators to view or edit your program is easy.

1. From the main programs menu, find the program you would like to add collaborators to.
2. Click the program to be taken to be taken to the editing page.
3. On the top right-hand side of the new navigation menu, click the “Settings” option.
4. Scroll to the bottom and click “Add collaborators.”
5. Enter the email addresses of the collaborators you wish to add, separated by an enter key or a comma.
6. Select whether you want the collaborators to have a status of “reader” or “editor.”
7. Click “Add.”

Now you will see these collaborators on your Settings page. You can edit their status at any time, or you can delete them by selecting their name and clicking “Delete selected.”

If you’d like your program to be private and only shared with a select group of people, be sure to check the box “Access restricted,” which is on the same page beneath “Access to program.” Then, only those people whose email addresses you have entered as collaborators will be able to view your program.

**Deleting a Program**

From the programs page, you can delete a program by clicking the “Delete” button, or by clicking on any program, clicking “More” and then clicking “Delete.” Be sure you actually do want to delete your program, as you cannot restore your program once it is deleted.

## Looking at the Data

The data page can give you a lot of information about each run of your program. To access this page, simply click “Data” from the homepage, or click “Data” in similar option menus on any of the other pages.

### *What the Data Page Shows*

The data page shows each and every run or preview that has ever been done on your program, whether the preview or run was completed fully or only partially completed. This page has several different columns, which we’ll go over one by one.

**User**

In this column, you’ll see the email address of the user who completed that preview or run, provided they have an account with GuidedTrack. If they do not have an account with GuidedTrack, or were logged out while completing their preview or run, then this field will be blank.

**Time Started**

This is the time in which a user began a preview or run.

**Total Time Spent**

This column lists, in minutes, the amount of time a user spent completing your run. If the user completed your run in multiple sittings, or if they left the program open for an extended period of time (perhaps while doing other things), then this time may look very large.

**Finished**

There will be a small checkmark here if the user finished your program, that is, if they got to a screen where they could not possibly go forward.

**Points**

If your program awards generic points (the type that users can see in the upper corner of their screen, i.e. points without a codeword), then the total number of points they got will be displayed here. Next to their total points, and in parentheses, will be the total percentage of points the user received out of the maximum possible number of points. So, if your program awarded up to 10 points and one user got 8 points, you would see “8 (80%)” for that user’s run.

If your program awards special points (the type that users **cannot** see in the upper corner of their screen, i.e. points with a codeword), then you will also see the total number of points the user got for that point type and the percentage of that point type’s total. So, if it were possible to get up to 40 “gnarly” points in your program and a user got 20 gnarly points, you would see “20 (50%)” under a column called Points (gnarly). If you’re not sure what we mean by special points, review this section on [Points](#h.ksh6vnvmus8d).

**Answers**

In the column that says the word “Answers” after each run or preview, you can click on any of the “Answers” to see the particular questions asked of the user during their preview or run and the exact answer that they gave.

By clicking on any of the questions on an “Answers” page, you can also see other answers that all other users gave for that question. You can also see all the answers given for a particular question by going to the “Questions” tab, which we’ll describe in a bit.

**Details**

This is the technical stuff, and is **not particularly useful for the majority of people; feel free to skip over this whole section**! Clicking “Details” for any user’s run or preview will show you a detailed play-by-play of exactly what that user clicked on, saw, and typed, all the keywords they came in contact with, as well as the exact time in which these things occurred and in what sequence. The details page is not for the faint of heart! Here’s an overview of what the details page contains:

*Time:* The time that the event occurred at.

*Category:*  There are a few different categories that could be here. Basically, a category will tell you the category of the event that occurred (e.g. “answer” if the answer just gave an answer; “start\_run” if it’s the beginning of the run).

*Node ID:* Nodes are unique identifiers, made up of a 7-character string of numbers and letters. Each line of text, keyword, question and answer pair, etc. will have a unique identifier. You don’t really need to know what it is, but it occasionally comes in handy.

*Type:* Type is similar to “Category,” but it’s more specific. The type will tell you the type of event that occurred (e.g. “text” for lines of text, “button” if there was a button, “question\_multiple\_choice” if there was a multiple choice question).

*Text:* Text will show you any text that was written at that point in the program: on a button, in a question, or in other areas where text is shown.

*Value:* This column is most useful for seeing the answers that a user gave if they answered a question at that particular point. This column may also include nodes, which can be safely ignored. Answers can also be obtained through much easier channels, as described earlier.

*Sequence:* If you used any \*randomize keywords in your program then the sequence of what the user saw will be shown here. The sequence will be provided in nodes, unless you gave each possible randomization a unique \*name, in which case you will see the codenames in the sequence in which they occurred. See the section on [naming randomizations](#h.6njp7tbwphsk) for more information.

*URL:* If you provided a URL to the user at this point in the program, then you will see the link to that URL here.

*Sequence #:* This column will be in numerical order. It tells you when this event occurred in the sequence of all events.

**Preview or Data**

This column will show you whether this row represents a preview or an actual run. If the icon here says “preview”, then the information will be shown in a slightly dimmed font and the data provided here will not be included in a downloaded CSV of the data.

If the icon here says “data”, then the information is in normal font color and the data provided will be included in a downloaded CSV of the data.

You can toggle these icons back and forth between “preview” and “data.” Allow the icon to say “data” if the row represents a legitimate run of your program and/or you are interested in analyzing the data of that run later on. Make the icon say “preview” if the row represents a preview run of your program and/or you do not want that data to be a part of your CSV.

### *Understanding and Looking at the Questions Tab*

The “Questions” tab will show you all of the questions asked in your program. In a small blue box, you can also see how many times that question was answered during runs of your program.

Your program may also have a column that says “Versions.” The most recent version of your program will be listed first. If you deleted any questions or significantly modified any questions, prior versions of questions will be listed as well, beneath earlier version numbers.

You can click on any of the questions to see the actual answers that users gave. You’ll come to a page called “Listing answer,” which will also give the following:

**Question**

The question that was asked

**A Chart**

The chart will show you a bar chart of the number of times an answer was given and the percentage that each answer was given. This chart will show for all types of questions, including multiple choice, text, and paragraph. However, if there are more than 15 unique answers (which may be especially likely in a text or paragraph question) then the chart will not appear.

**Answers**

The total number of times this question was answered

**Numerical**

The total number of times this question was answered numerically (i.e. someone wrote 7, 3.2, or another number as their answer).

**Average**

The average of all numerical responses

**Time**

The time at which the answer was given

**Value**

The actual answer that was given

### *Generating and Understanding a CSV of Your Data*

If you’d like to generate a CSV of your data that you can download onto your computer, simply click the large blue button that says “Generate CSV.” You will have to wait a few moments while the program crunches your data. Once it’s finished, a green “Download CSV” will be displayed and you can click on it to download or open your CSV. Remember, this CSV will only show you data that was generated from actual runs of your program, not quick [previews](#h.xpn8xq5z20p7) of your program.

Your CSV will contain a lot of information and may at first look a bit overwhelming. We’ll explain each column in detail.

**Run**

This is a unique number provided to that individual run, distinguishing it from all other runs in all other GuidedTrack programs.

**Program Version**

This is the version of your program that run occurred on.

**User**

In this column, you’ll see the email address of the user who completed that run, provided they have an account with GuidedTrack. If they do not have an account with GuidedTrack, or were logged out while completing their run, then this field will be blank.

**Time Started**

This is the time in which a user began a run.

**Time Finished**

This is the time in which a user completed a run.

**Time Spent**

This column lists, in minutes, the amount of time a user spent completing your run. If the user completed your run in multiple sittings, or if they left the program open for an extended period of time (perhaps while doing other things), then this time may look very large.

**Position**

This column will show a unique code (aka "node") that can tell you how far along a user has gotten in their run, before quitting, voluntarily leaving, or completing the run. The code represents the text or keyword that was at the top of the last visited screen.

**Points**

If your program awarded generic points (the type that users can see in the upper corner of their screen, i.e. points without a codeword), then the total number of points they got will be displayed here.

**Points %**

If your program awarded generic points, this column will have the total percentage of points the user received out of the maximum possible number of points. So, if your program awarded up to 10 points and one user got 8 points, you would see 80% for that user’s run.

**Additional Columns**

You will also see additional columns for these types of things (if you’ve added them to your program).

*Special points:* If your program awards special [points](#h.ksh6vnvmus8d) (the type that users **cannot** see in the upper corner of their screen, i.e. points with a codeword), then you will also see the total number of points the user got for that point type and the percentage of that point type’s total. So, if it were possible to get up to 40 “gnarly” points in your program and a user got 20 gnarly points, you would see “20” under a column called “Points (gnarly)” and “50%” under a column called “Points % (gnarly)”.

*Randomize*: If you used this keyword, you’ll see the unique identifier of that randomization, with the single or sequence of randomizations that occurred for that user. Be sure to visit the [naming randomizations section](#h.6njp7tbwphsk) to make sure the data here is easily readable.

*Variables:* If you’ve added any [variables](#h.zcqh86rktvec), you’ll see the codename of the variable in the column heading and their final ending values below. For \*set variables, these will appear as “TRUE” if the user tripped the \*set keyword or the cells will be blank if the user did not trip the \*set variable.

*Questions:* If your data includes [questions](#h.2w217a80et01), you’ll see each question in the column, with the answers beneath it. If the user answered the same question multiple times (either by pressing a back button or via a \*goto keyword), then each of their answers will be separated by a pipe: |

# **BASIC ELEMENTS OF A GUIDEDTRACK PROGRAM**

GT program files are processed line by line. In most text editors, a line ends when you type “enter” or “return” on your keyboard. Some text editors will automatically wrap lines that are too long, but this is not the same as a line actually ending (the text editor merely makes it look as though the line ended, for visual convenience).

There are three basic types of lines in GT, each of which has its own special purpose:

**1. Text**

Text lines display text to the user.

**2. Keywords**

Keyword lines cause something to happen (e.g. a question to be asked to the user).

**3. Comments**

Comment lines are just for recording notes to yourself or other people who edit the file, and have no effect on the user experience.

## Text Lines

Text lines consist of exactly what they sound like: they are lines of the file that contain ordinary text. For example, consider:

The red, evil fox swam with the slightly less evil, silver dog.

[▶Run](http://www.guidedtrack.com/programs/sn5jrqx/run)

This is a valid GT program that consists of only one line of text. When this program is run, it just displays the text to the user. That’s it! Multiple lines of text are also allowed, such as:

This is the first line.

And here’s the second.

The third line! This line’s the best.

[▶Run](http://www.guidedtrack.com/programs/sizih4k/run)

When this program is run, it too would simply display all this text to the user. The user would see a blank line between the second and third line, just as it appears above.

### *Bold, Italic and Underlined Text*

The way the text looks can be modified. You can make the text bolded, underlined, or italicized by adding special symbols before and after the text you want to change.

Place two asterisks around text you’d like to \***bold**\*. Place two backslashes around text you’d like to /*italicize*/. And place two underscores around text you’d like to \_underline\_.

You can place these symbols around whole phrases. Here are some examples:

This will let you \*show some text\* in bold.

This will let you /show some te8xt/ in italics.

This will let you \_show some text\_ that is underlined.

In the above examples, “show some text” would be bolded in the first sentence, italicized in the second sentence, and underlined in the third sentence. Note that when modifying text using these symbols, there must be a space preceding the first “\*”, “/” or “\_” that you use, with a character immediately following this symbol, and there must be a space immediately after the last “\*”, “/” or “\_”, with a character immediately preceding this symbol. So here are some examples that would ***not*** work:

This \*won’t work\*for bold with no space after the 2nd asterisk.

This\*won’t work\* for bold with no space before the 1st asterisk.

This \* won’t work\* for bold with a space after the 1st asterisk.

This \*won’t work \* for bold with a space before the 2nd asterisk.

You can change the way the text looks in more than one such way at a time—for example, you can bold and italicize, or you can italicize and underline. However, if you use more than one type of modification in the same section of text, the order of the symbols appearing in the beginning and end must mirror each other. Whichever of these symbols comes first on the left, must come last on the right, whichever comes second on the left, must come second to last on the right, etc.

This will let you \*/\_put some text\_/\* in bold, italics, and underline.

In the example above, notice how the asterisks stay on the outside, the backslashes stay in the middle, and the underscores stay snug beside the text. The phrase “put some text” will now be bolded, italicized, and underlined.

## Keyword Lines

The second type of line in a GT program is a keyword line. Keywords always start with an asterisk character (i.e “\*”). There is only a fixed set of different keywords that you can use. Keywords either act as commands, or modify the behavior of other commands. They tell the program to do something. For example:

\*quit

This is a keyword that stops the currently running program file. GT programs can consist of one text file or multiple text files that refer to each other. If this \*quit keyword is encountered and your GT program just consists of one program file, that program file will immediately stop running and the GT session will be over. You don’t have to type \*quit at the end of your program to make it stop running (the program will stop automatically when it reaches the end of your main file), but \*quit can be a handy feature in more advanced programs. We’ll explain why in more depth later.

### *Keyword Configurations*

Many keywords have configuration options that can be set. For instance, if you’re using a keyword to ask the user a question, you’ll need to specify what question you’re actually asking. The most important configurations that keywords have are set using a colon (“:”) right after the keyword. Below is a simple example. To ask a question, one uses the question keyword, which is written \*question. But, you’ll need to make it clear what you want the text of the question to be, so you’ll use a colon to configure this. Here’s a simple example of a GT program that asks a question.

\*question: How are you feeling today, Dracula?

[▶Run](http://www.guidedtrack.com/programs/9uwq1yl/run)

When run, this program will simply ask the user “How are you feeling today, Dracula?” and will display a text box where the user has to type their answer. There will also be a submit button which the user will have to click when they are done typing.

## Comment Lines

The third type of line in a GT program is a comment. Comments are merely for taking notes, which someone reading your GT text file can see later. They don’t alter what the program does. You can use comments to record things you want to remember about your program, give instructions to other people who read the GT file, etc. Comments are written as follows:

--this is a comment. It’s just for developers to read.

Text, keywords and comments can all be used in the same GT program. For instance:

Lose hope, all ye who enter here!

--We’ll have to add more content here later.

\*quit

Hi there, funny bones!

[▶Run](http://www.guidedtrack.com/programs/nfsmog9/run)

When run, this program displays the text “Lose hope, all ye who enter here!”. The comment line does nothing (since it merely acts like a note). Then the keyword \*quit causes the run to exit immediately (so “Hi there, funny bones!” will never get shown to the user).

## Indentation

“Indentation” is when one or more tabs (i.e. indents) are placed at the beginning of a line to offset it from the lines before it. Note that **indentation must be created using tabs**. If you use spaces instead to achieve the indents, the thing you’re trying to do won’t work, even though the code itself will look the same to the naked eye (a tab *looks* like a bunch of spaces, but it isn’t the same thing in GT).

GT programs use indentation for a special purpose. If a GT line is indented so that it starts to the right of the GT line that’s just above it, it means that the indented line “is part of” or “belongs to” the line above it. For instance:

\*question: What’s your least favorite number?

\*type: number

[▶Run](http://www.guidedtrack.com/programs/pbrd325/run)

When these two lines are run, the \*question keyword displays a question (“What’s your least favorite number?”) to the user. There is also a text box that the user can type their answer into. But because the type of the question (determined by the \*type keyword) is “number,” the user is only able to put a number in the text box (the user won’t be able to submit their answer unless it’s a number). The fact that \*type is indented beneath \*question indicates that the \*type keyword is part of/belongs to the question.

On the other hand, consider the following program:

\*question: What’s your least favorite number?

\*type: number

The result would not be the same as the GT program we just saw above. This program would give an error because \*type doesn’t make sense when it is not part of a question, and without the indentation, GT doesn’t realize that \*type is referring to the question. As you can see, your program’s behavior can change a lot based on the indentation used on each line.

Let’s look at another common usage of indentation. Consider the following program, which asks a multiple choice question to the user:

\*question: What’s better, ham or butter?

Ham

Butter

It’s all the same to me.

[▶Run](http://www.guidedtrack.com/programs/7vlfc08/run)

In this case, the question “What’s better, ham or butter?” will be presented to the user, and they will be given the choice between three options which they can select (“Ham,” “Butter” and “It’s all the same to me.”) The user will have to select one (and only one) of these options, at which point the program will continue on.

Keywords can also have comments and multiple keywords indented beneath them, which alter their behavior. Take a look at this example:

\*question: How many marbles are in the bag?

--later, let’s change this to something besides marbles.

\*type: number

[▶Run](http://www.guidedtrack.com/programs/3li8s6n/run)

Here, \*question has two lines indented beneath it. The comment, which begins with “--” is just a note and doesn’t actually do anything when the program is run. The keyword \*type has the option “number,” so the user’s response must be a number.

Different types of keywords require and**/**or allow different things to be indented beneath them. For the \*question keyword, you can indent a \*type keyword beneath it in order to specify the type of this question, though this is not required. But trying to indent \*image, for example, below \*question would not be allowed, because an image cannot be an answer to the question.

**Complex Indentation**

Indentation can be more complex than the examples we’ve just seen. For instance, you can use multiple levels of indentation. That is, indented lines can themselves have lines indented beneath them. To finish up this section, let’s now look at a larger example of indentation:

Hi there!

\*question: What’s your favorite color?

Chartreuse

Yuck

Mauve

Awful

Taupe

What’s taupe?

Some other stupid color.

That’s nice! My favorite color is clear.

--this is a comment, it’s really just a note.

\*question: If you had dogs, how many would you have?

\*type: number

That’s a stupid number of dogs to have.

[▶Run](http://www.guidedtrack.com/programs/8ez7uyd/run)

When run, the program does the following:

* It displays the text “Hi there!”
* It also (at the same time) displays a question to the user (“What’s your favorite color?”), which appears beneath the text “Hi there!”
* The user is additionally presented with four options they can choose among.
* Once the user picks one of these options, the program then displays to the user a particular line of text, dependent on which answer they choose.
* At the same time (appearing immediately after the text) another question is displayed to the user (“If you had dogs, how many would you have?”) and the user is presented with a text box in which they enter a response, which has to be a number.
* Finally, the text “That’s a stupid number of dogs to have.” is displayed to the user.

Don't forget that there are some keywords that work on their own, and some keywords that actually modify the behavior of another keyword. Guided Track won't recognize the latter as valid if you don't indent them under an appropriate keyword for them to modify (\*type for \*question, as in the above example)!

Also keep in mind that some keywords (like \*if, and \*while) may have any non-modifying keyword indented beneath them (for example, a \*question keyword can be indented beneath an \*if keyword). In these cases though, the indented keywords are only executed if the 'owning' keyword’s conditions are met, but not otherwise. For example, if you want a question to only appear for users who are above 18, you might have a line like \*if: age>18, with a risqué question indented beneath that. This question will only pop up though if the age condition is met first.

Keep these things in mind, and you'll be comfortable working with indented commands in no time!

## Adding Blank Lines

Note that in prior examples**,** there are blank lines in the script, between lines with text. These blank lines are not necessary, and are only there to make the program more readable. The following program would do *exactly* the same thing if it had blank lines between the questions and the text (as written above):

Hi there!

\*question: What’s your favorite color?

Chartreuse

Yuck

Mauve

Awful

Taupe

What’s taupe?

Some other stupid color.

That’s nice! My favorite color is clear.

--this is a comment, it’s really just a note.

\*question: If you had dogs, how many would you have?

\*type: number

That’s a stupid number of dogs to have.

But this is both ugly and difficult (for you, the programmer) to read. You should use line breaks (blank lines) to make your programs prettier. The only time lines breaks change what your program *does*, is when they are used to between two lines of text. So:

Well, what do you know?

If it isn’t Mr. duck.

[▶Run](http://www.guidedtrack.com/programs/coy8qpl/run)

Would display “Well...” and “If it isn’t...” on adjacent lines, one on top of the other, whereas:

Well, what do you know?

If it isn’t Mr. duck.

[▶Run](http://www.guidedtrack.com/programs/kh9cseo/run)

would display the same thing, but with a blank line between them. So, when you’re using multiple text lines in a row, you do need to make sure that the blank lines you use reflect what you want the user to actually see.

# **ADVANCED OPTIONS**

## Customizing the Look of Your Program (for CSS Smarties)

CSS smarties can add custom \*classes beneath individual elements of GuidedTrack, or beneath the \*settings keyword.

For example, if you're embedding a GuidedTrack program and want just the \*header on the first screen of your program to match those of your website, you can define some classes on your website and then do something like…

\*header: Hello Person!

\*classes: alert, alert-success

By indenting \*classes under just about any individual elements (buttons, questions, headers, even text), and adding the CSS code, you can snazzify your program so that specific elements look just the way you want them. The classes you create will appear in the HTML that GuidedTrack generates. Then, you can include rules for these classes inside the CSS of your website.

If you want to change something globally in your program, you can indent some options beneath the \*settings keyword. For example, if you want to right-align all the text in your program you can write something like the following:

\*settings

\*classes: text-right

Of course you can also override all of GuidedTrack's CSS on your website. But, if you want to override elements using the \*settings menu, or just have one individual element like a single button be rebelliously unique, \*classes has got you covered.

## Sharing a “Run” Link with Custom-Loaded Variables

Imagine this scenario. You have a GuidedTrack program that you want to share with the world, but you want to give slightly different versions of it to different groups of people. No problem. You don't need to create multiple GuidedTrack programs in order to accomplish this goal, you just need to tweak the share URL a little bit.

### *Adding a single text variable*

Let's say your teenage daughter wants to start dating. You're sort of okay with that, but you want each date to first take an extensive survey that includes criminal history questions and compatibility scales to ensure they're a good fit for your little poodlesticks. You want to personalize it a little by adding a variable that uses their name every now and then, so they don't think you're weird. Rather than include their name as a variable in the program (which you'd then have to change for the next person who wants to date your daughter), you can add it to the URL like this:

https://www.guidedtrack.com/programs/uniqueprogramID/run**?name=Danny**

The wording in bold font is what you would add to the end of your share URL in order to ensure that the variable "name" is defined as "Danny." Then, you just need to make sure your program uses this variable, with plenty of references like "Howdy, {name}!"

### *Adding a single numerical variable*

The process for adding a numerical variable to your program is pretty similar. Here's an example:

https://www.guidedtrack.com/programs/uniqueprogramID/run**?x=4**

You can define x as any number that you want, including decimal and negative numbers. Just don't use any commas.

### *Adding a single collection*

Suppose you wanted to do the equivalent of the following command, but through the URL:

>> beatles = ["John", "Paul", "George", "Ringo"]

You just have to ensure you add brackets after the name of your variable, like this: beatles[]. You also separate each item in the collection using an ampersand.

https://www.guidedtrack.com/programs/uniqueprogramID/run**?beatles[]=John&beatles[]=Paul&beatles[]=George&beatles[]=Ringo**

### *Adding multiple variables*

Suppose you wanted to do the equivalent of this in the URL:

>> x = 3

>> y = 4

>> z = 5

To do this through the URL, you'd separate each of the 3 variables with an ampersand:

https://www.guidedtrack.com/programs/uniqueprogramID/run**?x=3&y=4&z=5**

# **ASKING QUESTIONS**

As we’ve seen, there are a variety of different types of questions that can be asked, including multiple choice questions that require the user to select one option from a fixed set of options, questions that require them to enter text into a text box, and number questions which let the user enter any number. It’s important to understand how to control what type of question will be asked.

## Standard Multiple Choice Questions

If you’d like to provide users with a multiple choice question, you have to indent the available answers below the question.

\*question: Ever dance with the devil in the pale moonlight?

Yes

No

Get out of my face, joker!

[▶Run](http://www.guidedtrack.com/programs/1rj8l69/run)

In this example the user will be presented with a multiple choice question and must click on one of the three possible answers of their choosing. There is no text box for them to type in. Once they click an answer, they will be brought to the next screen.

### 

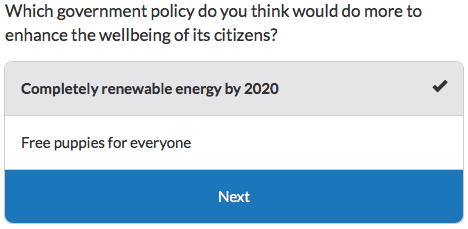
### *Have Users \*confirm Their Selection*

Ordinarily, when a user clicks a multiple choice answer response they immediately progress. With \*confirm, they select their response and then click a 'Next' button to advance.

If you have a long series of multiple choice questions, it can be easy for users to get sloppy, misclicking an answer here or there.

You can help users out by adding [\*back buttons](#h.tupsdph8mhot) to your program so they can correct a mistake, and you can also have them \*confirm their selection.

With the new \*confirm keyword, users select a multiple choice option and it stays highlighted on their screen. Then, they click 'Next.' This 2-step process should reduce mistakes.



To use \*confirm, simply enter it beneath a multiple choice question:

\*question: Select one

\*confirm

Yes

No

## Standard Text Box Questions

The general rule of thumb is that when nothing has been indented below the question, it will default to a text box question. Let’s look at some examples.

\*question: Ever dance with the devil in the pale moonlight?

This will produce a text box question, allowing the user to enter arbitrary text as a response. Since no information is specified (using \*type or otherwise) about what possible answers might look like, GT assumes that any text answer is valid.

You can also provide users a text box question by specifying the \*type as “text”. Take a look below.

\*question: Ever dance with the devil in the pale moonlight?

\*type: text

This will produce exactly the same result as the first example. We’re simply being more explicit here about the fact that this is a question where the user responds with text.

### *Paragraph Text Box*

Using a simple text box will provide users with a small space in which to write a brief answer, but what if you’d like users to write several sentences, or even a whole paragraph? By specifying the \*type of the question as a “paragraph,” you can give users a larger space in which to type.

\*question: Describe your strangest childhood culinary experience.

\*type: paragraph

In the above example, users will be given a lot more space to record their thoughts.

### *Number Only Responses*

If you only want users to enter a number as their answer, you can specify that using the \*type keyword and “number”.

\*question: If you had dogs, how many would you have?

\*type: number

That’s a stupid number of dogs to have.

In this example, users are provided a text box, but only answers that consist solely of numbers will be accepted. The user could type 3, 3,000, or even -3 or 3.64.

### *Allowing Blank Responses*

By default, users are required to provide a response to any question you give them, unless you decide to accept “blank” answers, that is, no response at all. Take a look:

\*question: Please tell me your deepest darkest secret.

\*blank

In the above example, users will be shown the question, but because blank inputs are allowed they don’t have to answer. They can go right onto the next screen when clicking “Submit,” even if they haven't typed anything in response to the question.

Note: the ordering of your keywords doesn’t always matter. In this example, it will make no difference if you put the \*blank keyword first or the \*type: paragraph keyword. The question will work the same either way.

\*question: Please tell me your deepest darkest secret.

\*blank

\*type: paragraph

### *Adding Text \*before and \*after*

You can add text immediately before and immediately after the box in which users can write an answer to a question. It looks like this:



These keywords are easy to add.

\*question: Please fill in the blank.

\*before: When I see

\*after: I get excited!

They also work with any other question add-ons. So, if you'd like users to enter a number, but want to provide a dollar sign for them, simply enter something like the following:

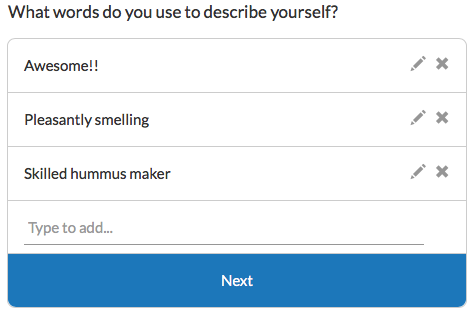
\*question: How much money will you give me?

\*type: number

\*before: $

### *Allowing Users to List \*multiple Answers to a Single Question*

Some questions have more than one answer. For example:



When asking users a question that requires list-like responses (e.g. asking them to list their favorite things, their brainstormed ideas, or their bank account numbers - no, don't do that one), it's easiest to use the \*multiple keyword. Just add it beneath a regular text box question, like so:

\*question: Which of your dance moves most impress the ladies?

\*multiple

When users see the question, they'll be able to add multiple responses, edit any of the responses on their screen, and delete responses, as the little icons in the example imply.For example, in the picture above, the user first saw the question, "What words do you use to describe yourself?", then wrote in "Awesome!!", "Pleasantly smelling," etc.

## Standard Checkbox Questions

If you want your user to have the option of selecting multiple answers, you have to indent the available answers below the question and ensure the \*type of question is “checkbox.”

\*question: Which of the following pets do you have?

\*type: checkbox

Dog

Cat

Hamster/other small furry thing

Fish

Leopard, lion, panther

None of the above

In this example the user will be presented with a checkbox-style question and may click on as many of the answer options as they desire. Once they select their answers and click a big “Next” button, they will be brought to the next screen.

## Standard Slider Questions

You can also provide your users with a question that presents an interactive slider, allowing them to move the slider along a line to the appropriate choice. Simply set the \*type as “slider.”

There are two basic types of sliders: discrete and continuous.

With discrete sliders, you provide specific options for the user to select from. Here’s an example:

\*question: Do you like bacon?

\*type: slider

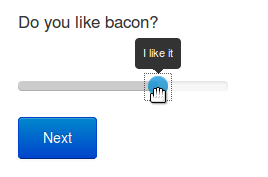
Hate it

It's okay

I like it

I love it

In this example, users will see a question that says “Do you like bacon?” Beneath that, they’ll see a line running from left to right. They can slide their cursor over the line to see different options and choose their answer. The cursor will snap into place at each of the four answer choices.



When no answers are provided to a slider, it becomes a continuous slider. Continuous sliders allow the user to select anywhere in a numerical range. By default, this range goes from 0 to 100, but this can be specified by using the \*min and \*max keywords for the slider. Here are a couple examples:

\*question: How sure are you from 0 to 100?

\*type: slider

\*question: How sure are you from 1 to 10?

\*type: slider

\*min: 1

\*max: 10

In the first example, the slider will flow freely along a continuum of 1 to 100. In the second, the slider will flow freely from 1 to 10.

You can also use the \*before and \*after keyword to add special text on either end of the slider line. For example:

\*question: How sure are you from 1 to 10?

\*type: slider

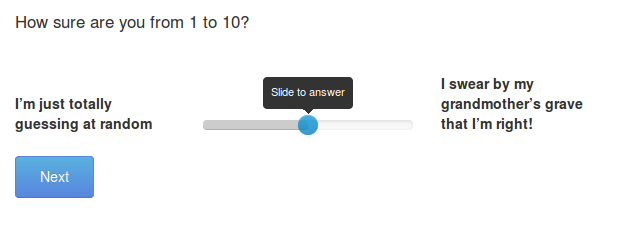
\*min: 1

\*max: 10

\*before: I’m just totally guessing at random

\*after: I swear by my grandmother’s grave that I’m right!

Here’s how this last question looks:



## Multiple Choice Where What Happens Depends on the Answer

When you want to make the entirety of what happens to the user depend on his or her answers to questions, you use multiple indentation, creating a “choose-your-own-adventure” style of experience. Take a look at the following example:

\*question: Which of these animals is least awesome?

Goat

So, you’re hating on goats

\*question: What have goats ever done to you?

Fish

Fish disfavorer! Get out of my face.

\*quit

Now for something completely different.

[▶Run](http://www.guidedtrack.com/programs/ceq3anv/run)

In this case, the answer the user selects to the question “Which of these animals is least awesome?” will determine what happens next. If the user selects the answer “Goat” to the multiple choice question presented to them, they’ll next see the text “So, you’re hating on goats” and then they’ll be asked the question “What have goats ever done to you?” Once the user answers, there are no more lines indented beneath “Goat”, so the program continues past the end of the question. That means that the user will then have “Now for something completely different.” displayed to them.

If instead of selecting “Goat” the user selected “Fish,” they would see the text “Fish disfavorer! Get out of my face.” and then the program would immediately (with no delay) stop running due to \*quit being reached. The phrase “Now for something completely different.” would not be displayed to the user, due to the \*quit keyword causing the program to exit this GT run immediately.

Indentation can go as “deep” as you like. For instance, take a look at this example, in which what happens is totally determined by the answers given to the questions at each indentation level:

\*question: Pick a number

1

\*question: Pick a letter

a

\*question:Pick an animal

Dog

Cat

b

\*question: Pick an animal

Bat

Rat

2

That’s enough for today!

[▶Run](http://www.guidedtrack.com/programs/1yt63zf/run)

The program will begin by saying to the user “Pick a number” and presenting two options for them to choose from (“1” and “2”). If the user selects “1,” then the lines of GT that are indented beneath the “1” answer run next. If instead the user had selected “2,” what would happen next would be determined by the line indented beneath the “2” answer. In the case of selecting “1,” the user would receive another question (”Pick a letter”) and then present the user with the two options, “A” and “B”. If he or she then chooses “A,” the response “Pick an animal” will follow, with the options “Dog” and “Cat”. If instead the user were to choose “B” at this stage rather than “A”, he or she would again get the response “Pick an animal,” but this time with the options “Bat” and “Rat”.

We see that by using indentation, we can dictate what the user experiences depending on how he or she answers questions.

## Text Boxes Where What Happens Depends on the Answer

Sometimes you might have a guess as to what a user will type in a text box. If so, you can choose to tailor what happens next in your program based on what they put in the text box.

\*question: What is the biggest city in New York?

\*type: text

New York

Brilliant!

ny

Excellent!

Jupiter

You’re an idiot!

\*other

Sorry, you are incorrect.

In the above example, it’s important to write \*type: text so that the user will see a text box in which to write their answer, rather than a multiple choice question. In this example, users who type “New York” will then see the text “Brilliant,” whereas users who type “NY” will then see “Excellent!” and so on. Users who write something “other” than what the programmer expected will receive the message “Sorry, you are incorrect.” Without adding the \*other keyword, users who wrote something other than one of the select answered would simply see the next screen, without a tailored response.

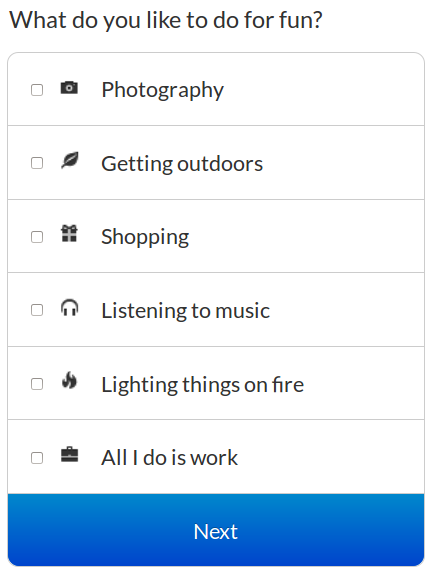
Note that the options you provide are not case sensitive, so if the user put “NeW YoRk” in the text box, they would still see the response “Brilliant!”, just like they’d get if they had typed “New York”. Also, the ordering of your keywords doesn’t matter here. You could have put the \*other keyword and its indented response above Jupiter and its indented response. Either way, the question would function in exactly the same way.

## Question Add-Ons

There are several other interesting ways you can modify a question.

### *Adding an \*icon*

You can add pieces of flair to multiple choice and checkbox questions, like this:



To add an icon to an answer choice, find the icon you like from this link:<http://getbootstrap.com/components/#glyphicons>

Then, indent the keyword \*icon: beneath the answer option and type in the full name of the icon. Below is an example:

\*question: What next?

Continue in this section

\*icon: glyphicon-arrow-right

Return to the main menu

\*icon: glyphicon-home

### *Displaying answer options from a collection of answers*

The \*answer keyword works for multiple choice, checkbox, and slider questions.

Consider this scenario - You're lazy and want to type as little as possible in your program. You have to ask your users 10 yes/no/maybe questions in a row. Rather than type the answers each time, you save your answer options to a [collection variable](#h.6evhh8hfcft) and then use \*answers, like this:

>>options=["yes", "no", "maybe"]

\*question: Have you ever seen the rain? Comin' down on a sunny day?

\*answers: options

\*question: Do you ever feel like a plastic bag, lifting through the wind, wanting to start again?

\*answers: options

In the above example, users will see each of these multiple choice questions along with the answer options of "yes," "no," and "maybe."

Here's a second scenario - you have a program that allows users to create their "Bucket List," all the things they want to do before they die. It's totes amazeballs. You know that some unimaginative users will come up with two goals though and the over-achievers will have hundreds. At the end of the program, you want to show users their entire bucket list and ask them to select the one thing they'll do first. That's where \*answers will come in.

Here's an example:

>>bucketList=[]

>>userStatus="writing"

\*while: userStatus="writing"

\*question: What awesome thing do you want to add to your bucket list?

\*tip: When you're done, just leave this box blank and click "Submit" a final time.

\*save: newIdea

\*blank

>>userStatus="done"

\*other

>>bucketList.add(newIdea)

Great! Keep going!

Your bucket list is below.

\*question: Select the one thing you wanna do first

\*answers: bucketList

\*save: firstThing

Awesome! Now get out there and {firstThing}!

In this program, the \*while keyword allows the user to add as many things to their bucket list as they want, until they leave the box blank. Each of their ideas is added to a collection called "bucketList." When they finish, they see a question that asks for the goal they want to achieve first, and they see a multiple choice list of all their bucket list ideas.

### *Giving a Tip*

Sometimes you might want your question to include an example, or a helpful tip. Doing so is easy with the \*tip keyword.

\*question: What was your most recent strange dream like?

\*tip: For instance, you might say “I dreamt I turned into a toaster.”

The user will see the question asked in normal text. Below it, they will see a lighter-colored text that begins with the words “For instance.” Then, they will see the answer box in which to write.

### *Providing Default Answers to Questions*

Giving users a little boost in answering a question, or allowing them to revise a previous answer, is possible with \*default.

You use \*default when you want to pre-populate a question text box with an answer. Or, when you want a multiple choice, checkbox, or slider question to be pre-selected.

**Text and paragraph questions**

When a text box is clicked by your users, your default text won't disappear (it's not hint text), but rather your users can submit the answer as-is immediately by clicking submit or edit the text before submitting.

Here's how the code might look:

\*question: Finish the start of this story.

\*default: It was a bright cold day in April...

\*save: story

And here's how the above code would look to the user:



**Checkbox questions**

To use default with a checkbox question, just add all the options you want selected, separated by commas.

\*question: Which of my super awesome hourly newsletters do you want to sign up for?

\*default: Catasaurus Rex, Catopian Dreams, Kitty Kaleidoscope, Meow Meow Times

\*type: checkbox

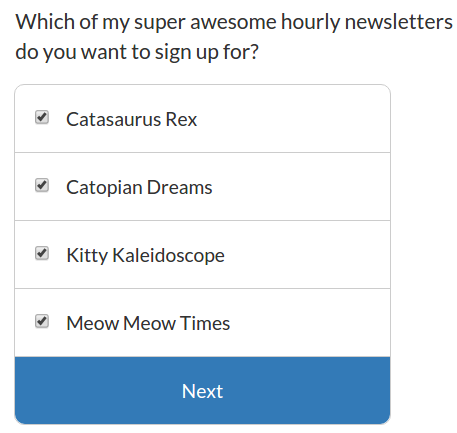
Catasaurus Rex

Catopian Dreams

Kitty Kaleidoscope

Meow Meow Times

Here's how the above code would look:



**Multiple choice questions**

To give a default answer to a multiple choice question, just write the exact text to the one answer you want preselected.

For example:

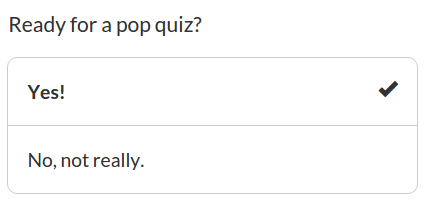
\*question: Ready for a pop quiz?

\*default: Yes!

Yes!

No, not really.

Here's how this would look:



**Variables**

You can also use variables with \*default, like so:

\*question: Make revisions to your story below

\*default: {story}

### *Save a User’s Answer*

It’s often nice to remember a specific answer a user has given, such as their name, so that you can use it within your program.

\*question: What’s your name?

\*save: name

Welcome, {name}!

In the example above, we used the \*save keyword to save the user’s answer. If the user had typed “Ducky,” they would next see, “Welcome, Ducky!” To save the user’s answer, you have to type in a unique name or code for the answer you want to save. We wrote “name” to the right of the \*save keyword and later put “name” in braces because that was the code for the answer we wanted to recall. You can use any kind of code for the answer you want to save, just as long as it only contains letters, numbers, and no spaces. Remember that the codes *are* case sensitive.

You can also use the \*save keyword with math equations:

* + addition
* - subtraction
* \* multiplication
* / division
* () parentheses to set operator preference
* ^ to the power of

\*question: How old are you?

\*save: age

That’s about {(age-2)\*4+21} in dog years

In the above example, if a user entered an age of 12, their dog years would be calculated like this: (12-2) x 4 + 21. Their dog years would be 61.

It is also worth mentioning that spacing does not matter when performing calculations. For example, in order to make it easier to read for yourself, you might want to add spaces around the \* and + and write the above equation as:

That’s about {(age-2) \* 4 + 21} in dog years

Here are two more examples, using the caret (^) to do exponents and square roots:

Your age squared is {age^2}

The square root of your age is {age^1/2}

If you like, you can see examples of using the \*save keyword with the [\*if keyword](#h.4ieiokufb9r9). This is a more complex way of using \*save, which we’ll go over later on.

### *Shuffle Answer Options*

Did you know that people select the first option in a multiple choice question more often than any other brilliant answer you could have given them? Is it out of laziness? Subconscious beliefs that the first option is best? Reckless captivation of the heart? We don't know.

What we do know, is that you can now remedy this problem by randomizing your answer options so that your results aren't skewed by position bias.

It's rather easy:

\*question: Would you rather...

\*shuffle

Uncontrollably sneeze after every sentence you spoke

Uncontrollably fart every time that you laughed

In the above example, users will either see the sneeze option first or the latter possibility.

Just indent a \*shuffle beneath any multiple choice, checkbox, or slider question (numeric values excepted) that you'd like to randomize.

Note though, that \*shuffle is probably not useful for answer options that lie on a continuum (e.g. cold, cool, warm, hot). Shuffle these only if you want to annoy your users.

### *Summarizing Entered Data*

When users answer a question, you can save the question and their answer in order to display both back to them later on. This is great for reminding the user of what they answered to a question previously, or summarizing their previous answers for them. To do this, you use the \*tags keyword and the \*summary keyword.

\*question: Do you like Tiny Tim, considering all he’s done for you?

\*tags: TinyTimQuestion

Here is the question you answered about Tiny Tim, and your answer to that question:

\*summary: TinyTimQuestion

In this example, we’ve indented a \*tags keyword underneath the question we wanted to recall later. The keyword \*tags marks the question as well as the user’s response with tags, and “TinyTimQuestion” is what we decided to use as our tag in this case. Below the text “Here is the question you answered...” the \*summary keyword was placed, with the name of the tag we want to recall here. After answering the Tiny Tim question, the user will see “Here is the question you answered...”, then they will see the text of the Tiny Tim question repeated back to them followed by the answer they provided.

Multiple tags can also be used on the same question and should be separated by commas. Multiple questions can also have the same tag. This allows the user to answer several questions, and then receive a summary of their answers.

\*question: What’s your favorite number?

\*questions: What’s your favorite color?

\*tags: personal

Red

Blue

Greenwich

\*question: What city do you live in?

\*tags: demographics

\*question: Do you work with robots?

\*tags: personal, demographics

Here are the personal questions you answered:

\*summary: personal

And here are the demographic ones:

\*summary: demographics

And here are ALL the questions you answered:

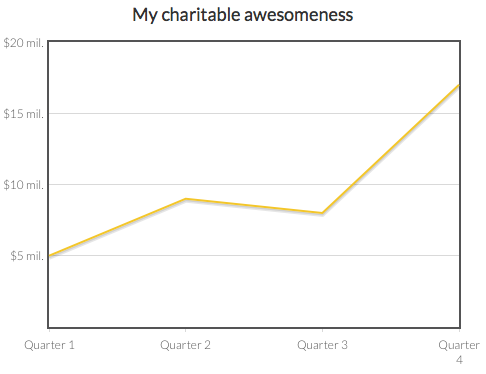
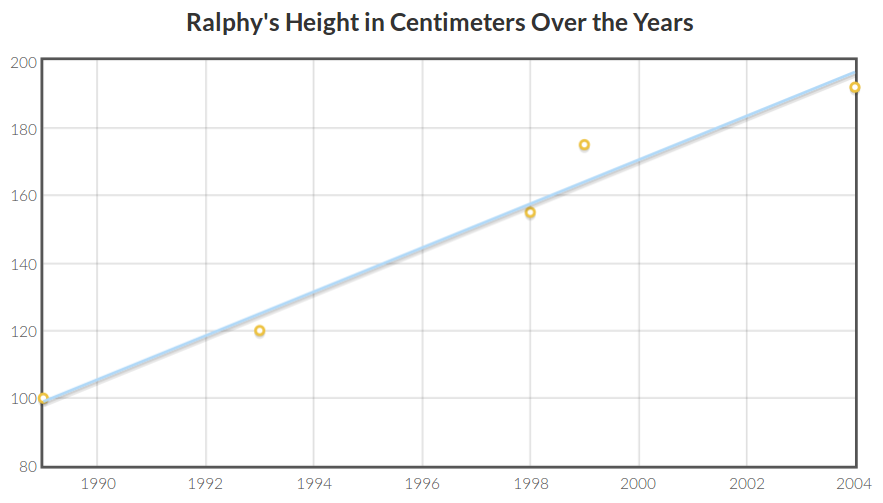
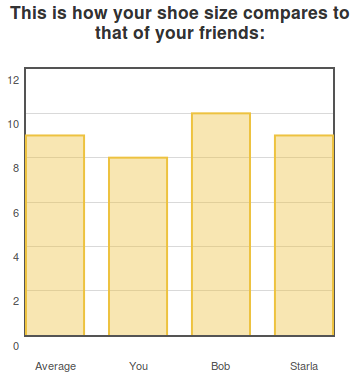
\*summary

At the end of this example, the questions/answers tagged as “personal” will be summarized first, followed by the questions/answers tagged as “demographics.” Some questions/answers will be summarized in both the personal and demographics summaries, because they have both tags. At the end, any question/answer given a \*tags keyword (whether they are configured as “personal” or “demographic”) will be summarized. The question “What’s your favorite number?” will not be summarized though, because it was not given a tags keyword.

# **ADDITIONAL KEYWORDS**

## Charts

Charts provide eye candy for your users. Here's a few samples of how they currently look:



Charts involve 3 main keywords:

1. **\*chart:***You'll need to type in the \*chart keyword and optionally provide a title (e.g. "\*chart: My Chart").*

2. **\*type:** *You'll need to define what kind of chart you'd like to plot. For now, we have "****bar****", “****line****” and “****scatter****”.*

3. **\*data:** *You'll need to provide the data to chart. The data you enter will always be in* ***[****square brackets****]****.*

Optional keywords include:

4. **\*color**: *provide the rgb code to the color you’d like, or for basic colors write the name (e.g. “\*color: red”)*

5. **\*xaxis and \*yaxis**:

* *Set a* ***\*min*** *and* ***\*max*** *value for the y- and x-axes of your chart*
* ***\*position*** *your y-axis left or right, and your x-axis up or down*
* *Add* ***\*ticks*** *to improve the labels displayed on your axes*

Scatter charts also have an additional, optional keyword:

6. **\*rollovers:** *Optional text that pops up when the user rolls their mouse over any of the data points.*

Now let's talk about this square brackets business. Anything in [square brackets] is considered a "collection" and is simply a collection of multiple smaller things. [Collections](#h.6evhh8hfcft) in GuidedTrack are defined as a list of numbers, strings, variables, other collections between square brackets, or a combination of these things.

### *Basic Bar Chart and Charting 101*

#### **Basic Bar Chart**

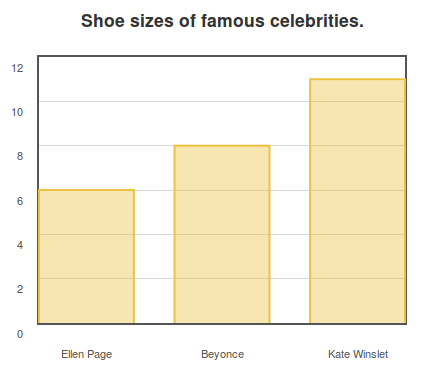
To illustrate a collection and how to use it we'll start with the most basic way you can add data to a bar chart.

\*chart: Shoe sizes of famous celebrities.

\*type: bar

\*data: [[“Ellen Page”, 6], [“Beyonce”, 8], [“Kate Winslet”, 11]]

The above program will display a bar chart that looks like this:



Let's look at the GuidedTrack code again so we can take a look at how collections are used.

\*chart: Shoe sizes of famous celebrities.

\*type: bar

\*data: [[“Ellen Page”, 6], [“Beyonce”, 8], [“Kate Winslet”, 11]]

The entire data needed for this chart is in one big collection (see the square brackets on the outer edges?). This large collection consists of three smaller collections, one for each of the three important pieces of data we want to plot. Each smaller collection consists of the bar label in "quotes", a comma, and the data point. Each smaller collection is separated from the next one with a comma. This is the simplest way to add data to your bar chart.

#### **Using Simple Variables with \*data**

Look how our previous code could change if we instead used a couple variables in our data.

>>ellen = “Ellen Page”

>>Bshoe = 8

\*chart: Shoe sizes of famous celebrities.

\*type: bar

\*data: [[ellen, 6], [“Beyonce”, Bshoe], [“Kate Winslet”, 11]]

The above program would display the exact same chart as the one we made earlier. Here you can see the data keyword contains two variables, "ellen" and "Bshoe." You can substitute variables for pieces of your data collection, so long as you defined those variables previously in the program using the \*save keyword or >>, like we did above. Note when using variables, you do not need quotes (like how the variable "ellen," when used after the \*data keyword, does not have quotes around it).

For a refresher on variables, check out [Introduction to GuidedTrack](https://docs.google.com/document/d/1RKblBEwG7BrMMeTXFITSNDJQsszHotqaE_q-E9q0QHA/edit#heading=h.zcqh86rktvec).

#### **Using a Variable of a Collection with \*data**

We could also use more complex variables in our data. Check out this example now. It produces the exact same chart.

>>ellen = [“Ellen Page”, 6]

>>beyonce = [“Beyonce”, 8]

>>kate = [“Kate Winslet”, 11]

\*chart: Shoe sizes of famous celebrities.

\*type: bar

\*data: [ellen, beyonce, kate]

Here you can see that the variable "ellen" is now itself a collection. We can substitute the variable "ellen" (as well as "beyonce" and "kate") right after our \*data keyword to produce the same chart we had before.

#### **Using a Variable of a Collection of Collections with \*data**

One final example:

>>ellen = [“Ellen Page”, 6]

>>beyonce = [“Beyonce”, 8]

>>kate = [“Kate Winslet”, 11]

>>Celebrity\_Shoes = [ellen, beyonce, kate]

\*chart: Shoe sizes of famous celebrities.

\*type: bar

\*data: Celebrity\_Shoes

This again produces the same chart we showed you earlier. Now, after the \*data keyword, is a variable that represents a collection of collections. You can see how the variable "Celebrity\_Shoes" was defined as a collection of the variables "ellen", "beyonce", and "kate," which were themselves collections.

Now you've seen three ways to use variables with the \*data keyword. You can mix and match all different types of variables or non-variables to produce your charts in a way that is most convenient for you.

#### **Setting the \*min and \*max Values of Your \*xaxis and \*yaxis**

By default, the x- and y-axis of a chart will stretch to a size relative to the largest and smallest bits of data you're displaying.

You can customize the size of your chart with ease. For instance:

\*chart Your test results

\*type: bar

\*data: [[“Your score”, score], [“Average score”, 65]]

\*yaxis

\*max: 100

\*min: 0

Setting a \*min of 0 won't be necessary if your chart data is always in the positive, but it can be useful when you have very specific ranges you would like to set.

You can additionally customize the \*xaxis with \*scatter charts.

\*chart: Ralphy’s Height in Centimeters Over the Years

\*type: scatter

\*data: [[1989, 100], [1993, 120], [1998, 155], [1999, 175], [2004, 192]]

\*rollovers: ["Toddler", "First year of school", "HS Freshman", "HS Sophomore", "College Freshman"]

\*xaxis

\*min: 1980

\*max: 2000

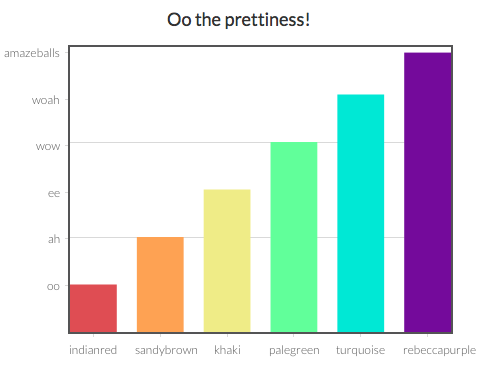
\*yaxis

\*min: 50

\*max: 200

#### **Adding a Pretty Chart \*color**

Splash up your graphical displays with some color.



To start, you can find your favorite color from this website: [http://colours.neilorangepeel.com](http://colours.neilorangepeel.com/)

Copy down the rgb code. For example, if you're a fan of peachpuff, the code you want looks like this: rgb(255,218,185)

When you've got your fav, you simply add it to your chart's \*data using the \*color keyword, like so:

\*chart: Got peachpuff in your life?

\*type: bar

\*data: [["bar 1 label", 4], ["bar 2 label", 5], ["bar 3 label", 6]]

\*color: rgb(255, 218, 185)

In this example, we're only using one color, not six. We'll get to adding multiple colors in a bit.

If you want boring colors like solid "blue" or everyday "red", then you don't need the rgb. You can just type \*color: blue. This will work for many of the basic colors, and a few of the more creatively named.

If you like the color you have, but it's a little too intense, you can add \*opacity to make it somewhat see-through, like this:

\*chart: Seeking less peachpuff in your life?

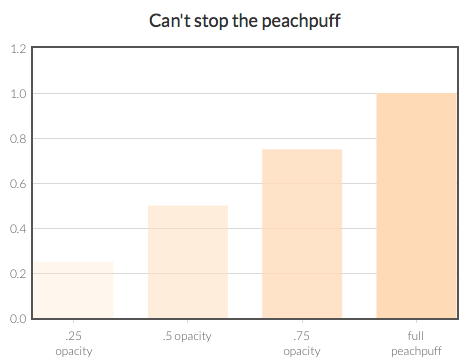
\*type: bar

\*data: [["bar 1 label", 4], ["bar 2 label", 5], ["bar 3 label", 6]]

\*color: rgb(255, 218, 185)

\*opacity: .5

Opacity should be a number ranging from 0 (totally see through) to 1 (totally solid). Here's how peachpuff looks with different levels of opacity:



If you have a custom color you want to use, go for it. The \*color keyword will work with any rgb, not just the ones listed on the website above.

#### Multiple colors and multiple \*data keywords

Now let's return to colors. Remember my chart earlier? With all the super awesome colors? The key to this snazziness is using multiple \*data keywords. Each \*data keyword gets its very own \*color keyword.

If you just indent \*color: yourcolorhere beneath the \*chart keyword, that color is going to apply to the entire chart. If you want to be more specific, then you've got to be more specific with your indents. Check out the below code and see what I mean:

\*chart: Oo the prettiness!

\*type: bar

\*data: [["indianred", 1]]

\*color: rgb(205,92,92)

\*data: [["sandybrown", 2]]

\*color: rgb(244,164,96)

\*data: [["khaki", 3]]

\*color: rgb(240,230,140)

\*data: [["palegreen", 4]]

\*color: rgb(152,251,152)

\*data: [["turquoise", 5]]

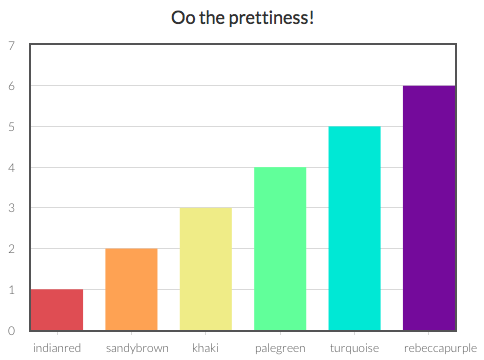
\*color: rgb(64,224,208)

\*data: [["rebeccapurple", 6]]

\*color: rgb(102,51,153)

Instead of lumping all my data points into one \*data keyword, I split them up, so that I could then apply a unique \*color attribute to each \*data point.

The outcome of the above code will look like this:



If you want mostly light blue bars, with just one vibrant orange, then you do this:

\*chart: This 'lil chart of mine

\*type: bar

\*color: lightblue

\*data: [["Fact 1", 12], ["Fact 2", 4], ["Fact 3", 3]]

\*data: [["Super emphasized Fact 4", 18]]

\*color: orange

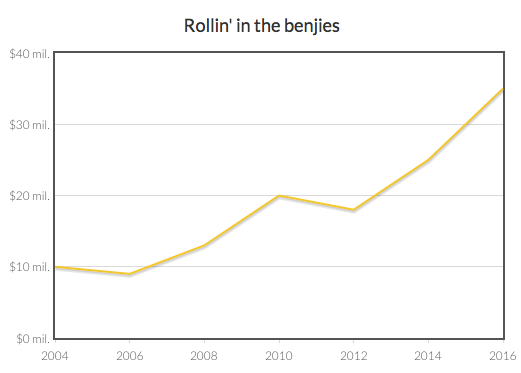
\*data: [["Fact 5", 7], ["Fact 6", 5]]

The lightblue becomes the default color for all the bars, because it's indented beneath the \*chart keyword, but the color for the fourth bar is overridden with an orange color.

#### **Add \*ticks - I.e., Customize the Look and Position of the Intervals on the Axes**

Charts with money are fun, but they're hard to read without $ signs. You can customize your x- and y-axes to say anything you want!.

Like this:



Here's how to do that:

\*chart: Rollin' in the benjies

\*type: line

\*data: [[2004, 10000000], [2006, 9000000], [2008, 13000000], [2010, 20000000], [2012, 18000000], [2014, 25000000], [2016, 35000000]]

\*yaxis

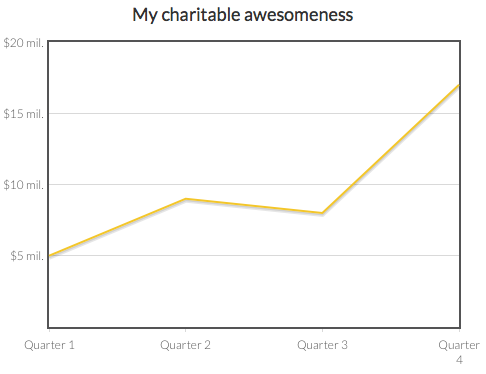
\*ticks: [[0, "$0 mil."], [10000000, "$10 mil."], [20000000, "$20 mil."], [30000000, "$30 mil."], [40000000, "$40 mil."]]

It's the same line graph as the one in the [line graph example](#h.493vi273iqe1), but this time I've added something called "\*ticks" indented beneath the \*yaxis keyword. The \*ticks keyword let's me specify [the numerical value of where I want the tick to go, "the label I want to give that tick"]

In the case of \*ticks, the value comes first and the label comes second. With bar graphs, the reverse is true (bar label, then value). We didn't do this to try to confuse you, but here's a tip: in each case try to put the most important info first. Whereas both the bar label and value are required, the label on \*ticks is optional information. You could also just have something like \*ticks: [2, 4, 6, 8, 10], which would display just the even numbers and label them for you automatically.

By the way, let’s say that I decided to learn the principles of Effective Altruism and give my money to highly efficient charities and save lives. Go me!

My new goal is to chart my quarterly distributions to charity. Instead of having an x-axis with years, I instead want them to say "Quarter 1", etc... like this:



The code is very similar as the earlier code, except now I've also included \*ticks for the \*xaxis:

\*chart: My charitable awesomeness

\*type: line

\*data: [[1, 5000000], [2, 9000000], [3, 8000000], [4, 17000000]]

\*yaxis

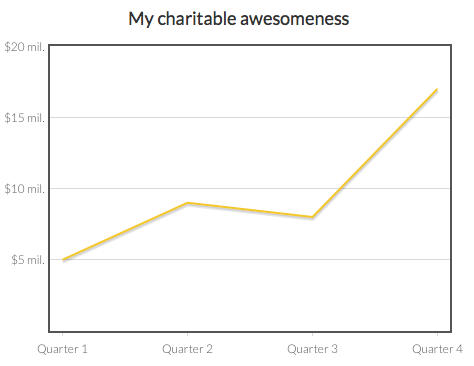
\*ticks: [[0, ""], [5000000, "$5 mil."], [10000000, "$10 mil."], [15000000, "$15 mil."], [20000000, "$20 mil."]]

\*xaxis

\*ticks: [[1, "Quarter 1"], [2, "Quarter 2"], [3, "Quarter 3"], [4, "Quarter 4"]]

You'll also notice that for the first \*yaxis tick I used "" as the label, which means the label is just blank. This can sometimes be helpful for making your chart look a little smoother.

Don't forget you can also use \*min and \*max beneath your \*xaxis and \*yaxis to help smooth things out. For example, the x-axis is looking a little crowded. If we set our \*min at .9 and our \*max at 4.1, they'll be a little more room.



Here's the code with the new \*min and \*max added:

\*chart: My charitable awesomeness

\*type: line

\*data: [[1, 5000000], [2, 9000000], [3, 8000000], [4, 17000000]]

\*yaxis

\*ticks: [[0, ""], [5000000, "$5 mil."], [10000000, "$10 mil."], [15000000, "$15 mil."], [20000000, "$20 mil."]]

\*xaxis

\*ticks: [[1, "Quarter 1"], [2, "Quarter 2"], [3, "Quarter 3"], [4, "Quarter 4"]]

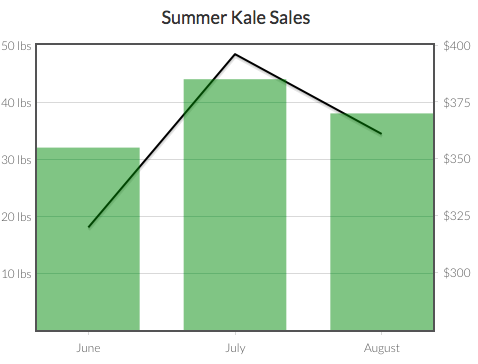
\*min: .9

\*max: 4.1

#### **Multiple Chart Types on One Chart - and Multiple Axes on One Chart**

Let's say you want a line graph AND a bar graph on one chart, cause you're a chart maniac.

For example, you're a farmer charting how many pounds of kale you sold and how much money you brought in. You want your final chart to look like this:



To start, you have two different types of charts. You'll need to add multiple data keywords, one for each type of chart.

\*chart: Summer Kale Sales

\*data: [[6, 32], [7, 44], [8, 38]]

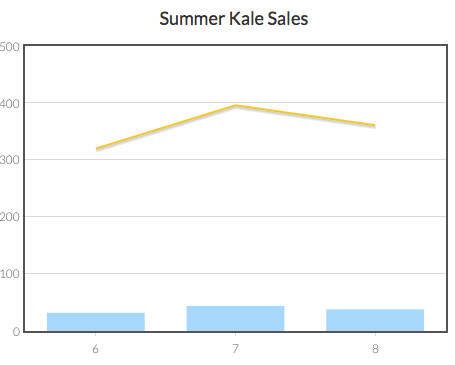
\*type: bar

\*data: [[6, 320], [7, 396], [8, 361]]

\*type: line

In our code above, we've used the numeric values of the summer months (6 for June, 7 for July, 8 for August). We've also indented the \*type keyword beneath each data keyword to specify how to display the data. We're showing the pounds sold as our bar graph data and the income made as the line graph data.

This code results in a pretty ugly chart though:



More work is needed! The number of pounds sold ranges from 32-44, while dollars earned ranges from $320-$396. Each data type will need its own \*yaxis for good chart chemistry.

We can have multiple y-axes and/or multiple x-axes. We just have to be sure to give each a name, and then link their name beneath the data they serve. We also have to tell the program where to \*position each axis: on the left, right, top, or bottom, like so:

\*chart: Summer Kale Sales

\*data: [[6, 32], [7, 44], [8, 38]]

\*type: bar

\*yaxis: pounds

\*data: [[6, 320], [7, 396], [8, 361]]

\*type: line

\*yaxis: dollars

\*yaxis: pounds

\*position: left

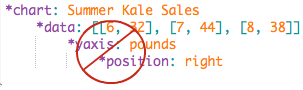
\*yaxis: dollars

\*position: right

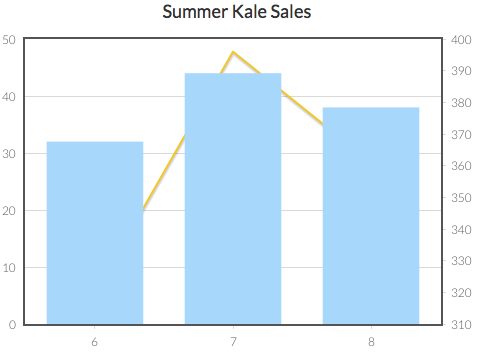
In this code, we've now indented \*yaxis: pounds beneath the bar graph data, because that's the data it's attributed to. "pounds" is just the name we made up, you can call your axes anything. We've also indented \*yaxis: dollars beneath our line graph data.

Next, if you have multiple axes, you should specify their \*position. In this case, we want the pounds' \*yaxis to be on the left side of the chart and the dollars' \*yaxis to be on the right side of the chart.

You always indent the specifications of an axis beneath a \*yaxis or \*xaxis that is indented beneath the \*chart keyword, not beneath a \*data keyword. Axes are attributes of the entire chart, and multiple \*data keywords can share the same \*yaxis or \*xaxis information. That's why, when you want to provide details about the axes (such as their \*ticks, \*position, \*min, or \*max), you include this info beneath the \*chart keyword more broadly. In other words, this is bad:



Okay, now our chart looks like this:



Getting better! But still not great... Let's change the \*ticks on the y-axes so they're more clear...

\*chart: Summer Kale Sales

\*data: [[6, 32], [7, 44], [8, 38]]

\*type: bar

\*yaxis: pounds

\*data: [[6, 320], [7, 396], [8, 361]]

\*type: line

\*yaxis: dollars

\*yaxis: pounds

\*position: left

\*ticks: [[10, "10 lbs"], [20, "20 lbs"], [30, "30 lbs"], [40, "40 lbs"], [50, "50 lbs"]]

\*yaxis: dollars

\*position: right

\*ticks: [[300, "$300"], [325, "$325"], [350, "$350"], [375, "$375"], [400, "$400"]]

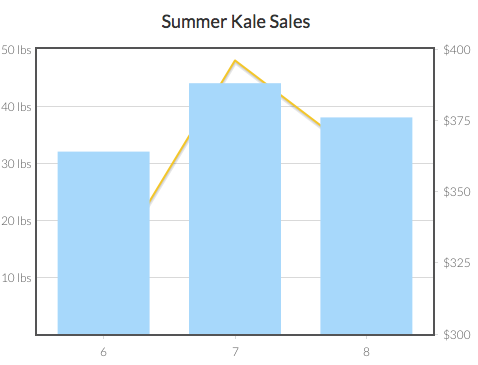
This is the same code as before, except now we've specified how we want the labels to read.

The easiest way to add \*ticks to a chart with multiple axes is to first look at where the ticks naturally fall (in our earlier chart you can see there are lines on the 10, 20, 30, 40, and 50 marks on the left axis).

Once we know where the lines naturally fall, we can work with these existing lines to give them better labels. "10" becomes "10 lbs", "20" becomes "20 lbs", and so on. If you don't like where the lines naturally fall, you can try first adding \*min and \*max values to the dominant axis.

The lines will only appear for one axis, not both. For the second axis, you just have to add labels wherever you'd like them to appear. In this case, we wanted the labels to increase by $25, starting with $300. We added code for the \*ticks at the 300 mark, which would read "$300", another at the 325 mark, to read "$325", and so on.

Now our chart looks like this:



That's pretty good! New \*ticks for our \*xaxis and some colors would improve things though. Plus, it's hard to see the lines through the bars, so we should add some \*opacity.

If we set the \*min for the \*yaxis: dollars to 275, that would also align the dollar values with the lines corresponding to the pounds, which would look swell!...

\*chart: Summer Kale Sales

\*data: [[6, 32], [7, 44], [8, 38]]

\*type: bar

\*yaxis: pounds

\*color: green

\*opacity: .5

\*data: [[6, 320], [7, 396], [8, 361]]

\*type: line

\*yaxis: dollars

\*color: black

\*opacity: .75

\*yaxis: pounds

\*position: left

\*ticks: [[10, "10 lbs"], [20, "20 lbs"], [30, "30 lbs"], [40, "40 lbs"], [50, "50 lbs"]]

\*yaxis: dollars

\*position: right

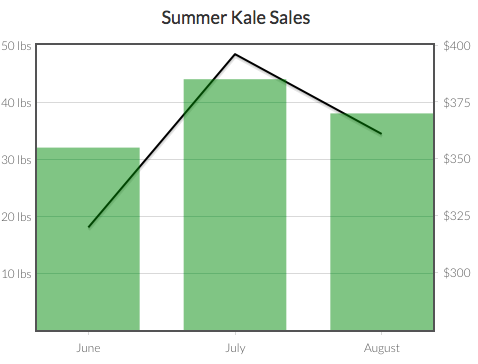
\*ticks: [[300, "$300"], [325, "$325"], [350, "$350"], [375, "$375"], [400, "$400"]]

\*min: 275

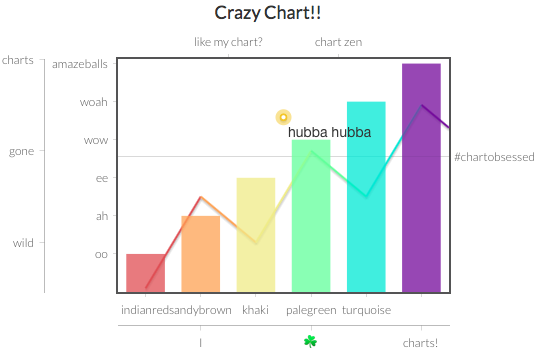
\*xaxis

\*ticks: [[6, "June"], [7, "July"], [8, "August"]]

This final code generates the image we started this section with. For a reminder, it looked like this:



With our code, we didn't need to specify a name for the \*xaxis, because there's only one. If we wanted to though, we could add more. One at the top too. We could even add multiple x-axes at the bottom! We could have 6 y-axes and 7 x-axes just because we were feeling crazy!



### *Scatter Plots*

The code for making scatter plots works similarly to bar charts. You need the \*chart keyword and an optional title for your chart. You also need the \*type keyword (in this case it should say: \*type: scatter). Third is the \*data keyword, which is a bit different from bar graphs. Your data should be organized in this general format: \*data: [[x1, y1], [x2, y2], [x3, y3], [etc...]]

\*chart: Ralphy's Height in Centimeters Over the Years

\*type: scatter

\*data: [[1989, 100], [1993, 120], [1998, 155], [1999, 175], [2004, 192]]

In our example, there are five data points. Each data point is surrounded by [brackets]. The value of the x-axis is written first (in our case, that's the year) and second is the y-axis (e.g. centimeters). Commas separate the values within each data point and they separate data points from each other. Additional brackets surround the entire dataset.

Variables can be substituted for any parts of the data, similarly to how we illustrated above.

#### **Using the \*rollover Keyword with Scatter Charts**

You can add optional text values to a scatter charts so that when users hover their mouse over any of the above data plots, a small box pops up with additional information. Here’s how the code looks:

\*chart: Ralphy's Height in Centimeters Over the Years

\*type: scatter

\*data: [[1989, 100], [1993, 120], [1998, 155], [1999, 175], [2004, 192]]

\*rollovers: ["Toddler", "First year of school", "HS Freshman", "HS Sophomore", "College Freshman"]

In the chart that this example produces, when users hover their mouse over the 1993 data point, it reads, "First year of school." (you can try this example out yourself [here](http://gimbeltech.us7.list-manage.com/track/click?u=9e5957c81cac843c342446e34&id=06e231bb11&e=9d8b220c85)).

The \*rollover keyword is optional. If you include it, you must have as many entries as there are data points. In our example, the first rollover text, (i.e. "Toddler") corresponds to the first data point (i.e. [1989, 100]), and so on.

#### **Adding a \*trendline to Scatter Charts**

By adding the \*trendline keyword, you can show whether your data trends upward, downward, or just flat.

Trendline is off by default. To add a trendline to your graph, use \*trendline, like this:

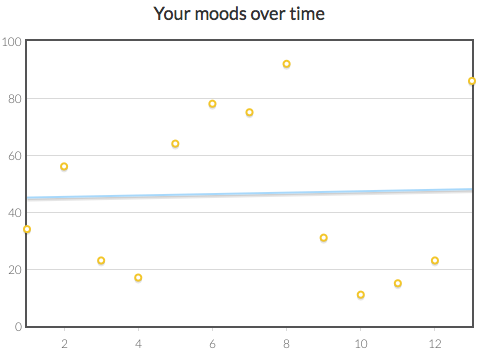
\*chart: Your moods over time

\*type: scatter

\*data: MoodData

\*trendline

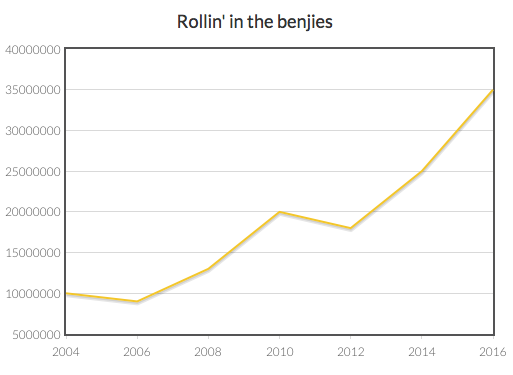
Trendlines look something like this:



### *Line Graphs*

Let's say I want to chart how my millions of dollars have grown over time ('cause I'm saving for a yacht big enough to hold my helicopter).

I can use a line chart like this:



The code for this is pretty simple. You add data similarly to how you would in a scatter plot, indicating the x and y points. You also do \*type: line

Here's the example:

\*chart: Rollin' in the benjies

\*type: line

\*data: [[2004, 10000000], [2006, 9000000], [2008, 13000000], [2010, 20000000], [2012, 18000000], [2014, 25000000], [2016, 35000000]]

## Clearing the Screen

The \*clear keyword allows you to clear the screen of everything on it. For example, you can have a program like this:

You’ll now have 10 seconds to find the cow in this image.

\*image: http://www.scenicreflections.com/files/ispy\_animal\_1024.jpg

\*wait: 10 sec

\*clear

\*question: Where was the cow?

In this sample program, once the timer is up the instructions and the image cleared from the screen and the user only sees the question, "Where was the cow?"

The \*clear keyword operates much differently from [creating a page break](#h.f2ctivxug144) using \*question or \*button. With these latter options, the user has to click a button in order to proceed to a new screen. With \*clear, the screen clears automatically.

Note: Without the \*wait keyword in place in this sample program, the user would not ever see the instructions and image as the clear would occur immediately. The user would only see the question. In most cases, you'll want to use the \*clear keyword after a \*wait, unless you’re using it to clear [maintained text](#h.jua2wrmq7oqc).

## Emailing Users

You can email users any kind of message you want (content from your program, their results from your program, a message just to say hi). If users have a GuidedTrack account and are logged in, the below code will email them automatically. To learn how to ensure your users have an account and are currently logged into it, [click here](#h.8xvm8ca5k2pr).

I’ll send you an email now.

\*email

\*subject: An email for you

\*body

Hi,

I just wanted to send you an email.

\*This statement is in bold, because it’s important.\*

Bye,

Me

So long as you hit “Allow” your email will arrive shortly in your inbox.

The most important parts of an email are the keywords \*email, \*subject, and \*body. You must set up an email with all three parts, being mindful of tabbing \*subject and \*body beneath the \*email keyword, and tabbing the content of your email beneath \*body, as the example shows. Note: \*subject has a colon, but \*body does not.

In the above example, the user will see the line, “I’ll send you an email now.” Then they will see a “Next” button. After hitting the button, they’ll see a popup prompt, asking whether they give their permission for the program to email them. The users must hit “Allow” in order to receive the email. Once they are done with the prompt, they will see the next line of the program, which in this case begins with “So long...” The email they receive will look exactly like that which is tabbed beneath the \*body keyword. The \*subject from our example would read “An email for you.”

Feel free to include any other GuidedTrack code in the body of the email. It will appear in the user’s email exactly as you intend it to. For example, in the above email, the sentence with \*’s around it, indicating that sentence should be bolded, will appear like this: **This statement is in bold, because it’s important.**

If users are not [logged in](#h.8xvm8ca5k2pr) to GuidedTrack, or you would like to send an email to yourself instead, you can use the \*to keyword.

\*email

\*subject: Drivel from one of my users

\*to: [myemail@emailaddress.com](mailto:myemail@emailaddress.com)

\*body

Someone said this about my program:

{complaint}

-Me

The \*to field can also contain a variable. So, you could ask user's their email address, save it as a variable, then email them some content from the program. Of course, you should always ask them first if it’s okay to email them and only send the email if they say yes.

## Experiments

The \*experiment keyword operates just like [\*randomize](#h.hy1j1pd6pyqp), except it keeps group numbers balanced. So if you are dividing people into groups A, B, and C, using \*experiment, the first participant to take your study could randomly be assigned any one of those. Let’s say they get assigned group B. The second participant could only be assigned A or C, since B was just taken. If they get assigned C, then the third participant will be in group A. The fourth participant once again has an equal chance of being assigned any one of the three groups. This ensures that the difference in the size of the groups is never more than 1.

\*experiment: My\_Experiment

\*group: GroupA

For the next 10 seconds, visualize yourself in a warm, happy place.

\*wait: 10s

\*group: GroupB

For the next 10 seconds, write about a warm, happy place.

\*question: My warm happy place is…

\*group: GroupC

For the next 10 seconds, visualize yourself in a cold, miserable place.

\*wait: 10s

Unlike \*randomize, it doesn’t make sense for \*experiment to choose more than one group. So, instead of using the inline text (here, "My Experiment") to specify how many things to randomize (e.g. \*randomize: 3, \*randomize: all), \*experiment uses it to name the experiment so that the CSV is much easier to read.

In most cases, when you’re running an actual experiment (where you want to see how a change in one thing in the program affects some other thing) you should use \*experiment. When you merely want things to be random (but you’re not conducting an experiment) \*randomize is probably the right choice.

## Going to Different Parts of the Program or to Other Websites

### *Going to Different Parts of the Program*

Sometimes, we may want the user to go immediately to other parts of the program, for example, depending on how they answer a given question. This can be done using the \*goto and \*label keywords. Note that \*label is different than \*tag, as \*label acts as a “road marker” or a flagged point at which to land, whereas \*tag “categorizes” something to recall later, similar to the “hashtag” (#) in many social media platforms. Take a look below.

\*label: woodchuckQuestion

\*question: How much wood can a woodchuck chuck?

\*question: Do you want to reconsider that answer?

Yes

\*goto: woodchuckQuestion

No

Fine then, moving on...

In this example, users who select “Yes” in the second question jump immediately to the label, and will once again see the woodchuck question. Users who select “No” will see the text “Fine then, moving on....” To make this happen, a \*label was placed above the woodchuck question. The purpose of a \*label is to specify a particular point in the program. A label’s name can have any combination of letters and/or numbers (though it *cannot* have spaces or special characters). Here, the \*label was given the name “woodchuckQuestion”. Once a user selects the “Yes” option, they reach the line with the \*goto keyword, which commands the program to go to the particular label that is specified. In this case, the \*goto keyword said “woodchuckQuestion,” sending the user immediately to the point in the program that has the label with that name.

### *Going Immediately to Other Websites*

You can use \*goto to automatically send your users away from your program and onward to a new website.

Why would you want to do this?

* At the end of your program, automatically send your users to your personal website.
* Create a tool that first understands your user's needs, then automatically sends them to a website that can help them.
* Send your users to other GuidedTrack programs (i.e. other run links). For instance, create a teaser program that describes the features of a much larger program (e.g. one that will take multiple weeks to complete). When the teaser ends and users decide they definitely want to try the rest, use \*goto to send them to a new program URL that requires them to [login](#h.7u2m0pg4bazr) before they can view more.
* Whatever else you can imagine!

The code for this is pretty simple. For example:

I'm done with you! Go to this website instead:

\*goto: http://www.guidedtrack.com

Users won't actually see the text in the example above (it'll only flash on the screen briefly). Instead, they will automatically be taken to the GuidedTrack website (in the same tab/window, not in a new tab/window). Note: whenever a \*goto keyword sends users to a new website, their current run in your program ends.

## Headers

Sometimes you may want to show the user large, header-sized text. To do so, use the \*header keyword.

\*header: This is some big, important text!

Don’t you agree?

In the above program, the user will see “This is some big, important text!” in large prominent font, with “Don’t you agree?” in normal font below it.

## If This… Then That

Sometimes you may want to display a part of the program to certain users. You can set what some of your users will see using the \*set and \*if keywords.

\*question: Do you prefer the mountains or the ocean?

Mountains

\*set: mountainlover

Ocean

\*set: beachbum

Want to know what I think?

\*if: mountainlover

Mountains are stupid.

\*if: beachbum

I love long walks on the beach!

In the program above, users will be asked their preference for the mountains or the ocean. By using the \*set keyword, you can remember this information about the user, and use it to personalize the program for them later on. In this example, users will see one of two phrases, depending on whether their preference was set as “mountainlover” or “beachbum.”

It’s important to note where the tabs are used in the above example. If you want to remember which multiple choice answer a user selected, you must indent the \*set keyword below their selection. Whenever you use the \*if keyword, you always indent what the user will see next on the line below the \*if keyword.

You can also use the \*if keyword with math operations:

* + addition
* - subtraction
* \* multiplication
* / division
* () parentheses to set operator preference
* > more than
* < less than
* >= more than or equal to
* <= less than or equal to
* = equals
* ^ to the power of
* and
* or
* not

Here’s an example using the [\*save keyword](#h.2r0dzihnrymi), the \*if keyword, and some math operations.

\*question: What number am I thinking of, between 1-10?

\*save: number

\*if: number = 5

Great job!

\*if: (number < 5) or (number > 5)

You’re wrong!

## Links

Links are easy to add. They require a couple brackets which are these things [ and ] and a fancily named key called a pipe, sometimes known as a bar, which looks like | that.

To remember all this stuff [click here|http://thoughtsaver.com] to create a ThoughtSaver account.

The words “click here” would be a clickable link to http://thoughtsaver.com, a fine site if you’ve never been. To recreate a link in your own program, begin with an open bracket immediately to the left of the text you want linked to. Once you have it all written out, add your pipe, then type the URL you’d like users to go to. At the end of the URL, place a close bracket.

If you'd like to bold, underline, or italicize your links, remember that you must have spaces before and after these indicators; here, that means:

[ \*click here\* |http://thoughtsaver.com] or [ \_click here\_ |http://thoughtsaver.com], not

[\*click here\*|http://thoughtsaver.com].

For a fancier way to automatically redirect users to a new website, see [this \*goto feature](#h.m4n4zixini54).

## Logging In

You can ask users to login or create an account to GuidedTrack. The benefits of doing so are to:

* Allow users to easily save their progress even when they return to your program from a different browser or computer.
* Track users who are completing a set of different programs (e.g. two parts to a questionnaire).

You can change the settings of your program so a login screen is either presented immediately when any user opens the run link to your program, OR, present a login screen somewhere in the middle of your program.

### *Adjust the settings to make the login screen appear in the beginning*

You can arrange it so that each and every time a user opens the run link to your program, they are encouraged (or required) to sign into GuidedTrack (if they are not already signed in).

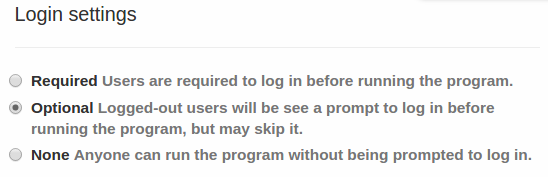
Once they sign in, they will be transported to the very last screen they visited in your program (even if they are accessing your program from a different computer or browser). If they have never taken your program before (or they fully completed their run last time), then they will start from the beginning of your program.

There are some caveats in which the login feature may be faulty if you make changes to a screen the user is on, so do be sure to see the "Important note" below before implementing this feature.

You can control whether or not users should login to your program from the "Settings" tab. To get there, click on a program from your "Programs" screen, then click "Settings."



From there, you'll see an option called "Login settings," with three options:



**Required:** Great if your program is highly complex, and users will need to take it in multiple sittings, and/or if the same users will be taking a bundle of your programs and you need to track them.

**Optional:** Nice if your program might take a while to finish and you'd like users to have the option of completing it in multiple sittings across devices, or if the program can be taken multiple times and you want to give users an option to store their past answers.

**None:** Best if your program is short and simple.

Of course, you may not always want the login screen to appear just in the beginning of the program. You may also want it to appear toward the end.

Using the [\*goto keyword](#h.h0tpsn9kjy3g) can also give you more options for introducing a login-required program.

**Important note:** It's risky to make changes to your program once you already have users who are actively taking it. Though improvements to this are in the works, currently if you make edits to a screen that one of your users has paused their progress on, when they return to the program they may need to start again from the beginning. This is because GuidedTrack is both trying to load the new changes and return the user to the position they were last on, which is problematic if their position has been deleted. Look for better solutions in the future, and for now, minimize changes to your program, consider instructing users to go to a safe spot in the program before exiting (such as a screen they can get to via the \*navigation menu that will never be updated), and consider adding code in the beginning of the program that can help users resume their spot quickly if they were bounced to the beginning.

### *Add a login screen in the middle or end of the program*

This option is especially useful if you've given users the option to login in the beginning of your program, they turned you down, and now you either want to give them a second chance or demand that they login. It’s also useful if users will be completing your program in one sitting, but you want them to login on the second screen or so in order to better track their responses (e.g. if they are taking a set of questionnaires from you).

To do that, you can throw a \*login keyword into your program wherever you'd like.

The basic syntax looks like this:

In order to take this super awesome program you must login to an existing account or create a new one.

\*login

\*required: yes

Great! Thanks.

In this program, users will see the first line of text. Then, they will see a “Next” button. After clicking the button, they will be presented with a popup asking them to login or sign up for a GuidedTrack account. Once they do either, they will see the last line of text, “Great! Thanks.”

In our example, the \*login keyword causes the popup screen to appear, but the additional \*required keyword, with its option of “yes” tells the program that logging in is required in order to continue with the program. You can also make it optional for users to login, as in our next example.

If you’d like to have the results of your “What type of exotic fruit are you?” quiz emailed to you, please login or create an account on the next screen. Otherwise, you’ll only be able to view the results in the program.

\*login

\*required: no

Thanks, here are your results...

As this example shows, you can set the \*required keyword to “no.” All users will be given the option to login, but whether they login or not, they will still make it to the next line of text, which begins with “Thanks.” By the way, if you’d like to know how to setup an email to send to your users (as mentioned in this example but not explicitly shown), then click here: [Emailing Users](#h.1mw275vkdtql).

One last thing; you don’t have to type in “\*required: no” in order to make the login optional. Simply writing \*login without the \*required keyword will produce the same result, giving users a non-mandatory opportunity to login if they desire.

Now’s your chance to login, if you’d like to.

\*login

Great, moving on.

If users decide to log into your program whether you require it or not, then their progress will be saved. However, you should also consider whether you need to ensure that the settings are set (as described above) so that a login screen pops up whenever users start your program. That way, they're immediately given an option to login and pick right back up from where they left off (i.e., if their progress from the last time they were in the program was saved, but they don't log in the next time they open the program, then it doesn't do them any good).

## Lists

There are three types of lists you can show to users. In a bulleted list, text is displayed using bullets. In a numbered list, text is displayed in a numbered fashion (i.e. 1., 2., 3., …). In an expandable list, you can have content that can hide or expand outward when clicked on.

### *Bullets*

Show the user a bulleted list using \*list, with the items you want to show indented beneath.

My favorite fruits:

\*list

apples

bananas

cars

In this example, the user sees the text “My favorite fruits” with

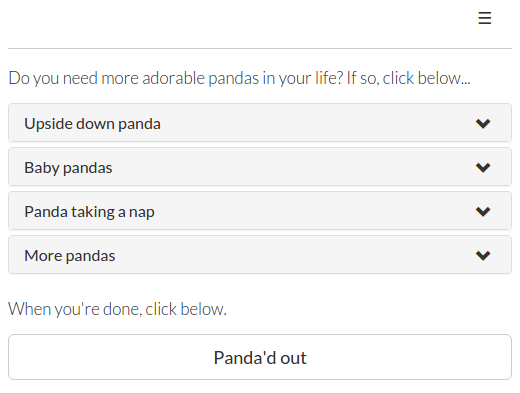
* apples
* bananas
* cars

as a bulleted list.

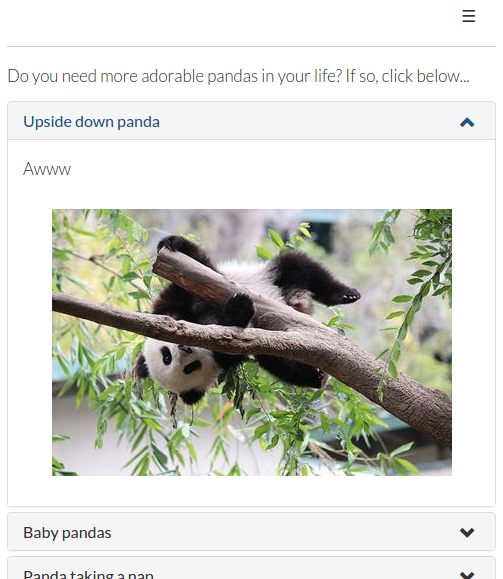
### *Expandable*

Do you have optional content that you want hidden or shown based on the user's whim? An FAQ, "learn more" section, list of definitions, help menu, or any other large block of content? If so, that's no problem with \*list: expandable. Users can open and close each item in your expandable list, which saves you from inundating them with a huge word wall or cute panda overload.

Here's how an expandable list looks when all its items are closed:



And here's how it looks when something is open:



The code is pretty simple. Type "\*list: expandable" and indent the names of each piece of content (for example, "Upside down panda"), and then indent whatever content you want to show up when users click that item. Here's the code from our example:

Do you need more adorable pandas in your life? If so, click below...

\*list: expandable

Upside down panda

Awww

\*image: http://i.imgur.com/rHutNnI.jpg?1

Baby pandas

Schmookum pies!

\*image: https://upload.wikimedia.org/wikipedia/commons/1/1b/Baby\_Pandas.JPG

Panda taking a nap

\*image: https://commons.wikimedia.org/wiki/Category:Gao\_Gao#/media/File:GaoGao03.jpeg

More pandas

Lots more pandas [here|https://goo.gl/ft7Wcl]!

When you're done, click below.

\*button: Panda'd out

Certain keywords cannot be included under expandable lists:

* \*button
* \*experiment
* \*goto
* \*label
* \*maintain
* \*navigation
* \*question
* \*program
* \*progress
* ... and other keywords that cause a page break, or affect the program on the whole

Text, images, and links are fair game! You can also nest other lists within expandable lists. Audio, videos, and charts do not work under expandable lists yet, but support for them is coming soon.

### *Numbered Lists*

Display a numbered list, using the \*list keyword, and the configuration “ordered.” Then type the things you want to show the user underneath.

How to eat breakfast:

\*list: ordered

Get out of bed

Walk into the living room

Open the fridge

Put something in your mouth

In this example, the user is shown “How to eat breakfast:,” then sees

1. Get out of bed
2. Walk into the living room
3. Open the fridge
4. Put something in your mouth

as a numbered list.

## Maintaining Text on the Screen

There may be instances when you want to keep some text on the screen, and not allow it to clear when a new question is introduced. To do this, you use the \*maintain keyword to keep the text maintained on the screen and the [\*clear keyword](#h.ixiznz3z534e) when you are ready to clear the text that is being maintained.

\*maintain: Make sure to answer honestly.

\*question: How are you?

\*question: Who do you love the most?

\*question: What’s a secret you’ve never told anyone?

\*clear

In this example, the user will see “Make sure to answer honestly.” Below that will be the first question, “How are you?” After the user answers, “Make sure to answer honestly” will remain, but the first question will disappear and the second question will appear below, and so on. After the last question, the entire screen will be cleared due to the \*clear keyword, so “Make sure to answer honestly” will then disappear.

In order to properly clear the maintained text, you must ensure both the \*maintain keyword and the \*clear keyword are immediately preceded by a question, a button, or a wait period. This means you purposefully time the maintained text and the clearing of that text to occur exactly when you intend them to, and neither will accidentally re-arrange or clear important information. Here’s another example:

Your exercise will begin when you click the button

\*button: Next

\*maintain: Remember to breathe deeply during this guided meditation.

The weather is beautiful and warm

\*button: Next

You are in a giant pool of jello

\*button: Next

You feel peaceful and at ease

\*button: Next

\*clear

That’s the end!

In the above example, the user sees the first line of text plus a button that says “Next.” After clicking the button, the user sees the maintained text at the top of the screen for the three screens that contain meditative imagery. After reading “You feel peaceful and at ease” and pressing the “Next” button, the maintained text is gone and the user only sees “That’s the end!”

It’s important here that both the \*maintained text and the \*clear keyword occurred after a button. If we hadn’t included a button after the text “Your exercise will begin when you click the button” the user would actually see the maintained text first with this text beneath it. If we hadn’t included a button after the text “You feel peaceful and at ease” the user would not actually see this line because it would have been cleared immediately as soon as the user got to that screen. The buttons ensure that the maintain and clear keywords are timed appropriately.

## Media

Programs can contain high-quality, embedded audio, images or videos.

### *Audio*

You can add audio files easily using the \*audio keyword.



**Add an audio file**

Here's the default way to add a file. Users will see an image like the one above and can click a play button to get your audio started.

\*audio: http://upload.wikimedia.org/wikipedia/commons/f/f8/Scaring\_a\_cat\_away.ogg

**Play your audio file automatically**

To have your audio file play automatically without the users needing to hit a play button, simply add \*start: yes beneath the file. It will start on its own.

\*audio: http://upload.wikimedia.org/wikipedia/commons/f/f8/Scaring\_a\_cat\_away.ogg

\*start: yes

**Hide your audio file**

You might not want your users to see an audio image like the one we showed you above. Simply type \*hide: yes to hide your file.

\*audio: http://upload.wikimedia.org/wikipedia/commons/f/f8/Scaring\_a\_cat\_away.ogg

\*start: yes

\*hide: yes

### *Images*

Suppose that you want to display an image. To do so, you must specify the location of where we can find that image file (e.g. the URL if it’s on the web, the link should end in .jpg or .png). So, to display an image, we’d use:

\*image: http://i.imgur.com/BZXv6eU.jpg

[▶Run](http://www.guidedtrack.com/programs/q9p2va1/run)

which would display to the user a ridiculously cute cat.

**Adding a Caption**

To give your image a caption, indent the \*caption keyword underneath, and add your desired text after a colon.

\*image: http://i.imgur.com/Yd5UYF1.jpg

\*caption: Isn’t it cute!

In this example, users will see adorable kittens, plus the caption “Isn’t it cute!” directly underneath.

**Where to Store Your Images**

We recommend you use [imgur](http://imgur.com) for storing your images. It’s free to store your images and you can get the .jpg and .png links that you need for GuidedTrack. [Flickr](http://www.flickr.com/) or some of the new photo-sharing sites like Google+ Photos, 500px, or Smugmug (a pay service) are other options that may suit your needs.

*Pros and Cons of imgur as of 8/2016*

* + Free
* + Easy to upload
* + Easy to edit
* + Easy to get photo links
* + Easy to navigate
* + Images are never deleted, unless deletion is requested
* +No limit to number of uploads
* - Large photos over 5 MB are compressed

### *GIFS*

Gifs can be displayed in GuidedTrack using the image function. When you find a GIF you'd like to use (if it's free for use--see our Guide to Using Images) simply right-click on the GIF at the location where it's stored and select "copy image location," and paste it in as you would an image location, with the \*image keyword.

**Using Symbols**

A number of symbols also work in GuidedTrack. For example, the symbols from this [list of miscellaneous symbols on Wikipedia](https://en.wikipedia.org/wiki/Miscellaneous_Symbols) all work wonderfully. Simply copy and paste the one you want into your program. (If you're writing your program in word, this same list is available under the subset "Miscellaneous Symbols.")

You might delight in dotting your program with little snowpeople, or find it useful to add checkmarks to multiple choice options after people complete a section of your program. You can even use the frowny face if you need to show your disappointment in your users. Of course, you can also add an [\*icon](#h.rbn29jup8a2e) to multiple choice or checkbox options as well.

### *Videos*

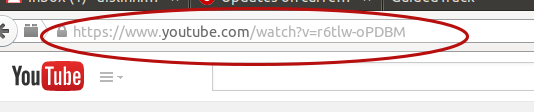
Videos can also be easily added if you have a youtube URL to link to. For example, you could show the user a video of some surfing dogs:

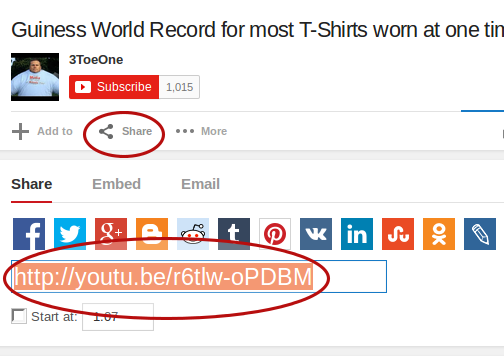
\*video: http://www.youtube.com/watch?v=Rg9lQoC8mkc

[▶Run](http://www.guidedtrack.com/programs/bi1e5ov/run)

This video would be displayed right on the page (i.e. it wouldn’t take them to a new page or pop up a window). The user would have to click the video’s play button to start the video.

To add a YouTube video to GuidedTrack, simply copy the video link from the address bar, or grab it from the share section. You can also customize the video so it starts at a certain time point or hides the video controls.





## Page Breaks

Often, the program content that is displayed to users will not be presented all at once. There are some instances in which a page break will occur automatically, requiring that users push a button to be shown the next segment of the program. Page breaks are caused automatically by questions and buttons.

### *Questions*

Each time you ask the user a question, the remaining content of your program will not be displayed until the question is answered.

\*question: Is a fork happier than a boat?

Thank you!

In this example, the users will see a question on their screen, but nothing below it. Once they answer the question, it, and anything else on the screen with it, will disappear. They will then see the text “Thank you!”

### *Buttons*

If you have a lot of text, you may want to break some of it up by adding a \*button keyword. The text of your button can be anything you like, such as “Next,” “Continue,” “Read More” or “Duck and cover!” and should be written as \*button: text. Here’s an example:

That was a lof of information! Let’s move onto the next lesson.

\*button: Yippee, let’s go!

All about ham sandwiches.

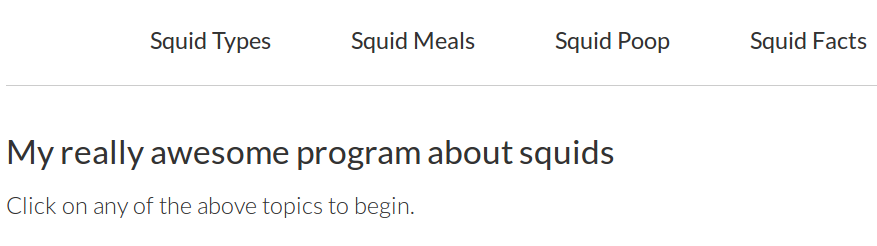
In this example, users will be shown the first sentence, then they will see the button “Yippee, let’s go!” Once clicked, the button and the first sentence will disappear, and the second sentence will be displayed.

### *The Clear Keyword*

The \*clear keyword does not force a button to appear on the page, though it is a method of removing old content that is on the screen to make way for new content. To learn more about the \*clear keyword, [click here](#h.ixiznz3z534e).

## Page Navigation

Let’s say you were bored one day so you wrote a GuidedTrack program entirely about squids. You can use the \*navigation keyword to provide permanent links at the top of your program that will allow users to jump to different sections of your work.



With navigation, your programs can be a lot more like full-fledged apps. Maybe you want to use \*navigation to help users access different programs of a comprehensive tool you’ve made, or revise their settings in your well-being program. There’s a lot you can do with navigation and adding it is easy. Here’s an example of how the squid code might look:

\*navigation

Squid Types

There’s like, a gazillion types of squids.

Squid Meals

Squids like to eat a lot of stuff. Big squids can eat whales. And people like to eat calamari.

Squid Poop

Like all cephalopods, squid have complex digestive systems. From the muscular stomach, the bolus moves into the caecum for digestion. Solid waste is passed out of the rectum. Beside the rectum is the ink sac, which allows a squid to rapidly discharge black ink into the mantle cavity.

Squid Facts

Giant squid have the largest eyes in the animal kingdom.

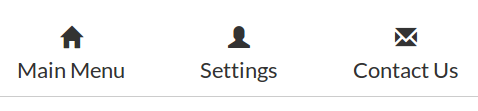
\*header: My really awesome program about squids

Click on any of the above topics to begin.

The program above begins with the navigation keyword, which will be visible immediately when the user starts the program. Each of the four navigation items are indented beneath the navigation keyword. Indented beneath each option is the content users would immediately see if they clicked on that option.

Once users are done viewing content from a navigation option, they return to the place they were at before they clicked the navigation.

Navigation content can include all other GuidedTrack keywords, not just text. For example, you can add [icons](#h.rbn29jup8a2e) to your navigation program, which could look like this:



You can even use the \*goto keyword to redirect users to other parts of your regular program, as this example does.

\*header: How to have a sexy mustache

\*button: Begin

\*navigation

Select a sexy stache style

\*goto: style

Eat the right foods that will grow your stache

\*goto: diet

Find the right accessories to accentuate your goods

\*goto: accessories

Learn to trim your stache

\*goto: trim

\*label: style

\*program: Stache Styles

\*label: diet

\*program: Stache Dietary Needs

\*label: accessories

\*program: Accessorizing Your Stache

\*label: trim

\*program: Keepin’ it Tidy

That’s the end! You can click on the links above to review past topics.

In this program, the navigation options don’t appear until the user clicks the “Begin” button and the navigation is activated. Once the user clicks “Begin,” the program called “Stache Styles” will also immediately start. Users can go through this program regularly and then begin the next programs in sequence. At any point, the users can also click one of the navigation options to easily review content they may have already seen or to jump ahead to new content that comes later in the sequence.

If at any particular point you want to hide the navigation, you can do so by typing: \*navigation: hide

STOPPED HERE

## Points

Points can be used just about anywhere in a program, but they’re especially useful to include after multiple choice answers. If you want to award or take away points for correct or incorrect answers, you can do so using the \*points keyword. The user will see them totaling up on their screen. Here’s an example of how this works.

\*question: How many licks does it take to get to the center of a tootsie pop?

1

Nice try.

252

\*points: 2

That’s right! You got two points.

1024

\*points: -1

Nope, sorry! You lost a point.

In this example, users who guess “1” won’t get any points, users who guess “252” will receive 2 points, and users who guess “1024” will have 1 point subtracted from their point total. The total points a user has will remain on their screen. The user always starts with 0 points, but points only begin to be displayed after hitting a \*points keyword for the first time.

You may want to assign lots of points in your program, but there may also be different types of points you want to assign. For example, let’s say you have a quiz with lots of real questions and lots of trick questions. You may want to know how many points the user got on the real questions versus how many they got on the trick questions.

\*question: If you have two apples and someone gives you two more, how many apples do you have altogether?

2

\*points: 0 realQuestion

4

\*points: 10 realQuestion

\*question: If there are three unicorns and you take one away, how many do you have?

1

\*points: 10 trickQuestion

2

\*points: 0 trickQuestion

In this example, separate totals will be calculated for the user’s realQuestion points and their trickQuestion points. However, unlike the first example, users will not see how many realQuestion or trickQuestion points they have. Only the total of unlabeled points are shown on the screen. There won’t be any data about points on the screen unless you explicitly tell the user how many points of a particular type they have. To do this, you’ll write something like this:

Congratulations! You got {realQuestion} points on all the real questions and {trickQuestion} points on all the trick questions. In total, you got {realQuestion + trickQuestion} points.

By putting the unique name for the type of point in braces { } you are able to tell the user how many points they’ve gotten for that type of point. You can also use math functions to add up the total number of points for the user.

You can assign some points with a \*type and some points without a \*type. If you do, the user will see the running tally of non-typed points on their screen, but any points with a \*type will not be added to that tally.

\*question: How many licks does it take to get to the center of a tootsie pop?

252

\*points: 2

1024

\*points: -1

\*question: If there are three unicorns and you take one away, how many do you have?

1

\*points: 10 trickQuestion

2

\*points: 0 trickQuestion

In the above example, users will be able to see new points added onto their screen when they answer the tootsie question, but their answer to the unicorn question will not affect the score on their screen because it has a type associated with it. You’ll have to tell the user how many trickQuestion points they got separately.

Points are not only valid beneath questions, though. You can merrily sprinkle the \*points keyword just about anywhere in your program.

For example…

You made it to screen 3,417 of my diary!?! You're super swell!

Here's 10,000 points!

\*points: 10000

The \*points keyword can also work anywhere in your program with special types of points (points that don't get visibly added on the corner of the screen, but are saved nonetheless).

You just completed the optional homework!

\*points: 10 brownie\_points

You can also set the user’s points to a specific value like this:

>>points = 100

You've automatically advanced to 100 points!

## 

## Progress Bars

You can mark your user's progress, adding a progress bar to the top of their screen. To do so is quite simple, with the \*progress keyword.

\*progress: 0%

Let’s begin!

\*button: Next

\*progress: 50%

Almost done.

\*button: Next

Done!

\*progress: 100%

In the above program, the progress bar will change from 0% to 50% to 100% as the user progresses through each screen of the program.

You control what the progress bar says at all times by simply typing "\*progress:" and the percent of the progress that is complete (from 0% to 100%).

When a user presses the back button or jumps to a different part of the program, you'll have to make sure the progress bars still work as intended. Progress bars are updated on a user's screen only when they pass a \*progress keyword in the program. It may be a good idea to have a \*progress keyword on every screen of your program (if you've enabled back buttons, so the progress gets updated when the user presses back) as well as just after every \*label keyword (so the user's progress is updated as they jump to new sections of the program.

You can also choose to hide a progress bar at a given moment. The keyword "\*progress: hide" will cause your progress bar to disappear until a user trips another one.

## Quitting a User’s Run

You can cause the program to end wherever and whenever you like. When inserting the \*quit keyword, users won't be able to see any line of text or keyword that may follow it.

For instance,

You’re reading a line of text.

\*quit

But you can’t see this second line of text because the program has quit.

Users taking this program will only be able to see the first line of text. The second will be invisible because nothing is ever displayed below a \*quit keyword that a user has hit.

The \*quit keyword may be especially useful in multiple choice options, where one option causes the program to end and the other allows it to continue. This example was used earlier in our introduction:

\*question: Which of these animals is least awesome?

Goat

So, you’re hating on goats

\*question: What have goats ever done to you?

Fish

Fish disfavorer! Get out of my face.

\*quit

Now for something completely different.

[▶Run](http://www.guidedtrack.com/programs/ceq3anv/run)

In this case, if the user selects the answer “Goat,” they’ll next see the text “So, you’re hating on goats” and then they’ll be asked the question “What have goats ever done to you?” Once the user answers the text “Now for something completely different.” will be displayed to them.

If instead of selecting “Goat” the user selected “Fish,” they would see the text “Fish disfavorer! Get out of my face.” and then the program would immediately (with no delay) stop running due to \*quit being reached. The phrase “Now for something completely different.” would not be displayed to the user, due to the \*quit keyword causing the program to exit this GT run immediately.

## Randomizing What is Displayed

There comes a time in every GT programmer’s life (that’s you by the way; you’re becoming a GT programmer as you read this) when you want to make a program vary each time it is run. The most basic way to accomplish this is with the \*randomize keyword (Note: if you are actually running an experiment or need to keep randomized groups of equal participant size, then the [\*experiment keyword](#h.5kw3jlihw299) may be a better fit for you).

Let’s take a look at \*randomize:

\*randomize

You STINK!

Go AWAY!

You couldn’t beat YOURSELF in a fight!

[▶Run](http://www.guidedtrack.com/programs/v53ewuj/run)

This is a great way to sling a randomly selected insult at the unsuspecting user. In this case, the user will see one (and only one) of the three insults.

What if you wanted to give them a double whammy though? The old one-two punch. You’d do it like this:

\*randomize: 2

You STINK!

Go AWAY!

You couldn’t beat YOURSELF in a fight!

[▶Run](http://www.guidedtrack.com/programs/ezq0zvj/run)

This means that rather than picking one insult to show, it will show two instead. As it stands, it will pick two *different* insults (i.e. it selects at random and without reusing each option, so that each option is used at most once).

### *Randomize All*

What if you want to show all of the items, but in a random order? Well, my friend, you’re in luck. Just use:

\*randomize: all

You STINK!

Go AWAY!

You couldn’t beat YOURSELF in a fight!

[▶Run](http://www.guidedtrack.com/programs/8bb8ww7/run)

### *Randomizing Every Time*

Once a user activates a \*randomize keyword, the program remembers the randomized selection and will re-display it whenever the user passes that same \*randomize again (i.e. if they press the back button or return to it via a \*goto keyword, they will see the same randomized thing they saw the first time). This makes sense, since a lot of the time you don't want the user to realize they're being shown random content.

There's a way to allow a \*randomize keyword to re-randomize \*everytime the user passes it.

\*label: beginning

\*randomize

\*everytime

Starfish

Catfish

Duckfish

\*question: Want to see another fish?

Yes

\*goto: beginning

No

With the addition of the \*everytime keyword, your randomized content will re-randomize. This will pave the way for new fishes for your user to enjoy. Every time.

Note: If you’d like to learn about randomizing answer options within a question, see the [\*shuffle keyword](#h.9tcq8ymt6zgq).

### *Embedded Expressions in \*randomize*

You can also make the number of randomizations a bit more personal by adding variables or mathematical expressions.

\*question: How many appetizers would you like on your menu?

1

2

3

4

\*save: number

\*randomize: {number}

pickled oysters, macomber turnip remoulade & toast

marinated fennel with feta & preserved lemon

gruyere stuffed black truffle gougères

chicken liver mousse, pickled onions & toast

confit duck leg, beet purée, frisée & walnut brittle

In the above program, the number of appetizers randomly shown will be the same number that the user asked to see.

You can also include mathematical expressions:

\*question: How many appetizers would you like on your menu?

1

2

3

4

\*save: number

\*randomize: {number+1}

pickled oysters, macomber turnip remoulade & toast

marinated fennel with feta & preserved lemon

gruyere stuffed black truffle gougères

chicken liver mousse, pickled onions & toast

confit duck leg, beet purée, frisée & walnut brittle

Although you asked for {number}, we gave you one bonus!

In this example above, the user will be shown one more appetizer than the one they asked for.

### *Randomizing a Group of Commands*

Let’s say you want to randomly show the user not just one command or line of program at a time, but multiple commands or lines of program code grouped together. You can do so by indenting two or more \*group keywords beneath the \*randomize keyword, and by placing your commands or text one indentation further beneath each group keyword. Here’s an example:

\*randomize

\*group

Hi!

This is great

\*group

Hey!

This is wonderful

In the example above, users will either see “Hi!” with “This is great” beneath it OR they will see “Hey!” with “This is wonderful” beneath it. The \*group randomize can get much more complicated than that. You can add sequences of commands, such as questions, as long as they are grouped together by being indented beneath each \*group.

### *Make Viewing Data Easier by Naming Randomizations*

Giving a proper name to each randomization or randomized \*group will greatly ease the process of analyzing your data. While the user won’t see the name of the group they were assigned to, you’ll be able to quickly see which users got assigned to which group names later on when you’re reviewing the results of your program. Take a look at the example.

\*randomize

\*name: intervention

\*group: jump\_pushups

Jump up and down 10 times!

Then do 5 pushups.

\*group: spin\_run

Spin in a circle 5 times!

Then run around for 10 seconds.

Now when viewing the data, you’ll see a column called “Randomize (intervention).” Beneath this column each run will have an entry of “jump\_pushups” or “spin\_run” depending on whether the user of that run was randomly assigned to the jumping and pushup group or the spinning and running group.

Since \*randomize can choose one or several things (e.g. “\*randomize: 2”), when that's the case the value for each run will be a comma-separated list of the sequence chosen, in the order it was chosen (e.g. “spin\_run, jump\_pushups”).

Without adding names to your randomizations, you’ll instead see 7-letter codes in the data. You would have to then figure out which randomization was assigned which 7-letter code. This can be tedious and difficult, so it’s best to name the randomization and groups when it’s important to know which randomization(s) a user received.

## Repeating

There are times when you may want to repeat some stuff for the user over and over again. You can save a bunch of typing if you use the \*repeat keyword. Here is how it works:

\*repeat: 6

All work and no play makes jack a dull boy

[▶Run](http://www.guidedtrack.com/programs/g2qqkei/run)

When run, this will display to the user six repetitions of the text, which will look like this:

All work and no play makes jack a dull boy

All work and no play makes jack a dull boy

All work and no play makes jack a dull boy

All work and no play makes jack a dull boy

All work and no play makes jack a dull boy

All work and no play makes jack a dull boy

Saves a lot of typing, doesn’t it? But text isn’t the only thing you can repeat. For instance:

\*repeat: 6

All work and no play makes jack a dull boy.

\*question: What do you think?

[▶Run](http://www.guidedtrack.com/programs/pzx6vqs/run)

This will display the text “All work and no play makes jack a dull boy” then ask the question “What do you think?”, wait for the user to respond, then display the text “All work and no play makes jack a dull boy” again, ask “What do you think?” again, wait for response, and repeat the whole process a total of 6 times. This is a great way to drive your user insane, especially if they are trapped with their family in a creepy hotel for the duration of the winter.

Here’s a more helpful example:

I’d like you to take a minute now to think of three things that you’re grateful for. This provides an immediate mood boost for many people. I’ll walk you through the process, step by step.

\*repeat: 3

\*question: What’s one thing you’re grateful for?

For the next few seconds, visualize what life would be like if you didn’t have this thing.

\*wait: 10 seconds

Now remind yourself how your life is improved by this thing that you’re grateful for.

\*wait: 10 seconds

[▶Run](http://www.guidedtrack.com/programs/u9trkbp/run)

This program, when run, will ask “What’s one thing you’re grateful for?” When you answer, it will say “For the next few seconds, visualize what life would be like if you didn’t have this thing.” Then, it will pause for 10 seconds (showing nothing else to the user during this time). Next it will say “Now remind yourself how your life is improved by this thing that you’re grateful for.” It will then pause for another 10 seconds. This whole process will repeat a total of 3 times.

Not only does the \*repeat keyword help prevent your hands from getting tired, but it makes it easier to make changes to your GT program later on. If you decide to write out repetitious stuff rather than using the repeat keyword, for instance using:

All work and no play makes jack a dull boy

All work and no play makes jack a dull boy

All work and no play makes jack a dull boy

and then you decide to make a change, as in:

All socks and no stares makes jack a sad clown

All socks and no stares makes jack a sad clown

All socks and no stares makes jack a sad clown

then you’ll have to make the change on all three lines (how annoying!). Instead, with the \*repeat keyword being used, you’d just have to change one line. A generally good principle in programming (whether it be in GT, or another programming language) is that you should reduce code redundancy as much as possible. That is, you should avoid having the same stuff written out multiple times, and instead find a way to just write it once while still getting the desired outcome (e.g. by using the repeat keyword). One reason this is a good idea is that it prevents mistakes that arise when you change a repeating element of your program in one place but forget to change it in all the other locations it appears.

The \*repeat keyword can also be used with variables. Let's take the following example:

\*question: How many times do you want to do this activity?

\*type: number

\*save: times

\*repeat: times

Take a deep breath in

\*button: Next

And take a deep breath out

\*button: Next

The end

In the example above, the user would be allowed to specify how many times they'd like to do the activity and consequently see the two pages of the activity: the first containing the text "Take a deep breath in" and a "Next" button and the second containing "And take a deep breath out" and a "Next" button. So, if users entered "2" in the question box, they would be instructed to take a deep breath in and out twice.

You can write any number expression after a \*repeat keyword to state how many times the indented content should be repeated (e.g. 3, a variable such as "times", points + 1, etc.).

## Repeating While a Given Condition is Being Met

The \*while keyword is the same as \*repeat, except that it allows the author to determine how many times to execute the code dynamically. So, instead of repeating something a specified number of times, it repeats it eternally \*while a given condition is being met. Let's illustrate that with an example:

If you can guess my name you’ll win $10,000,000.

>>answer = “wrong”

\*while: answer = “wrong”

\*question: What’s my name?

\*type: text

Rumpelstiltskin

>>answer = “correct”

\*other

That’s not my name. Try again.

You did it, you won the money!

In the above program, the user will see the text enticing them to guess the narrator's name. They will also see the question "What's my name?" If they guess the narrator might be named "Harold" they will see the text "That's not my name. Try again." Then, they'll once more be presented with the question "What's my name?" They can keep trying new names (Sally, Gilbert, Veronica), but so long as their answer is anything \*other than Rumpelstiltskin, they will repeatedly see the text telling them to try again and will repeatedly be asked "What's my name?" in a debilitating, never-ending cycle.

As soon as the user guesses "Rumpelstiltskin" (with appropriate caps and spelling), they will finally break free from this cycle and will at last be told they won the money.

So how does this work?

The program will execute the code indented beneath \*while so long as the variable "answer" is still being defined as the string "wrong". As an editor, you must create some kind of condition in which the value of that variable could potentially change so the user can continue on with the program.

Remember that with \*type: text questions, the user is presented with a box in which to write an answer, but as an editor you can also create customized content for specific answers they could potentially write. We created customized content for the answer "Rumpelstiltskin" as well as anything \*other than "Rumpelstiltskin".

Only if the user writes "Rumpelstiltskin" would the content *>>answer = "correct"* be activated by the user. Once that happens, this would change the variable "answer" from "wrong" to "correct". Only while "answer" is "wrong" would the question repeat.

(Note: instead of re-defining "answer" as "correct" we could have also written "you got it", "bingo" "183847438". So long as "answer" was anything other than "wrong", the cycle of questions would end).

Here’s one more example, using the [\*clear keyword](#h.ixiznz3z534e).

>>seconds = 10

\*while: seconds > 0

Countdown

{seconds}

\*wait: 1 sec

\*clear

>>seconds = seconds - 1

Blast off!

The above program is a countdown. The user will see the text "Countdown" with a number beneath it. To start, this number shown will be 10. Then there will be a wait of 1 second and the number shown will appear to change to 9, then another second wait and 9 will change to 8, and so on. The next-to-last thing users will see is the text "Countdown" and the number 1. Then, the screen will \*clear and they will see the words "Blast off!"

Here, \*while the variable "seconds" is greater than the number 0, the indented content will repeat indefinitely.

We know the content won't repeat indefinitely though, because the "seconds" variable is slowly decreasing by 1 each time the program passes this line: >>seconds = seconds - 1. And it passes this line exactly 10 times, until "seconds" = 0 and the \*while keyword no longer kicks in. That's when users will see "Blast off!".

Here's how the \*clear keyword comes into play. Without it, the user's screen would begin to look something like this:

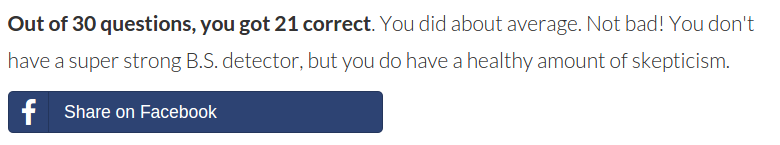


However, because we've added the \*clear keyword, the user's screen will clear each time they get to the \*clear keyword section of the \*while content. That means, the words "Countdown" as well as the number on display will disappear and will quickly be replaced by a new "Countdown" and a new number. Usually these clears happen very quickly though, so to the user, it will seem as though "Countdown" stays on the screen the entire time and only the numbers change.

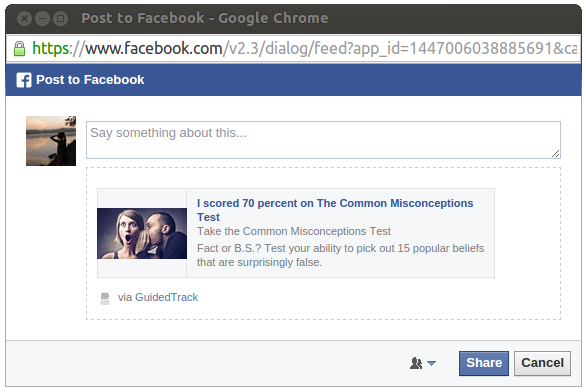
## Share Buttons

Users will be able to easily share your program via Facebook.

Here's how the button will look to users:



Here's what they'll see when they click the button:



And here's the sample code:

\*share

\*title: I scored {score} percent on The Common Misconceptions Test

\*picture: http://i.imgur.com/ANPnpdhl.jpg?1

\*description: Fact or B.S.? Test your ability to pick out 15 popular beliefs that are surprisingly false.

\*caption: Take the Common Misconceptions Test

These posts will automatically link to your program and can be customized with four pieces of information:

* \*description: (required) the text description of the program to be shared.
* \*title: (optional) the larger title that will display across the user's post.
* \*picture: (optional) the image that will appear in the post.
* \*caption: (optional) the gray text that is slightly larger than the description, but smaller than the title. It appears at the bottom of the post. If you don't include this keyword, it will automatically appear as the link the user is sharing.

A couple tips to keep in mind:

* You can use variables in any part of your share data.
* The ideal image size is 1200 x 630 pixels, or, a 1.9:1 apect ratio.
* Facebook doesn't know quite what to make of certain symbols, such as a %. These may need to be written out (e.g. writing "percent" instead of using a % sign), as we've done in our example.

## Using Collections

A collection is a grouping of items, all saved under the same variable name. You can create a collection from scratch like this:

>>myCollection = [“thing 1”, "thing 2", “thing 3”]

When checkbox responses are saved, they're saved in what's called a collection, which could look something like this:

["Dog", "Cat", "Hamster/other small furry thing"]

All you need to do to save checkbox responses into a collection is add the \*save keyword. For example, imagine we asked users to check off which pets they had and saved their responses as "pets."

\*question: Which of the following pets do you have?

\*type: checkbox

\*save:pets

Dog

Cat

Hamster/other small furry thing

Fish

Leopard, lion, panther

None of the above

You can use the "in" expression to provide special content to users who checked off a certain item. If we then wanted to give a follow-up question to users who had checked off "Cat," we could do the following:

\*if: "Cat" in pets

\*question: Are you a crazy cat person or just a normal cat person?

Normal

Crazy

Very crazy

Capitalization and punctuation are important when using "in" expressions. Writing "cat" in lowercase would not have worked.

Now, we're going to delve even deeper into collections. This functionality is essential for complex programs like the following:

* You want to store each answer that a user provided to a repeated question.
* You want to combine a user's answers from two or more questions into one variable.
* You need to alphabetize a list of things
* You need to get the mean of a list of things
* You need to know the size of a list of things

### *Getting and using a specific item in a collection*

Let's suppose you have a collection of numbers that looks like this:

>> mood\_scores = [6, 8, [4, 3, 5]]

We'll pretend these represent a person's mood scores (how they felt on a scale of 1-10) across time. Notice how collections can contain collections within them (as the third position in this collection does). For now, just imagine that the mood scores of the third position (4, 3, 5) are clumped together because they were all collected in the same day.

There are several ways the program can use collections to interact with the user. You can simply show the user their collection:

Your mood scores were {mood\_scores}.

This would produce a line of text that looked like this to the user:



You can also show the user specific entries in their entire collection. In our example, we want to show the user the first and second things in their collection:

Your first ever mood score was {mood\_scores[1]}.

Your second score was {mood\_scores[2]}.

Users would see the following:



You can also show users a specific entry of a collection within a collection:

Yesterday, you said {mood\_scores[3][1]}, then {mood\_scores[3][2]}, and finally {mood\_scores[3][3]}.

With the above code, users would see the following:



Collection entries can be used similarly to other variables. For example, you can do the following things:

\*if: mood\_scores[1] >= 7

You started with a pretty good mood.

\*repeat: mood\_scores[2]

:-)

In the first example, if user's first mood score in their collection were a 7 or above, they'd see "You started with a pretty good mood." In the second example, users would see as many smiley faces as their mood rating. In this case, they'd see 8 smiley faces.

### *Get the size of a collection*

Getting the size of a collection, meaning the number of entries, is pretty easy. Using our earlier pet example, we could say:

You have {pets.size} different kinds of pets.

This would then say "You have 3 different kinds of pets" because our pets collection contained a dog, cat, and a small furry thing.

### *Get the mean value of a collection of numbers*

If you have a collection of numbers, such as [10, 10, 30, 30], you can easily find the mean of the collection, like this:

The mean of these values is: {yournumbers.mean}.

This would read as "The mean of these values is: 20."

### *Add something to the end of a collection*

Adding new items to a collection is pretty easy. If you have a collection called "user\_badges," you could add a new entry to the collection like so:

>>user\_badges.add("Topped 1,000 points")

The new entry, "Topped 1,000 points," would be added to the end of the collection called user\_badges.

Now let's use a more complicated example about a question that asks users to painstakingly enter 50 hobbies. Here, the collection is initially empty.

>>hobbies=[]

\*repeat: 50

\*question: Enter one of your hobbies.

\*save: newHobby

>>hobbies.add(newHobby)

Great! Now add another

Here are the hobbies you listed: {hobbies}

First, we have to set up the collection that the user's answers will go into. We've written >>hobbies=[ ] at the top. Initially, it's an empty collection, but it will soon fill up with user data.

Users will next see "Enter one of your hobbies" and can enter whatever they like in the box. Their answer is saved as the variable "newHobby." Then, this variable is added to the end of the collection called hobbies. Since the collection is empty in the beginning, the first answer users provide will momentarily be the only thing in the collection. The user will next see "Great! Now add another" and the question will reappear. Users will repeat the process of typing in each new hobby a total of 50 times. Each time, the variable "newHobby" will change to the latest thing they wrote and this thing will be added to the end of the user's collection. When users are finished, they will see "Here are the hobbies you listed:" followed by all 50 of the hobbies they typed.

You'll notice that in the user\_badges example what's called a "text string" was added to the collection ("Topped 1,000 points") and in the second example the value of a variable was added (newHobby). In all aspects of GuidedTrack, there are three different types of information you can store and work with: text strings, numbers, and collections. There are also variables, which store any of the three types. Here's how a text string, number, and collection looks:

>>hobbies.add("lawn mowing")

>>hobbies.add(2048)

>>hobbies.add([newHobby, "lawn mowing", 2048])

Here's how a variable looks:

>>hobbies.add(newHobby)

### *Insert something in a specific point in the collection*

When you add something to a collection, it gets added to the very end of the collection. But what if you want to insert something at a very specific spot in the collection? You'd do so like this:

>>hobbies.insert("lawn mowing", 3)

This inserts "lawn mowing" as the 3rd thing in the hobbies collection. Within the parentheses, the first part is always the thing you want to insert. The second part is the position that you want to insert the thing in.

In this next example, the user's answer has been added to the beginning of their collection.

\*question: What's your favorite hobby?

\*save: favorite

>>hobbies.insert(favorite, 1)

Your favorite hobby is listed first: {hobbies}

### *Sort a collection*

It's pretty easy to sort a collection so that all the entries are in increasing order (a, b, c...1, 2, 3). Or, in decreasing order (9, 8, 7...z, y, x)

>>hobbies.sort("increasing")

Here are your hobbies alphabetized: {hobbies}

>>hobbies.sort("decreasing")

Here are your hobbies listed from z to a: {hobbies}

Sort is also very handy for collections of numbers. It can be used to put all the numbers in either increasing or decreasing numerical order.

### *Randomize a collection*

Similarly, you can sort the items in a collection so they're in a completely random order:

>>hobbies.randomize

Here are your hobbies in a totally random order: {hobbies}

### *Combine two or more collections*

Let's say you have two collections and you want to combine them into one collection. We'll use these two as an example:

>>listA=[5, 8, 22]

>>listB=[7, 23, 6, 4]

In order to add the items of listB into listA, you'd use the following code:

>>listA.combine(listB)

Your list: {listA}

In the above example, listB would still contain its original 4 items, but listA would now have 7 items (its original 3, plus the 4 from listB). If you were to show listA to users it would look like this:



Let's imagine the variable listB didn't actually exist; you just had some data points. You can still add these new numbers to listA like so:

>>listA.combine([7, 23, 6, 4])

## Using Multiple GuidedTrack Programs

Sometimes it is easiest to use multiple GT programs that refer to each other. Let’s say you have a program called “How to take a bath” and another program called “How to wash behind your ears.” You may want your “How to take a bath program” to also contain instructions for washing behind the ears. But rather than rewrite some of the program, or copying and pasting one into the other, you can run the “How to wash behind your ears” program directly from the “How to take a bath” program. (This has the advantage of adhering to the "reduce redundant code" principle we mentioned above.) Here’s an example:

After you’ve sufficiently scrubbed your cheeks, you’ll need to work on your ears. Naturally, the most important section is the bit behind the lobes.

Now, I’m going to teach you about how to wash behind your ears.

\*program: How to wash behind your ears

Great, now that we’re done with your ears, grab some shampoo and get ready to lather those eyebrows.

In the above example, users will see the text “After you’ve...” and immediately following it on the same screen they will also see whatever is contained within the “How to wash behind your ears” program. After this program ends, the users will then be able to see “Great, now that we’re done with your ears...”

## Using Variables

You may want to save yourself some typing by using variables. For example, pi = 3.14159, right? If for some reason you need to use pi a lot in your program, you can set the variable like this:

>> pi = 3.14159

Setting a variable simply means adding two >> angle brackets to the program, then writing the name of the variable you want to set (e.g. “pi”) and the value of that variable. Let’s go back to our previous example now.

>> pi = 3.14159

The value of pi is {pi}.

To use a variable, you can simply put that variable in {braces}. **Variables are case sensitive**. In our example, you’d have to write {pi}; {PI} would not work. With this code, your users would see the line “The value of pi is 3.14159.”

The value of a variable can be an equation (e.g. area = height \* width). If your variable is an equation, then any other variables used must have already been saved (e.g. in the example >> area = height \* width, height and width must have already been previously saved).

>> pi = 3.14159

\*question: What is the radius of your circle?

\*save: r

>>area = pi\*r\*r

The area of your circle is {area}.

In this example, users will see a question asking them to list the radius of their circle. After they answer, their answer will be saved as “r”. If users had entered “2” as their answer, they will then see the line “The area of your circle is 12.57.”

Note: in this example, it is important to set the “area” variable after saving “r”. If you had tried to set the “area” variable before the question, this program would not have worked.

You can change the value of a variable. Once a variable is set, you can always modify it later, as in this example.

>>scoops = 5

\*question: You have {scoops} of ice cream. How many will you eat right now?

2

>>scoops = scoops - 2

5

>>scoops = scoops - 5

From now on, you have {scoops} scoops of ice cream.

If users select to eat 2 scoops of ice cream now, then the variable will be changed as follows: >>scoops = scoops - 2. This equation is essentially saying: the new number of scoops = the prior number of scoops (which is 5) - 2. Users will thus see the line, "From now on, you have 3 scoops of ice cream." If users select to eat 5 scoops of ice cream now, then they will have 0 scoops of ice cream at the end of the program.

The value of a variable can also be a word, as this example shows.

\*question: Would you like to read a story about a girl or a boy?

Girl

>>characterName = "Stacey"

Boy

>>characterName = "Brian"

Once upon a time, {characterName} got lost in the woods and eaten by dinosaurs. The end.

In this example, the user is able to choose whether they’d like to read a story about a girl or a boy. Depending on what they select, the program will either set the “characterName” variable as “Stacey” or “Brian.” Note, it’s important that you put the value of text variables in “quotes,” otherwise the program won’t work (it will think the value is actually a variable itself). Users who selected girl will next see the line, “Once upon a time, Stacey got lost in the woods and eaten by dinosaurs. The end.”

Finally, you can use variables to further manipulate [points](#h.ksh6vnvmus8d). You can automatically give users points, or modify their existing points.

You have points!

>>points = 100

In this example, users will see an animation and 100 points added to their tally on their screen. They will also see the line “You have points!”

If the user already has points, then it is not necessary to write “>>points = 100” (or another amount other than 100). However, you could do this if you wanted to the user to have a point tally that you set and control. The below example shows another method of manipulating points.

\*question: Click the number of points you want

50 50

\*points: 50

100 100

\*points: 100

1,000 1,000

\*points: 1000

Oh no, you were attacked by a sea monkey and just got robbed of half your points!

>>points = points \* .5

## Waiting a Specified Amount of Time

Suppose that you wanted to create a pause so that some time lapses between when one thing is shown to the user and when the next thing is displayed. You could do that easily using a program like this one:

Take a few seconds now to solve all your problems.

\*wait: 5 seconds

Times up!

[▶Run](http://www.guidedtrack.com/programs/1p5831m/run)

This program would display “Take a few seconds now to solve all your problems.” Then, nothing would happen for 5 seconds. After the 5 seconds, “Times up!” would be displayed. The \*wait keyword can be useful if you want to make sure the user takes a few seconds to think before moving on, or if you want to show some text, and give the user time to read it, before you display more text. You can designate the amount of time you’d like the program to wait by either writing out the words “seconds” and/or “minutes,” OR you can just use the letters s (for seconds) or m (for minutes), such as:

\*wait: 5s

You can also pause your program long enough for all the data the user has entered to sync up with the system. Data is stored periodically throughout your user's run anyway, but this keyword may be useful in spots where users may close out of your program too soon. This allows you to ensure, at any point in the program, that the data is totally synced.

I'm crunching all of your answers.

Please wait just a moment.

\*wait: data

Thanks!

## Special Settings

### *Back Buttons*

By default, there are no back buttons on GT programs, so users cannot return to a previously cleared screen. However, back buttons can be added using a special setting.

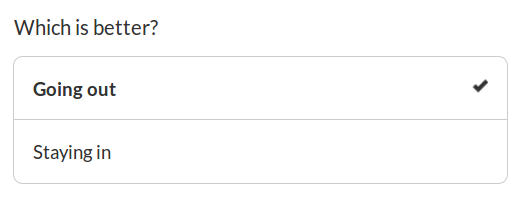
To add back buttons for the entire program, use the following command in the beginning of the program:

\*settings

\*back: yes

This will allow back buttons to be a feature on all the screens of your program, excluding the first page, which doesn’t need a back button.

When back buttons are pressed, users will be able to see the original answers they gave to past questions and they can edit those answers. This will make editing paragraph responses a breeze for users. This feature works whether the question type is text, paragraph, multiple choice, or checkbox.



In the above multiple choice example, this user can see their original answer was "Going out" and can change it or keep it by pressing either option.

Now, you may want to turn off the back button feature for a particular screen, or if you didn’t use the back button setting, you may have a particular screen that you would nevertheless like users to be able to go back to. Rest assured, the back button keyword can also be added to individual items like questions to turn back buttons on or off in that specific case.

When adding the back button feature to a question, you are indicating whether or not it’s okay for users to go back to re-enter their answer to that question.

\*question: How are you?

\*back: no

Glad to hear it!

In this example, users will answer the question, then see the text “Glad to hear it!” Once they see the text though, they will not be able to go back to change their answer, even if back buttons are turned on for the whole program using \*settings.

\*question: How are you?

\*back: yes

Glad to hear it!

In this example, users will answer the question, then see the text “Glad to hear it!” Once they see this text, they will also see a small back button, which will allow them to return to the question and re-enter an answer, even if back buttons are turned off for the whole program. In other words, the \*back setting you use on a specific question overrides (for that specific question) what has been set for the entire program.